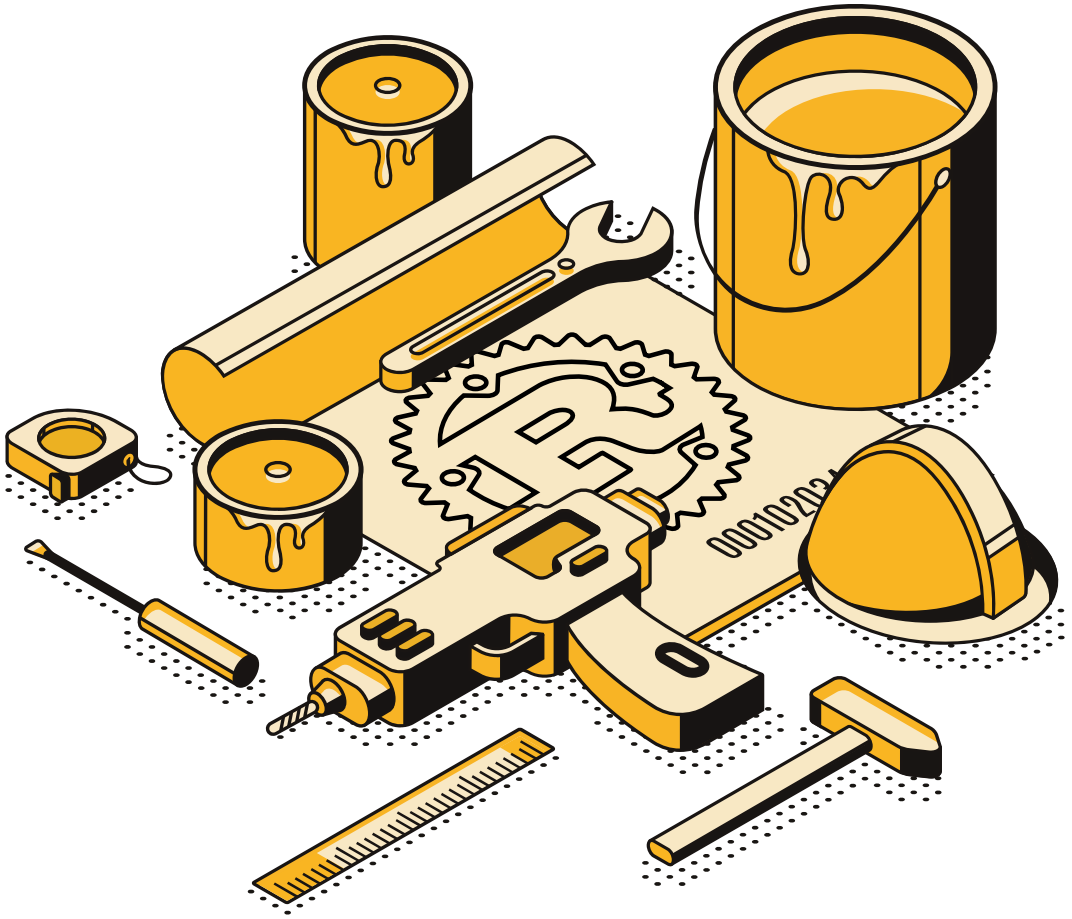
corrode

RUST INSIGHTS

TIPS FOR FASTER RUST COMPILE TIMES

Last updated: 2025-05-04



Slow Rust Builds?

Here are some tips to speed up your compile times. This list was originally released on my private blog, but I decided to update it for 2025 and move it here.

All tips are roughly ordered by impact so you can start from the top and work your way down.

TABLE OF CONTENTS

GENERAL TIPS

UPDATE THE RUST COMPILER AND TOOLCHAIN

Make sure you use the latest Rust version:

```
rustup update
```

Making the Rust compiler faster is an ongoing process. Thanks to their hard work, compiler speed has improved 30-40% across the board year-to-date, with some projects seeing up to 45%+ improvements. It pays off to keep your toolchain up-to-date.

USE CARGO CHECK INSTEAD OF CARGO BUILD

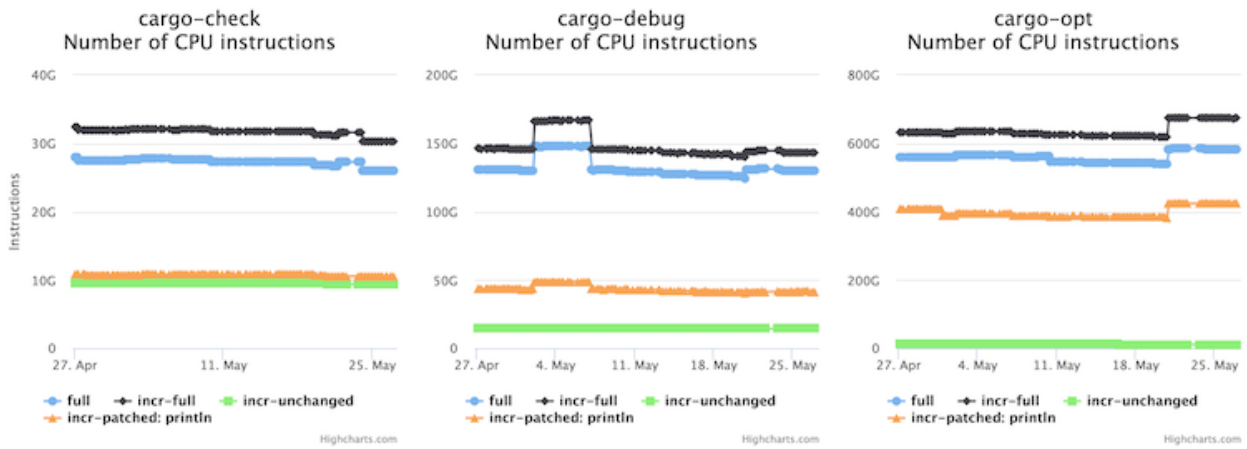
```
# Slow 🐢
cargo build

# Fast 🐇 (2x-3x speedup)
cargo check
```

Most of the time, you don't even have to *compile* your project at all; you just want to know if you messed up somewhere. Whenever you can, **skip compilation**

altogether. What you need instead is laser-fast code linting, type- and borrow-checking.

Use `cargo check` instead of `cargo build` whenever possible. It will only check your code for errors, but not produce an executable binary.

Consider the differences in the number of instructions between `cargo check` on the left and `cargo debug` in the middle. (Pay attention to the different scales.)



A sweet trick I use is to run it in the background with `cargo watch`. This way, it will `cargo check` whenever you change a file.

Bonus: Use `cargo watch -c` to clear the screen before every run.

REMOVE UNUSED DEPENDENCIES

```
cargo install cargo-machete && cargo machete
```

Dependencies sometimes become obsolete after refactoring. From time to time it helps to check if you can remove any unused dependencies.

This command will list all unused dependencies in your project.

```
Analyzing dependencies of crates in this directory...
cargo-machete found the following unused dependencies in <project>:
crate1 -- <project>/Cargo.toml:
        clap
crate2 -- <project>/crate2/Cargo.toml:
        anyhow
```

```
async-once-cell
dirs
log
tracing
url
```

More info on the cargo-machete project page.

UPDATE DEPENDENCIES

1. Run `cargo update` to update to the latest semver compatible version.
2. Run `cargo outdated -wR` to find newer, possibly incompatible dependencies. Update those and fix code as needed.
3. Run `cargo tree --duplicate` to find dependencies which come in multiple versions. Aim to consolidate to a single version by updating dependencies that rely on older versions. (Thanks to /u/dbdr for pointing this out.)

