The Rust Build System

Aron Hansen Berggen
Yannik Sander

April 2021

1 Introduction

There is a lot of talk about how DevOps supposedly solves all your problems when developing software, from the features planned to how they are used in production and everything between. Even open-source projects are expected to follow DevOps principles these days. The Rust programming language is no exception.
But how does DevOps Scale for projects as massive as rust, an advanced programming language which compiles to every mainstream platform with tooling, documentation and hundreds of contributors? This is what will be explored and explained in the upcoming sections.

We start by explaining what we are looking at, before diving into the tools that ensure this project's integrity.

1.1 What is Rust

According to the official Rust language website[1], Rust is a low-level programming language that focuses on performance, reliability and productivity. It extends these in the following ways:

- Performance:
  Memory efficient without any garbage collector or run-time.
- Reliability:
  Extensive type system and ownership models to guarantee memory and thread safety at compile time.
- Productivity:
  The tooling for working with rust such as documentation, build tools and compiler errors are feature-complete and reliable.

These points together with a strong type-system and its expansive ecosystem are pointed out as the reasons for Rust being voted as the most loved programming language 4 years in a row in the yearly surveys by Stack Overflow[2].

1.2 What is a "CI"

A CI, or Continuous Integration, takes some input and performs a set of steps to output some product at the end. This often includes code reviews, automated tests, code compilation, packaging of compiled code and further validation of the packages before publishing.

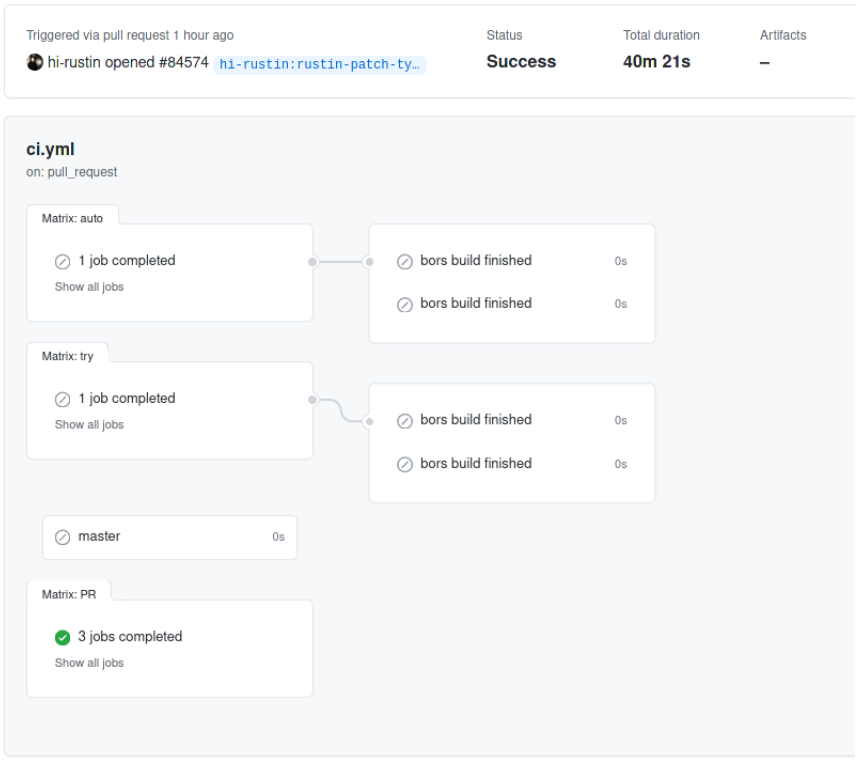

Figure 1: CI pipelines are run by different events, depicted are the standard tests for Rust PR's.

Throughout this process, one of the most important aspects is to monitor these results and act on that feedback.[3], [4]

2 The ~~Rabbit Hole~~ Rust Build CI

This section presents the specifics of the Rust project's build process. We strive to present a comprehensive picture of the most important pieces, as the overall process would not get a fair representation in this limited scope. We will focus on how Rust facilitates the **bors** bot and its development pattern while also introducing the social interactions bot **triagebot**.

