Scaling Rust builds with Bazel

✏ 2023-03-20 ✂ 2023-03-20

Introduction

As of March 2023, the Internet Computer repository contains about six hundred thousand lines of Rust code. Last year, we started using Bazel as our primary build system, and we couldn't have been happier with the switch. This article explains the motivation behind this move and the migration process details.

Cargo's limitations

Many Rust newcomers, especially those with a C++ background, swear by cargo. Rust tooling is fantastic for beginners, but we became dissatisfied with cargo as the project size increased. Most of our complaints fall into two categories.

Cargo is not a build system

> *Cargo is the Rust package manager. Cargo downloads your Rust package's dependencies, compiles your packages, makes distributable packages, and uploads them to crates.io.*
>
> The Cargo Book

Let's acknowledge the elephant in the room: cargo is not a build system; it's a tool for building and distributing Rust packages. It can build Rust code for a specific platform with a given set of features in a single invocation. Cargo chose simplicity and ease of use over generality and scalability; it does not track dependencies well or support arbitrary build graphs.

These trade-offs make cargo easy to pick up but impose severe limitations in a complex project. There are workarounds, such as xtask, but they will only get you so far. Let's consider an example of what many of our tests must do:

> Build a sandbox binary for executing WebAssembly.
>
> Build a WebAssembly program.
>
> Post-process the WebAssembly program (strip some custom sections and compress the result, for example).
>
> Build and execute a test binary that launches the sandbox binary, sends the WebAssembly program to the sandbox and interacts with the program.

This simple scenario requires invoking cargo three times with different arguments and appropriately post-processing and threading the build artifacts. There is no way to express such a test using cargo alone; another build system must orchestrate the test execution.

Poor caching and dependency tracking

Like notorious Make, cargo relies on file modification timestamps for incremental builds. Updating code comments or switching git branches can invalidate cargo's cache, causing long rebuilds. The sccache tool can improve cache hits, but we saw no improvement from using it on our continuous integration (CI) servers.

Cargo's dependency tracking facilities are relatively simplistic. For example, we can tell cargo to rerun build.rs if some input files or environment variables change. Still, cargo has no idea which files or other resources tests might be accessing, so it must be conservative with caching. Consequently, we often build way more than we need to, and sometimes our builds fail with confusing errors that go away after `cargo clean`.

The CI saga

Over the project life, we used various tools to mitigate the cargo's limitations, with mixed success.

The nix days

When we started the Rust implementation in mid-2019, we relied on nix to build all our software and set up the development environment in a cross-platform way (we develop both on macOS and Linux).

As our code base grew, we started to feel nix's limitations. The unit of caching in nix is a derivation. If we wanted to take full advantage of nix's caching capabilities, we would have to "nixify" all our external dependencies and internal Rust packages (one derivation per Rust package). After a long fight with build reproducibility issues, our glorious dev-infra team implemented fine-grained caching using the cargo2nix project.

Unfortunately, most developers in the team were uncomfortable with nix. It became a constant source of confusion and lost developer productivity. Since nix has a steep learning curve, only a few nix wizards could understand and modify the build rules. This nix-alienation bifurcated our build environment: the CI servers built the code with nix-build, and developers built the code by entering the nix-shell and invoking cargo.

The iceberg

The final blow to the nix story came around late-2020, close to the network launch. Our security team chose Ubuntu as the deployment target and insisted that production binaries link against the regularly updated system libraries (libc, libc++, openssl, etc.) the deployment platform provides. This setup is hard to achieve in nix without compromising correctness[1].

Furthermore, the infrastructure team got a few new members unfamiliar with nix and decided to switch to a more familiar technology, Docker containers. The team implemented a new build system that runs cargo builds inside a docker container with the versions of dynamic libraries identical to those in the production environment.

The new system grew organically and eventually evolved into a hot mess of a hundred GitLab Yaml configuration files calling shell and python scripts in the correct order. These scripts used the known

filesystem locations and environment variables to pass the build artifacts around. Most integration tests ended up as shell scripts expected some inputs that the CI pipeline produces.