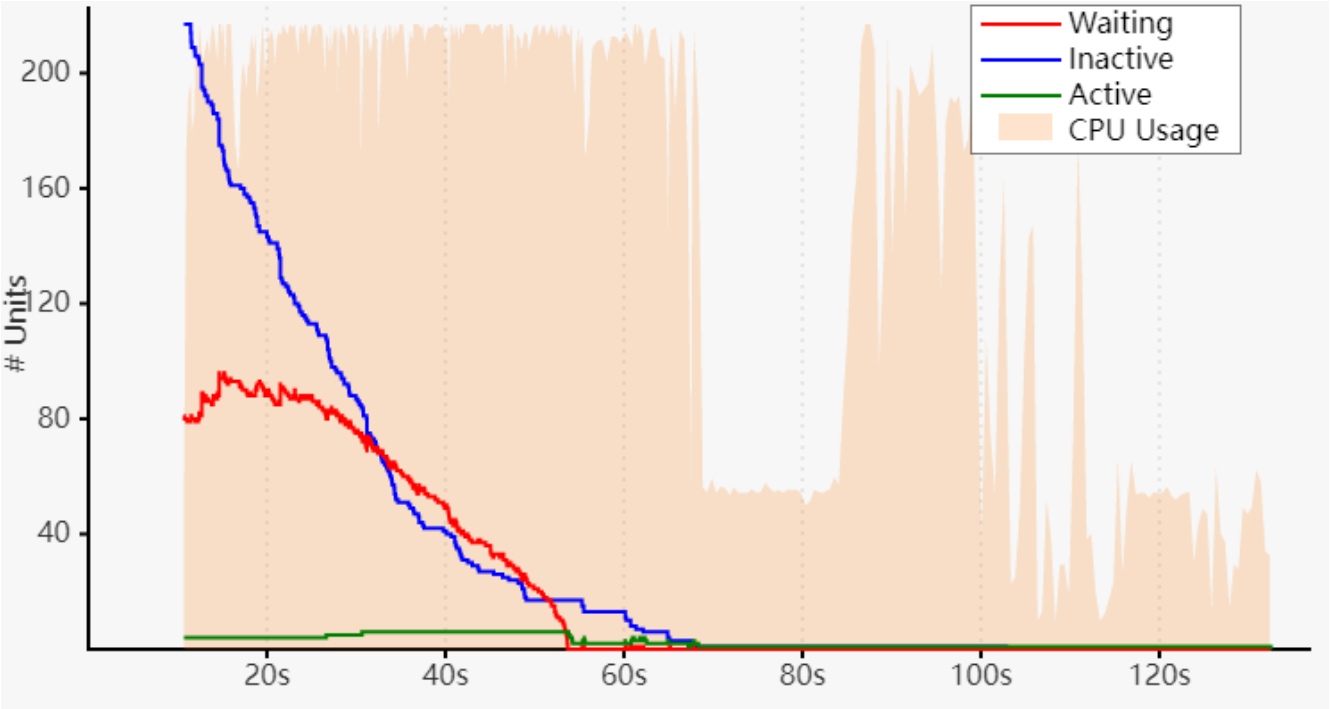
(Instructions by /u/oherrala on Reddit.)

On top of that, use `cargo audit` to get notified about any vulnerabilities which need to be addressed, or deprecated crates which need a replacement.

FIND THE SLOW CRATE IN YOUR CODEBASE

```
cargo build --timings
```

This gives information about how long each crate takes to compile.



The red line in this diagram shows the number of units (crates) that are currently waiting to be compiled (and are blocked by another crate). If there are a large number of crates bottlenecked on a single crate, focus your attention on improving that one crate to improve parallelism.

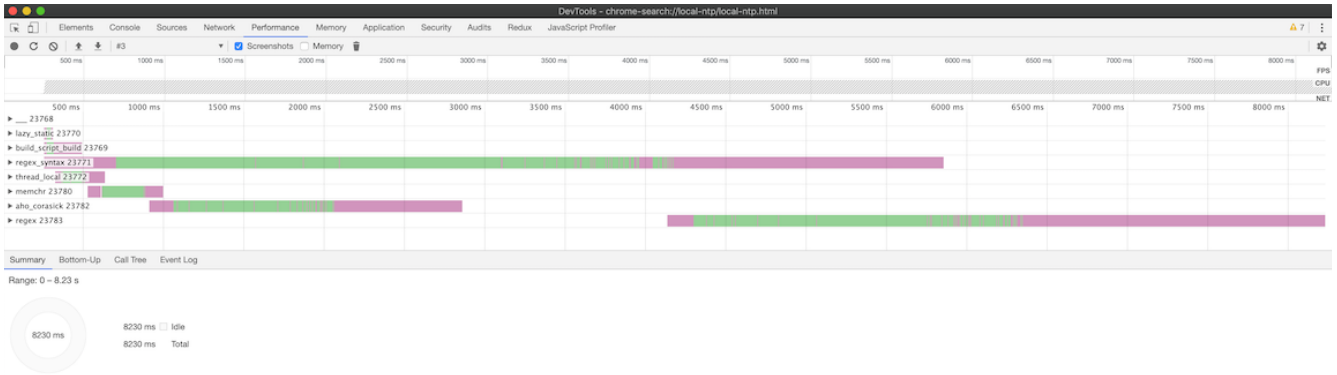
The meaning of the colors:

- *Waiting* (red) — Crates waiting for a CPU slot to open.
- *Inactive* (blue) — Crates that are waiting for their dependencies to finish.
- *Active* (green) — Crates currently being compiled.

More info in the documentation.

PROFILE COMPILE TIMES

If you like to dig deeper than `cargo --timings`, Rust compilation can be profiled with `cargo rustc -- -Zself-profile`. The resulting trace file can be visualized with a flamegraph or the Chromium profiler:



Another golden one is `cargo-llvm-lines`, which shows the number of lines generated and the number of copies of each generic function in the final binary. This can help you identify which functions are the most expensive to compile.

```
; cargo llvm-lines | head -20

  Lines         Copies          Function name
  -----         ------          -------------
  30737 (100%)   1107 (100%)   (TOTAL)
   1395 (4.5%)     83 (7.5%)   core::ptr::drop_in_place
    760 (2.5%)      2 (0.2%)   alloc::slice::merge_sort
    734 (2.4%)      2 (0.2%)   alloc::raw_vec::RawVec<T,A>::reserve_internal
    666 (2.2%)      1 (0.1%)   cargo_llvm_lines::count_lines
    490 (1.6%)      1 (0.1%)   <std::process::Command as cargo_llvm_lines::PipeTo>::p
    476 (1.5%)      6 (0.5%)   core::result::Result<T,E>::map
    440 (1.4%)      1 (0.1%)   cargo_llvm_lines::read_llvm_ir
    422 (1.4%)      2 (0.2%)   alloc::slice::merge
    399 (1.3%)      4 (0.4%)   alloc::vec::Vec<T>::extend_desugared
    388 (1.3%)      2 (0.2%)   alloc::slice::insert_head
    366 (1.2%)      5 (0.5%)   core::option::Option<T>::map
    304 (1.0%)      6 (0.5%)   alloc::alloc::box_free
    296 (1.0%)      4 (0.4%)   core::result::Result<T,E>::map_err
    295 (1.0%)      1 (0.1%)   cargo_llvm_lines::wrap_args
    291 (0.9%)      1 (0.1%)   core::char::methods::<impl char>::encode_utf8
    286 (0.9%)      1 (0.1%)   cargo_llvm_lines::run_cargo_rustc
    284 (0.9%)      4 (0.4%)   core::option::Option<T>::ok_or_else
```

REPLACE HEAVY DEPENDENCIES

From time to time, it helps to shop around for more lightweight alternatives to popular crates.