2.1 Triagebot

The Triagebot can be summarised as the people and issues managing bot of the Rust development process. Its documentation[5] explains how it achieves this:
It is the instantiator of tags on issues by being invoked by its GitHub name, "@rustbot". It also makes it easy to ping teams within the Rust organisation, for example, the documentation maintainers by tracking more than just people and issues. This makes it easier for the teams to assist without those in need of help knowing whom directly to contact, as well as for the teams to

find issues that need their attention.

2.2 Developing with bors

Inspecting the `rust-lang/rust`[1] repository two things become apparent:

First, none of the main branches shows any trace of being a *development branch*. On the contrary, their naming (`stable, beta, master`) rather suggests a connection with the release. Secondly, supporting the first insight, the commit history appears very automated as well. Many commits are structured as follows:

```
Auto merge of #84353 - estebank:as-ref-mir, r=davidtwco
```

Under closer inspection these commits become apparent as **merge-commits**, merging a set of changes into the master branch. All of these commits are merged by bors and refer to PR.

We see: bors automates the merging of PR. Indeed *none* of the commits made directly onto the master branch are by any other committer than bors.

Yet, where do these commits come from and more importantly are they even verified?

2.2.1 The Pull Request

When opening a Pull Request (PR) on the rust repository, if you did not ask for any specific reviewer, the **rust-highfive** bot assigns a rust team member to it. This affects the assignee on GitHub as well as leaving a comment picked up by triagebot.

A GitHub Action `pr` will test the code on three major platforms (Windows and two Linux systems with different toolchains).

If if the action succeeds and a maintainer reviewed your code they might comment in one of the following ways:

- `r+` In bors terms this is equivalent to pressing the green "Merge" button and comes with the same responsibility
- Mark the PR as rollup target. Note; We will cover rollups in depth after going through bors' workflow below.
- Instruct the author to do changes
- `r? @username` Ask for another review

In case of the first action (`r+`) we see bors commenting with something like:

> Testing commit f43ee8e with merge e888a57...

This, given the tests run successfully and the change is approved, will cause bors to merge the PR.
So far so good, but a question still remains:

> *"What is tested, where and how? After all, no clear traces of tests are present in the repository."*

[1] `https://github.com/rust-lang/rust`

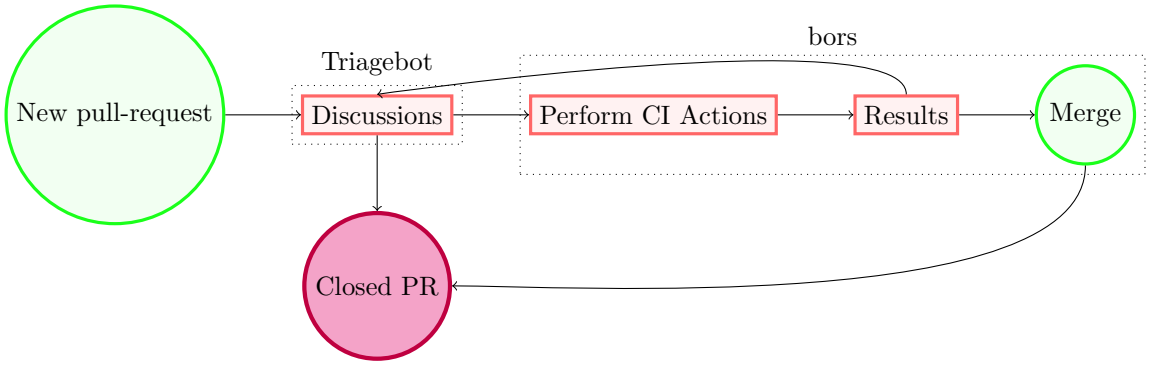

Figure 2: The PR workflow, where bors operates.