The new Docker-based build system lost the granular caching capabilities of nix-build. The infra team attempted to build a custom caching system but eventually abandoned the project. Cache invalidation is a challenging problem indeed.

With the new system, the chasm between the CI and development environments deepened further because the nix-shell didn't go anywhere. The developers continued to use nix-shell for everyday development. It's hard to pinpoint the exact reason. I attribute that to the fact that entering the nix-shell is less invasive than entering a docker container, and nix-shell does not require running in a virtual machine on macOS (Rust compile times are slow). Also, the infra team was so busy rewriting the build system that improving the everyday developer experience was out of reach.

I call this setup an "iceberg": on the surface, a developer needed only nix and cargo to work on the code, but in practice, that was only 10% of the story. Since most tests required a CI environment, developers had to create merge requests to check whether their code worked beyond the basic unit tests. The CI didn't know developers were interested in running a specific test and executed the entire test suite, wasting scarce computing resources and slowing the development cycle.

The tests accumulated over time, the load on the CI system grew, and eventually, the builds became unbearably slow and flaky. It was time for another change.

Enter Bazel

Among about a dozen build systems I worked with, Bazel is the

only one that made sense to me[2]. One of my favorite features of Bazel is how explicit and intuitive it is for everyday use.

Bazel is like a good videogame: it's easy to learn and challenging to master. It's easy to define and wire build targets (that's what most engineers do), but adding new build rules requires some expertise. Every engineer at Google can write correct build files without knowing much about Blaze (Google's internal variant of Bazel). The build files are verbose bordering plain boring, but it's a good thing. They tell the reader precisely what the module's artifacts and dependencies are.

Bazel offers many features, but we mostly cared about the following:

> Bazel is extensible enough to cover all our use cases. Bazel gracefully handled everything we threw at it: Linux and macOS binaries, WebAssembly programs, OS images, Docker containers, Motoko programs, TLA+ specifications, etc. The best part is: We can also combine and mix these artifacts in any way we like.
>
> Aggressive caching. The sandboxing feature ensures that build actions do not use undeclared dependencies, making it much safer to cache build artifacts and, most importantly for us, test results.
>
> Remote caching. We use the cache from our CI system to speed up developer builds.
>
> Distributed builds. Bazel can distribute tasks across multiple machines to finish builds even faster.
>
> Visibility control. Bazel allows package authors to mark some packages as internal to prevent other teams from importing the code. Controlling dependency graphs is crucial for fast builds.

Even more importantly, Bazel unifies our development and CI environments. All our tests are Bazel tests now, meaning that every developer can run any test locally. At its heart, our CI job is `bazel test --config=ci //...`.

One nice feature of our Bazel setup is that we can configure versions of our external dependencies in a single file. Ironically, cargo developers implemented support for workspace dependency inheritance a few weeks after we finished the migration.

The migration process

> *You are such a naïve academic. I asked you how to do it, and you told me what I should do. I know what I need to do. I just don't know how to do it.*
>
> Attributed to Andrew Grove; see "The 4 Disciplines of Execution" by Jim Huling, Chris McChesney, and Sean Covey, page xx.

The idea of migrating the build system came from a few engineers (read Xooglers) who were tired of fighting with long build times and poor tooling. To our surprise, a few volunteers expressed interest in joining the rebellion at its earliest stage. We needed a plan for executing the switch and getting the management's buy-in.

The first rule of large codebases is to introduce significant changes gradually. This section describes our process of migration, which took several months.

Build a prototype

We started migration by building a prototype. We created a sample repository that mimicked the features of our code base that we

expected to bring the most trouble, such as generating Protocol Buffer types using the prost library, compiling Rust to WebAssembly and native code in a single invocation, and setting up rust-analyzer support. Once we knew that the most complex problems we face have a solution at a small scale, we presented the case to the management, explained the final vision, how many people and time we needed, and got a green light. Now the real work began.