Again, `cargo tree` is your friend here to help you understand which of your dependencies are quite *heavy*: they require many other crates, cause excessive network I/O and slow down your build. Then search for lighter alternatives.

Also, `cargo-bloat` has a `--time` flag that shows you the per-crate build time. Very handy!

Here are a few examples:

Crate	Alternative
serde	miniserde, nanoserde
reqwest	ureq
clap	lexopt

Here's an example where switching crates reduced compile times from 2:22min to 26 seconds.

SPLIT BIG CRATES INTO SMALLER ONES USING WORKSPACES

Cargo has that neat feature called workspaces, which allow you to split one big crate into multiple smaller ones. This code-splitting is great for avoiding repetitive compilation because only crates with changes have to be recompiled. Bigger projects like servo and vector make heavy use of workspaces to reduce compile times.

DISABLE UNUSED FEATURES OF CRATE DEPENDENCIES

`cargo-features-manager` is a relatively new tool that helps you to disable unused features of your dependencies.

```
:argo install cargo-features-manager
:argo features prune
```

From time to time, check the feature flags of your dependencies. A lot of library maintainers take the effort to split their crate into separate features that can be toggled off on demand. Maybe you don't need all the default functionality from every crate?

For example, `tokio` has a ton of features that you can disable if not needed.

Another example is `bindgen`, which enables `clap` support by default for its binary usage. This isn't needed for library usage, which is the common use-case. Disabling that feature improved compile time of rust-rocksdb by ~13s and ~9s for debug and release builds respectively. Thanks to reader Lilian Anatolie Moraru for mentioning this.

> **Fair Warning**
>
> It seems that switching off features doesn't always improve compile time. (See tikv's experiences here.) It may still be a good idea for improving security by reducing the code's attack surface. Furthermore, disabling features can help slim down the dependency tree.

You get a list of features of a crate when installing it with `cargo add`.

If you want to look up the feature flags of a crate, they are listed on docs.rs. E.g. check out tokio's feature flags.

After you removed unused features, check the diff of your `Cargo.lock` file to see all the unnecessary dependencies that got cleaned up.

ADD FEATURES FOR EXPENSIVE CODE

```
features]
 Basic feature for default functionality
efault = []

 Optional feature for JSON support
son = ["serde_json"]

 Another optional feature for more expensive or complex code
```

```
:omplex_feature = ["some-expensive-crate"]
```

Not all the code in your project is equally expensive to compile. You can use Cargo features to split up your code into smaller chunks on a more granular level than crates. This way, you can compile only the functionality you need.

This is a common practice for libraries. For example, `serde` has a feature called `derive` that enables code generation for serialization and deserialization. It's not always needed, so it's disabled by default. Similarly, `Tokio` and `reqwest` have a lot of features that can be enabled or disabled.

You can do the same in your code. In the above example, the `json` feature in your `Cargo.toml` enables JSON support while the `complex_feature` feature enables another expensive code path.

CACHE DEPENDENCIES WITH SCCACHE

Another neat project is sccache by Mozilla, which caches compiled crates to avoid repeated compilation.

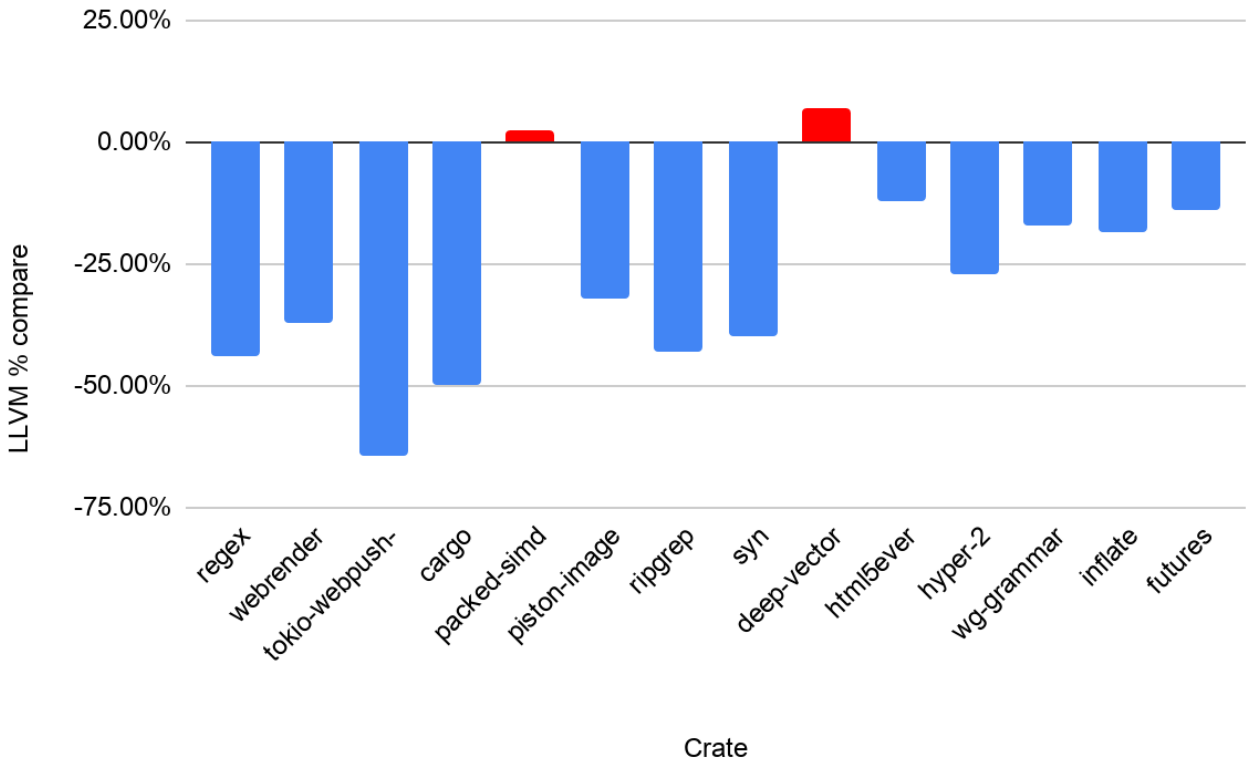
I had this running on my laptop for a while, but the benefit was rather negligible, to be honest. It works best if you work on a lot of independent projects that share dependencies (in the same version). A common use-case is shared build servers.

CRANELIFT: THE ALTERNATIVE RUST COMPILER

Did you know that the Rust project is using an alternative compiler that runs in parallel with `rustc` for every CI build?

rustc_codegen_cranelift, also called `CG_CLIF`, is an experimental backend for the Rust compiler that is based on the Cranelift compiler framework.

Here is a comparison between `rustc` and Cranelift for some popular crates (blue means better):



The compiler creates fully working executable binaries. They won't be optimized as much, but they are great for local development.

A more detailed write-up is on Jason Williams' page, and the project code is on Github.

SWITCH TO A FASTER LINKER

What is a linker?

A linker is a tool that combines multiple object files into a single executable.
It's the last step in the compilation process.

You can check if your linker is a bottleneck by running:

```
:argo clean
:argo +nightly rustc --bin <your_binary_name> -- -Z time-passes
```

It will output the timings of each step, including link time:

```
...
:ime:   0.000   llvm_dump_timing_file
:ime:   0.001   serialize_work_products
:ime:   0.002   incr_comp_finalize_session_directory
:ime:   0.004   link_binary_check_files_are_writeable
:ime:   0.614   run_linker
:ime:   0.000   link_binary_remove_temps
:ime:   0.620   link_binary
:ime:   0.622   link_crate
:ime:   0.757   link
:ime:   3.836   total
    Finished dev [unoptimized + debuginfo] target(s) in 42.75s
```

If the `link` step is slow, you can try to switch to a faster alternative:

Linker	Platform	Production Ready	Description
`lld`	Linux/macOS	Yes	Drop-in replacement for system linkers
`mold`	Linux	Yes	Optimized for Linux
`zld`	macOS	No (deprecated)	Drop-in replacement for Apple's `ld` linker

MACOS ONLY: FASTER INCREMENTAL DEBUG BUILDS

Rust 1.51 added a flag for faster incremental debug builds on macOS. It can make debug builds multiple seconds faster (depending on your use-case). Some engineers report that this flag alone reduces compilation times on macOS by **70%**.

Add this to your `Cargo.toml`:

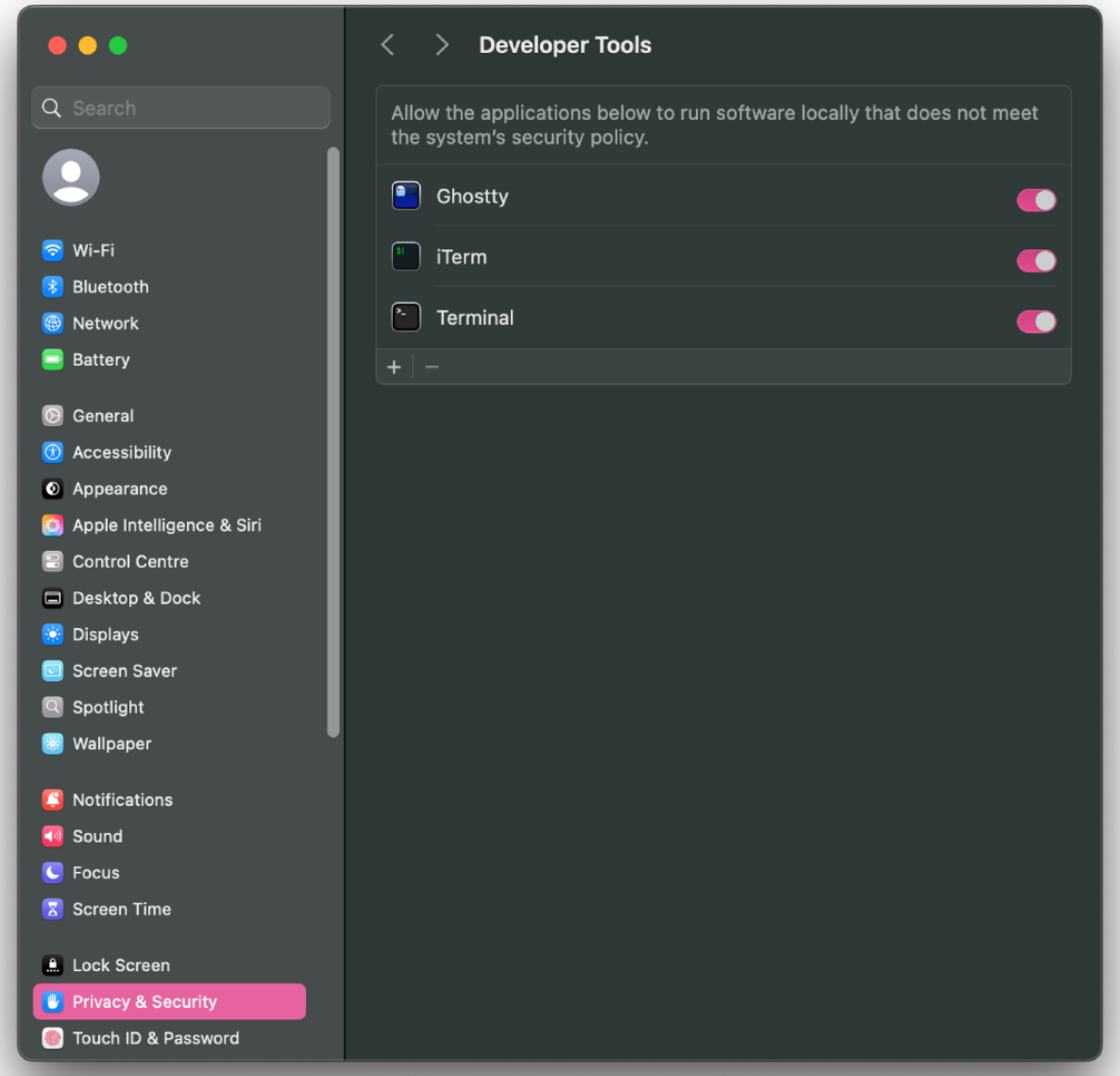
```
profile.dev]
plit-debuginfo = "unpacked"
```

The flag might become the standard for macOS soon. It is already the default on nightly.

MACOS ONLY: EXCLUDE RUST COMPILATIONS FROM GATEKEEPER

Gatekeeper is a system on macOS, which runs security checks on binaries. This can cause Rust builds to be slower by a few seconds for each iteration. The solution is to add your terminal to the Developer Tools, which will cause processes run by it to be excluded from Gatekeeper.

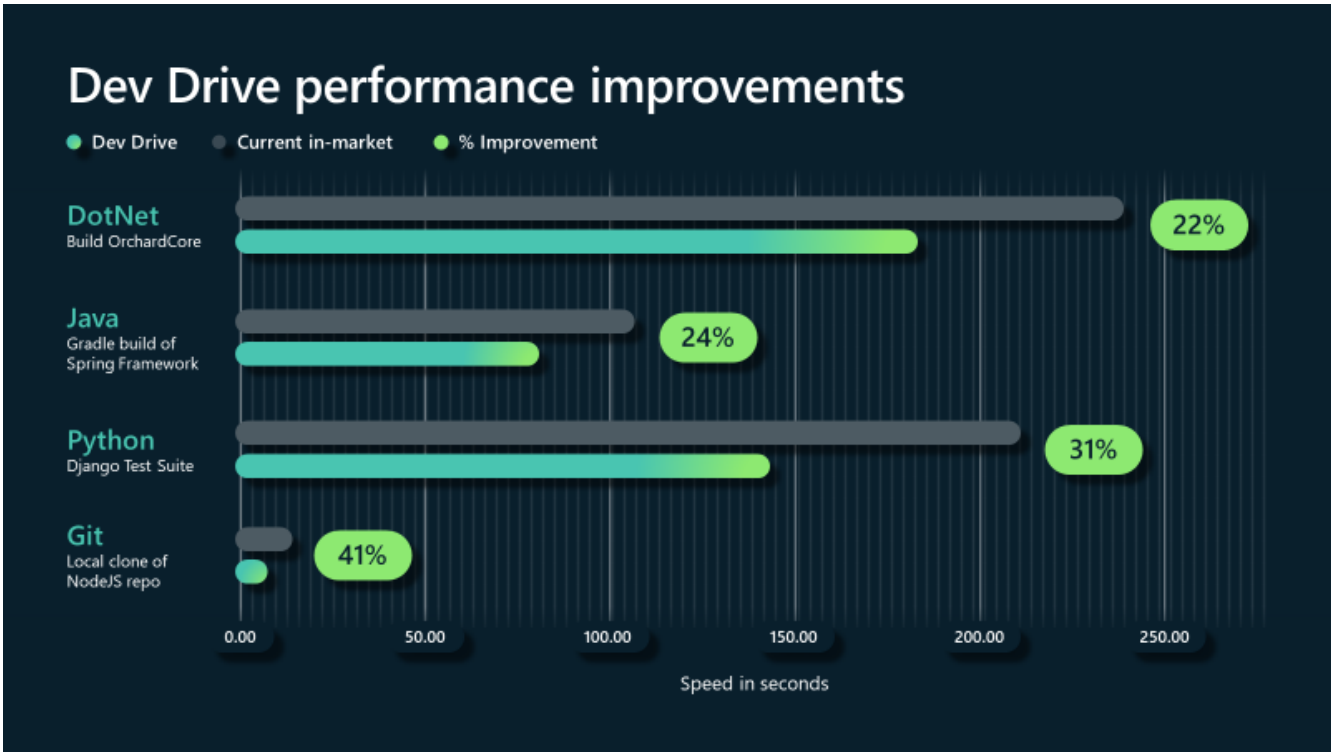
1. Run `sudo spctl developer-mode enable-terminal` in your terminal.
2. Go to System Preferences, and then to Security & Privacy.
3. Under the Privacy tab, go to `Developer Tools`.
4. Make sure your terminal is listed and enabled. If you're using any third-party terminals like iTerm or Ghostty, add them to the list as well.
5. Restart your terminal.