Dig a tunnel from the middle

Our CI was a multi-stage process that treated cargo as a black box producing binaries from the source code. There were two major work streams in our mission to minimize build times:

> Replace the spaghetti of YAML files and scripts using cargo as a black box with neat Bazel targets with explicit dependencies. This change would bring clarity and confidence to our CI routines and enable developers to access the build artifacts without an entire CI run.
>
> Use Bazel to build binaries from Rust code directly, bypassing cargo. This change would significantly improve our cache hit rate and allow us to avoid running expensive tests on every change.

These work streams require different skill sets, and we wanted to start working on them in parallel. To unblock the first workstream, we created a simple Bazel rule, `cargo_build`, that treated cargo as a black box and produced binaries for deployment and tests. This way, our infrastructure experts could figure out how to build OS images with Bazel, while our Rust experts could proceed with the Rust code “bazelification”.

Run CI early

We added the `bazel test //...` job to our CI pipeline as soon as we had the first `BUILD` file in our repository. The extra job slightly increased the CI wait time but ensured that packages converted to Bazel wouldn't degrade over time. As a side benefit, developers started to experience Bazel-related CI failures during their code refactorings. They actively learned to modify `BUILD` files and gradually became accustomed to the new world.

One package at a time

The goal of the second workstream was converting a few hundred Rust packages to the new build rules. We started from the core packages at the bottom of the stack that needed special treatment, and then project volunteers bazelified a few packages at a time when they had a free time slot. Two little tricks helped us with this tedious task:

> Automation. The infra team invested a few days in a script that converted a `Cargo.toml` file to a 90% complete `BUILD` file matching our guidelines. Many packages required manual treatment, and the generated `BUILD` file was far from optimal, but the script boosted the conversion process significantly.
>
> Progress visualization. One team member wrote a utility visualizing the migration progress by inspecting the cargo dependency graph and searching for packages with and without `BUILD` files. This little tool had a tremendous effect on our morale.

Eventually, we could build and test every piece of our Rust code with Bazel. We then switched the OS build from the `cargo_build` bootstrapping to the binaries built from the source using Bazel rules.

Ensure test parity

The last piece of the puzzle was ensuring the test parity. Cargo discovers tests automagically, while Bazel `BUILD` files require explicit targets for each type of test (crate tests, doc tests, integration tests). The infra team wrote another little utility that analyzed the outputs of cargo and Bazel build pipelines and compared the list of executed tests, ensuring that the volunteers accounted for every test during the migration and that developers didn't forget to update `BUILD` files when they added new tests.

Rough edges

Bazel solves most of our needs regarding building artifacts, but we have yet to replicate a few cargo features related to developer flow.

> Cargo check. Cargo does not produce binaries when run in check mode, making it much faster than cargo build. Developers often use this mode to check whether the entire code base compiles after a refactoring.
>
> IDE support. The rules_rust Bazel plugin offers experimental support for rust-analyzer, which worked perfectly in the prototype but choked on our code base. We invested a lot of effort in making the new setup work, but we still keep cargo files around to keep developers relying on IntelliJ Rust happy [3].
>
> Publishing packages. We want to publish some of our Rust packages to crates.io, and the rules_rust Bazel plugin does not provide a replacement for "cargo publish" yet.
>
> Access to the cargo ecosystem. Many helpful tools rely on cargo and don't have an analog in the Bazel world yet, such as `cargo-expand`.

Because of these issues, we still keep cargo files around. Luckily, this does not affect our CI times much because the only check we need is that `cargo check --tests --benches` succeeds.

Acknowledgments

The Bazel migration project was a definitive success. I thank our talented infra team and all the volunteers who contributed to the project.

Special thanks go to the developers and maintainers of the rules_rust Bazel plugin, who unblocked us many times during the migration, especially Andre Uebel and Daniel Wagner-Hall, and to Alex Kladov for taking the time to share his rust-analyzer expertise.

You can discuss this article on Reddit.

Similar articles

When Rust hurts

Designing error types in Rust

Rust at scale: packages, crates, and modules

←If composers were hackers

Book summary: Building a Second Brain→