Thanks to the nextest and Zed developers for the tip.

WINDOWS ONLY: SET UP DEV DRIVE FOR RUST

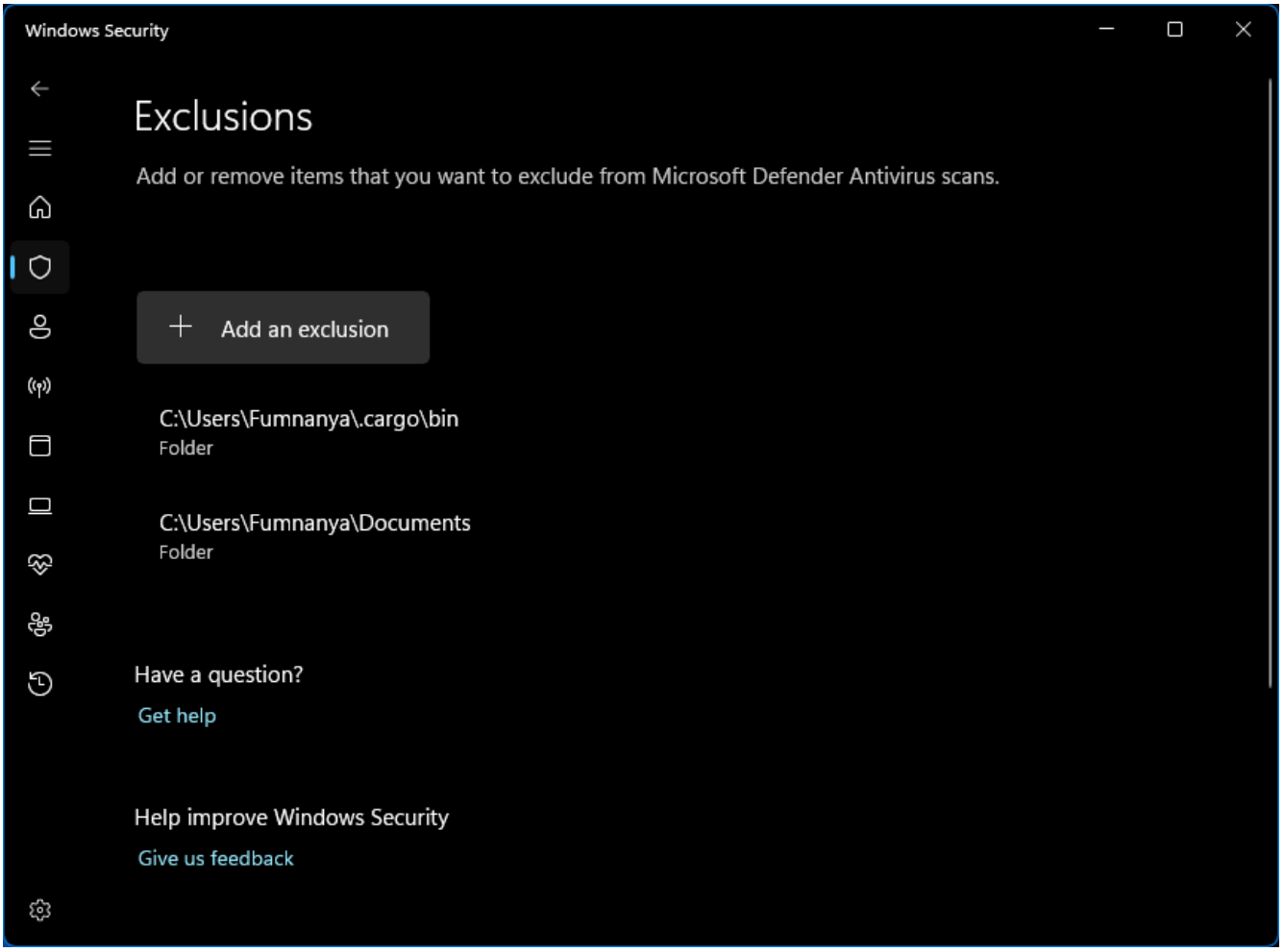
Windows 11 includes Dev Drive, a file system optimized for development. According to Microsoft, you can expect a speed boost of around 20-30% by using Dev Drive:



To improve Rust compilation speed, move these to a Dev Drive:

- Rust toolchain folder (`CARGO_HOME`)
- Your project code
- Cargo's `target` directory

You can go one step further and **add the above folders to your antivirus exclusions as well** for another potential speedup. You can find exclusion settings in Windows Security under Virus & threat protection settings.



Thanks to the nextest team for the tip.

TWEAK CODEGEN OPTIONS AND COMPILER FLAGS

Rust comes with a huge set of settings for code generation. It can help to look through the list and tweak the parameters for your project.

There are **many** gems in the full list of codegen options. For inspiration, here's bevy's config for faster compilation.

AVOID PROCEDURAL MACRO CRATES

If you heavily use procedural macros in your project (e.g., if you use serde), it might be worth it to play around with opt-levels in your `Cargo.toml`.

```
profile.dev.build-override]
pt-level = 3
```

As reader jfmontanaro mentioned on Github:

> *I think the reason it helps with build times is because it only applies to build scripts and proc-macros. Build scripts and proc-macros are unique because during a normal build, they are not only compiled but also executed (and in the case of proc-macros, they can be executed repeatedly). When your project uses a lot of proc-macros, optimizing the macros themselves can in theory save a lot of time.*

Another approach is to try and sidestep the macro impact on compile times with watt, a tool that offloads macro compilation to Webassembly.

From the docs:

> *By compiling macros ahead-of-time to Wasm, we save all downstream users of the macro from having to compile the macro logic or its dependencies themselves.*
>
> *Instead, what they compile is a small self-contained Wasm runtime (~3 seconds, shared by all macros) and a tiny proc macro shim for each macro crate to hand off Wasm bytecode into the Watt runtime (~0.3 seconds per proc-macro crate you depend on). This is much less than the 20+ seconds it can take to compile complex procedural macros and their dependencies.*

Note that this crate is still experimental.

Conditional compilation for procedural macros

Procedural macros need to parse Rust code, and that is a relatively complex task. Crates that depend on procedural macros will have to wait for the procedural macro to compile before they can compile. For example, `serde` can be a bottleneck in compilation times and can limit CPU utilization.

To improve Rust compile times, consider a strategic approach to handling serialization with Serde, especially in projects with a shared crate structure. Instead of placing Serde directly in a shared crate used across different parts of the project, you can make Serde an optional dependency through Cargo features.

Use the `cfg` or `cfg_attr` attributes to make Serde usage and `derive` in the shared crate feature-gated. This way, it becomes an optional dependency that is only enabled in leaf crates which actually perform serialization/deserialization.

This approach prevents the entire project from waiting on the compilation of Serde dependencies, which would be the case if Serde were a non-optional, direct dependency of the shared crate.

Let's illustrate this with a simplified example. Imagine you have a Rust project with a shared library crate and a few other crates that depend on it. You don't want to compile Serde unnecessarily when building parts of the project that don't need it.

Here's how you can structure your project to use optional features in Cargo:

In your `Cargo.toml` for the shared crate, declare serde as an optional dependency:

```
package]
ame = "shared"
ersion = "0.1.0"
dition = "2021"

dependencies]
erde = { version = "1.0", optional = true }
```

In this crate, use conditional compilation to include serde only when the feature is enabled:

```
[cfg(feature = "serde")]
se serde::{Serialize, Deserialize};

[cfg_attr(feature = "serde", derive(Serialize, Deserialize))]
ub struct MySharedStruct {
    // Your struct fields
.
```

In the other crates, enable the `serde` feature for the shared crate if needed:

```
package]
ame = "other"
ersion = "0.1.0"
```

```
dition = "2021"

dependencies]
hared = { path = "../shared", features = ["serde"] }
```

You can now use `MySharedStruct` with Serde's functionality enabled without bloating the compilation of crates that don't need it.

GENERICS: USE AN INNER NON-GENERIC FUNCTION

If you have a generic function, it will be compiled for every type you use it with. This can be a problem if you have a lot of different types.

A common solution is to use an inner non-generic function. This way, the compiler will only compile the inner function once.

This is a trick often used in the standard library. For example, here is the implementation of `read_to_string`:

```
ub fn read_to_string<P: AsRef<Path>>(path: P) -> io::Result<String> {
    fn inner(path: &Path) -> io::Result<String> {
        let mut file = File::open(path)?;
        let size = file.metadata().map(|m| m.len() as usize).ok();
        let mut string = String::with_capacity(size.unwrap_or(0));
        io::default_read_to_string(&mut file, &mut string, size)?;
        Ok(string)
    }
    inner(path.as_ref())
}
```

You can do the same in your code: the outer function is generic, while it calls the inner non-generic function, which does the actual work.

IMPROVE WORKSPACE BUILD TIMES WITH CARGO-HAKARI

Do you have a large Rust workspace with dependencies that:

1. Are used in multiple crates

2. Have different feature sets across those crates?

This situation can lead to long build times, as cargo will build each dependency multiple times with different features depending on which crate is being built. This is where `cargo-hakari` comes in. It's a tool designed to automatically manage "workspace-hack" crates.

In some scenarios, this can reduce consecutive build times by up to 50% or more. To learn more, check out the usage instructions and benchmarks in the official cargo-hakari documentation.

SPEEDING UP INCREMENTAL RUST COMPILATION WITH DYLIBS

```
# Install the tool
cargo install cargo-add-dynamic

# Add a dynamic library to your project
cargo add-dynamic polars --features csv-file,lazy,list,describe,rows,fmt,strings,tem
```

This will create a wrapper-crate around `polars` that is compiled as a dynamic library (`.so` on Linux, `.dylib` on macOS, `.dll` on Windows).

Essentially, it patches the dependency with

```
lib]
crate-type = ["dylib"]
```

With this trick, you can save yourself the linking time of a dependency when you only change your own code. The dependency itself will only be recompiled when you change the features or the version. Of course, this works for any crate, not just `polars`.

Read more about this on this blog post by Robert Krahn and the tool's homepage.

SWITCH TO THE NEW PARALLEL COMPILER FRONTEND

In nightly, you can now enable the new parallel compiler frontend. To try it out, run the nightly compiler with the `-Z threads=8` option:

```
RUSTFLAGS="-Z threads=8" cargo +nightly build
```

If you find that it works well for you, you can make it the default by adding `-Z threads=8` to your `~/.cargo/config.toml` file:

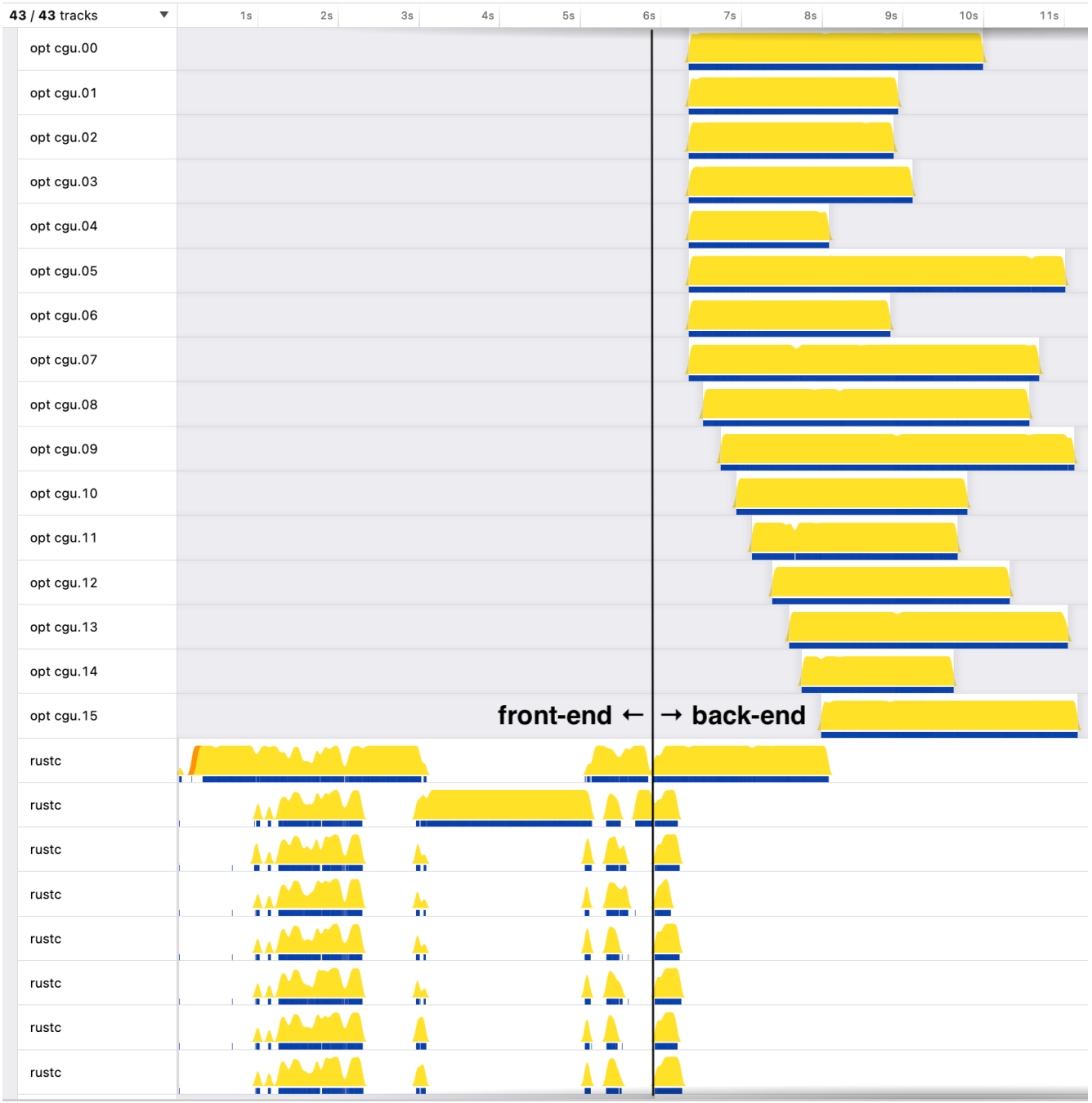
```
[build]
rustflags = ["-Z", "threads=8"]
```

Alternatively, you can set an alias for `cargo` in your shell's config file (e.g., `~/.bashrc` or `~/.zshrc`):

```
alias cargo="RUSTFLAGS='-Z threads=8' cargo +nightly"
```

When the front-end is executed in a multi-threaded setting using `-Z threads=8`, benchmarks on actual code indicate that compilation times may decrease by as much as 50%. However, the gains fluctuate depending on the code being compiled. It is certainly worth a try, though.

Here is a visualization of the parallel compiler frontend in action:



Find out more on the official announcement on the Rust blog.

USE A SCRATCH DISK FOR FASTER BUILDS

Your filesystem might be the bottleneck. Consider using an in-memory filesystem like for your build directory.

Traditional temporary filesystem like `tmpfs` is limited to your RAM plus swap space and can be problematic for builds creating large intermediate artifacts.

Instead, on Linux, mount an `ext4` volume with the following options:

```
o noauto_da_alloc,data=writeback,lazytime,journal_async_commit,commit=999,nobarrier
```

This will store files in the page cache if you have enough RAM, with writebacks occurring later. Treat this as if it were a temporary filesystem, as data may be lost or corrupted after a crash or power loss.

Credits go to /u/The_8472 on Reddit.

INVEST IN BETTER HARDWARE

If you reached this point, the easiest way to improve compile times even more is probably to spend money on top-of-the-line hardware.

As for laptops, the `M-series` of Apple's new Macbooks perform really well for Rust compilation.

The benchmarks for a Macbook Pro with M1 Max are absolutely *ridiculous* — even in comparison to the already fast M1:

Project	M1 Max	M1 Air
Deno	6m11s	11m15s
MeiliSearch	1m28s	3m36s
bat	43s	1m23s
hyperfine	23s	42s
ripgrep	16s	37s

That's a solid 2x performance improvement.

But if you rather like to stick to Linux, people also had great success with a multicore CPU like an AMD Ryzen Threadripper and 32 GB of RAM.

On portable devices, compiling can drain your battery and be slow. To avoid that, I'm using my machine at home, a 6-core AMD FX 6300 with 12GB RAM, as a build machine. I can use it in combination with Visual Studio Code Remote Development.

COMPILE IN THE CLOUD

If you don't have a dedicated machine yourself, you can offload the compilation process to the cloud instead.
Gitpod.io is superb for testing a cloud build as they provide you with a beefy machine (currently 16 core Intel Xeon 2.80GHz, 60GB RAM) for free during a limited period. Simply add `https://gitpod.io/#` in front of any Github URL. Here is an example for one of my Hello Rust episodes.

Gitpod has a neat feature called prebuilds. From their docs:

> *Whenever your code changes (e.g. when new commits are pushed to your repository), Gitpod can prebuild workspaces. Then, when you do create a new workspace on a branch, or Pull/Merge Request, for which a prebuild exists, this workspace will load much faster, because* ***all dependencies will have been already downloaded ahead of time, and your code will be already compiled****.*

Especially when reviewing pull requests, this could give you a nice speedup. Prebuilds are quite customizable; take a look at the `.gitpod.yml` config of nushell to get an idea.

CACHE ALL CRATES LOCALLY

If you have a slow internet connection, a big part of the initial build process is fetching all those shiny crates from crates.io. To mitigate that, you can download

all crates in advance to have them cached locally. criner does just that:

```
git clone https://github.com/the-lean-crate/criner
cd criner
cargo run --release -- mine
```

The archive size is surprisingly reasonable, with roughly **50GB of required disk space** (as of today).

TEST EXECUTION

USE CARGO NEXTEST INSTEAD OF `cargo test`

```
cargo install cargo-nextest
cargo nextest run
```

It's nice that `cargo` comes with its own little test runner, but especially if you have to build multiple test binaries, `cargo nextest` can be up to 60% faster than `cargo test` thanks to its parallel execution model. Here are some quick benchmarks:

Project	Revision	Test count	cargo test (s)	nextest (s)	Improvement
crucible	`cb228c2b`	483	5.14	1.52	3.38×
guppy	`2cc51b41`	271	6.42	2.80	2.29×
mdBook	`0079184c`	199	3.85	1.66	2.31×
meilisearch	`bfb1f927`	721	57.04	28.99	1.96×
omicron	`e7949cd1`	619	444.08	202.50	2.19×
penumbra	`4ecd94cc`	144	125.38	90.96	1.37×
reqwest	`3459b894`	113	5.57	2.26	2.48×
ring	`450ada28`	179	13.12	9.40	1.39×
tokio	`1f50c571`	1138	24.27	11.60	2.09×

COMBINE ALL INTEGRATION TESTS INTO A SINGLE BINARY

Have any integration tests? (These are the ones in your `tests` folder.) Did you know that the Rust compiler will create a binary for every single one of them? And every binary will have to be linked individually. This can take most of your build time because linking is slooow. 🐢 The reason is that many system linkers (like `ld`) are single threaded.

To make the linker's job a little easier, you can put all your tests in one crate. (Basically create a `main.rs` in your test folder and add your test files as `mod` in there.)

Then the linker will go ahead and build a single binary only. Sounds nice, but careful: it's still a trade-off as you'll need to expose your internal types and functions (i.e. make them `pub`).

If you have a lot of integration tests, this can result in a 50% speedup.

This tip was brought to you by Luca Palmieri, Lucio Franco, and Azriel Hoh. Thanks!

PUT SLOW TESTS BEHIND AN ENVIRONMENT VARIABLE

```
#[test]
fn completion_works_with_real_standard_library() {
  if std::env::var("RUN_SLOW_TESTS").is_err() {
    return;
  }
  ...
```

If you have slow tests, you can put them behind an environment variable to disable them by default. This way, you can skip them locally and only run them on CI.

(A nice trick I learned from matklad's (Alex Kladov) post.)

CI BUILDS

Tips for CI Builds

I wrote a dedicated article on how to speed up your CI builds. It covers a lot of the tips mentioned here in more detail and also includes more specific advice for Github Actions.

USE A CACHE FOR YOUR DEPENDENCIES

For GitHub actions in particular you can also use Swatinem/rust-cache.

It is as simple as adding a single step to your workflow:

```
obs:
 test:
   runs-on: ubuntu-latest
   steps:
     - uses: actions/checkout@v4
     - uses: dtolnay/rust-toolchain@stable
     - uses: Swatinem/rust-cache@v2
     - run: cargo test --all
```

With that, your dependencies will be cached between builds, and you can expect a significant speedup.

SPLIT UP COMPILE AND TEST STEPS

```
name: Compile
run: cargo test --no-run --locked

name: Test
run: cargo test -- --nocapture --quiet
```

This makes it easier to find out how much time is spent on compilation and how much on running the tests.

DISABLE INCREMENTAL COMPILATION IN CI

```
nv:
  CARGO_INCREMENTAL: 0
```

Since CI builds are more akin to from-scratch builds, incremental compilation adds unnecessary dependency-tracking and IO overhead, reducing caching effectiveness. Here's how to disable it.

TURN OFF DEBUGINFO

```
profile.dev]
lebug = 0
trip = "debuginfo"
```

Avoid linking debug info to speed up your build process, especially if you rarely use an actual debugger. There are two ways to avoid linking debug information: set `debug=0` to skip compiling it, or set `strip="debuginfo"` to skip linking it. Unfortunately, changing these options can trigger a full rebuild with Cargo.

- On Linux, set both for improved build times.
- On Mac, use `debug=0` since rustc uses an external strip command.
- On Windows, test both settings to see which is faster.

Note that without debug info, backtraces will only show function names, not line numbers. If needed, use `split-debuginfo="unpacked"` for a compromise.

As a nice side-effect, this will also help shrink the size of `./target`, improving caching efficiency.

Here is a sample config for how to apply the settings.

DENY WARNINGS THROUGH AN ENVIRONMENT VARIABLE

Avoid using `#![deny(warnings)]` in your code to prevent repetitive declarations. Furthermore, it is fine to get warnings during local development.

Instead, add `-D warnings` to `RUSTFLAGS` to globally deny warnings in all crates on CI.

```
env:
  RUSTFLAGS: -D warnings
```

SWITCH TO A FASTER GITHUB ACTIONS RUNNER

```
- runs-on: ubuntu-latest
+ runs-on: ubicloud
```

Services like Ubicloud, BuildJet, or RunsOn provide you with faster workers for your Github Actions builds. Especially for Rust pipelines, the number of cores can have a significant big impact on compile times, so it might be worth a try.

Here is an example from the Facebook Folly project using Ubicloud. Granted, this is a C++ project, but it shows the potential of faster runners:

After signing up with the service, you only need to change the runner in your Github Actions workflow file.

FASTER DOCKER BUILDS

USE `cargo-chef` TO SPEED UP DOCKER BUILDS

Building Docker images from your Rust code? These can be notoriously slow, because cargo doesn't support building only a project's dependencies yet, invalidating the Docker cache with every build if you don't pay attention. `cargo-chef` to the rescue! ⚡

> *`cargo-chef` can be used to fully leverage Docker layer caching, therefore massively speeding up Docker builds for Rust projects. On our commercial codebase (~14k lines of code, ~500 dependencies) we measured a **5x speed-up**: we cut Docker build times from **~10 minutes to ~2 minutes.***

Here is an example `Dockerfile` if you're interested:

```
# Step 1: Compute a recipe file
FROM rust as planner
WORKDIR app
RUN cargo install cargo-chef
COPY . .
RUN cargo chef prepare --recipe-path recipe.json

# Step 2: Cache project dependencies
FROM rust as cacher
WORKDIR app
RUN cargo install cargo-chef
COPY --from=planner /app/recipe.json recipe.json
RUN cargo chef cook --release --recipe-path recipe.json

# Step 3: Build the binary
FROM rust as builder
WORKDIR app
COPY . .
# Copy over the cached dependencies from above
COPY --from=cacher /app/target target
COPY --from=cacher /usr/local/cargo /usr/local/cargo
```

```
RUN cargo build --release --bin app

# Step 4:
# Create a tiny output image.
# It only contains our final binary.
FROM rust as runtime
WORKDIR app
COPY --from=builder /app/target/release/app /usr/local/bin
ENTRYPOINT ["/usr/local/bin/app"]
```

`cargo-chef` can help speed up your continuous integration with Github Actions or your deployment process to Google Cloud.

CONSIDER EARTHLY FOR BETTER BUILD CACHING

Earthly is a relatively new build tool that is designed to be a replacement for Makefiles, Dockerfiles, and other build tools. It provides fast, incremental Rust builds for CI.

> *Earthly speeds up Rust builds in CI by effectively implementing Cargo's caching and Rust's incremental compilation. This approach significantly reduces unnecessary rebuilds in CI, mirroring the efficiency of local Rust builds.*
>
> *Source: Earthly for Rust*

They use a system called Satellites, which are persistent remote build runners that retain cache data locally. This can drastically speed up CI build times by eliminating cache uploads and downloads. Instead of bringing the cache data to the compute, they colocate the cache data and compute, eliminating cache transfers altogether. Less I/O means faster builds.

Earthly also provides a `lib/rust` library, which abstracts away cache configuration entirely. It ensures that Rust is caching correctly and building incrementally in CI. It can be used in your `Earthfile` like this:

```
IMPORT github.com/earthly/lib/rust
```

If you're curious, Earthly's Guide for Rust details a simple Rust example with optimized caching and compilation steps.

IDE-SPECIFIC OPTIMIZATIONS

If you find that build times in your development environment are slow, here are a few additional tips you can try.

SLOW DEBUG SESSIONS IN VISUAL STUDIO CODE

If you're using Visual Studio Code and find that **debug sessions** are slow, make sure you don't have too many breakpoints set. Each breakpoint can slow down the debug session.

CLOSE UNRELATED PROJECTS

In case you have multiple projects open in Visual Studio Code, **each instance runs its own copy of rust-analyzer**. This can slow down your machine. Close unrelated projects if they aren't needed.

FIX RUST ANALYZER CACHE INVALIDATION

If you're using rust-analyzer in VS Code and find that you run into slow build times when saving your changes, it could be that the cache gets invalidated. This also results in dependencies like `serde` being rebuilt frequently.

You can fix this by configuring a separate target directory for rust-analyzer. Add this to your VS Code settings (preferably user settings):

```
"rust-analyzer.cargo.targetDir": true
```

This will make rust-analyzer build inside `target/rust-analyzer` instead of the

default `target/` directory, preventing interference with your regular `cargo run` builds.

Some users reported significant speedups thanks to that:

```
ıefore: 34.98s user 2.02s system 122% cpu 30.176 total
ıfter:   2.62s user 0.60s system 84% cpu 3.803 total
```

This could also help with rust analyzer blocking debug builds.

Credit: This tip was shared by asparck on Reddit.

SUMMARY

In this article, we've covered a lot of ground. We've looked at how to speed up your Rust builds by using better hardware, optimizing your code, and using better tools.

I hope that you were able to use some of these tips to speed up your Rust builds. In case you found other ways to speed up your Rust builds, or if you have any questions or feedback, I'd love to hear from you.

ADDITIONAL RESOURCES

- The Rust Perf Book has a section on compile times.
- List of articles on performance on Read Rust.
- 8 Solutions for Troubleshooting Your Rust Build Times is a great article by Dotan Nahum that I fully agree with.
- Improving the build times of a bigger Rust project (lemmy) by 30%.
- arewefastyet (offline) measures how long the Rust compiler takes to compile common Rust programs.
- Speeding up the Rust edit-build-run cycle : A benchmark-driven approach to improving Rust compile times.

Published: 2024-01-12

Last updated: 2025-05-04

Author: Matthias Endler

Editor: Simon Brüggen

notified when I publish them, you should sign up to my newsletter here. No spam. Unsubscribe at any time.

mail@example.com

Subscribe

↑Back to top

SERVICES

Consulting

Business Inquiries

Why Rust?

About

LEARN

Case Studies

Migration Guides

Conferences

Resources

MIGRATION GUIDES

Python

TypeScript

Java

Scala

CONNECT

Blog

Podcast

LinkedIn

Legal Notice