2.2.2 A bot's playground

The whole CI of the rust project is managed in a fork of the main repository. That fork lives in `rust-lang-ci/rust`[2]. It ensures that the main repository, especially its master branch, always builds without failure. Interactions with this repository are indirect and managed by bors. The two main branches that are of interest are `auto` and `try` both of which trigger GitHub Actions workflows with the same name as defined in the repository.[6]

When a maintainer approves a change (i.e. by commenting with `r+`) bors takes the changes and force-pushes them to the `auto` branch. Changes on this branch subsequently trigger the GitHub Actions system.

The `auto` Action The `auto` action is the heart of rust's CI build process. It builds and tests the rust compiler on all 57 supported platforms[7]. Most of the action sets up the CI environment including connecting to a compiler cache for intermediate built dependencies [3]**sccache** and initializing the repository by pulling all related submodules.

Once this is done the actual build will be executed. On available platforms this is run natively. All other platforms are cross-compiled using a custom docker image.

Finally, assuming the build succeeded the produced artefacts are pushed to an S3 object storage[4].

Unsurprisingly, as the build and tests need to succeed for such a large number of targets, a full CI run on `auto` takes *two-and-a-half to three hours*[8]. The whole CI runs on custom machines owned by the rust projects[9]. This means they are always available for bors, but only in a limited capacity. The extension of this is that it is not possible to run more than eight `auto` builds per day and thus at most eight PRs are merged every day!

The very curious reader might ask themselves now how this system is supposed to facilitate a CI workflow incorporating *regular* automated testing. In part, this is where the second action comes

[2] `https://github.com/rust-lang-ci/rust`
[3] `https://github.com/mozilla/sccache`
[4] `https://aws.amazon.com/s3/`

into play.

The `try` Action The `try` action, like `auto` is triggered by pushes made by bors targeting the `try` branch.

This action is structurally equivalent to the one described before. The major difference shows when looking at the build targets. Instead of 57 targets it only runs for *x86_64 Linux*, for it is the most widely available target. It allows a maintainer to get an idea of whether the change introduced by a PR will build as in contrast to the `pr` action this one build the complete rust toolchain including all tests.

With `try` and `auto` in mind one might wonder:

> *"Will all changes trigger a full build and test on 50+ platforms? Is that not pretty wasteful for simple changes such as correcting missspellings?"*

It is and that brings us back to the PR process from before.

2.2.3 Rollups

In Section 2.2.2 a maximum number of 8 PR's per day limit was exposed. However, documentation changes and other small fixes that do not impose any notable risk to the pipeline can be bundled. These bundled PR's are known as a *rollup*[10].

A rollup is initiated by a maintainer through bors' web interface [5]. It shows mergeability and the associated risk for open PRs and lets the maintainer select a group of PRs that will be queued for bors to push them onto the CI's `auto` branch. This causes them to be tested collectively thus saving resources.

2.3 The Green Master

> "**The Not Rocket Science Rule Of Software Engineering**:
> automatically maintain a repository of code that always passes all the tests"
>
> — Graydon Hoare, founder of Rust

As stated previously, successfully running/testing on auto causes bors to commit these changes back to master. As the auto branch is always the latest master this ensures that the main repository is always *green*.

Yet, not everything on the rust repository is the rust compiler! There are some custom tools that might fail to build but do not influence the compiler. These breakages should not fail the tests. Therefore when the CI merges changes into master it runs its final action `master` which collects information about which tools succeeded to build and publishes them to the toolstate repository[6].

[5] `https://bors.rust-lang.org/queue/rust`

[6] Toolstate of the latest build available under: `https://rust-lang-nursery.github.io/rust-toolstate/`

2.4 Action! Or lack there off

The Rust CI is running on the GitHub Actions platform[7], but what is it? One engineer, Jonas Hecht describes GitHub Actions in a blog post[11] as the next generation CI/CD. He goes further to explain why: CI/CD has been available as file configurations for a good while now, but GitHub Actions takes it one step further with the easily accessible pre-built actions available to everyone on GitHub. He describes them as the missing Lego™ blocks for building a complete CI/CD pipeline, with close to single click interactions with code analysis and deployment tools as examples.

Yet, analysing the actions file[12] we see that there are not any GitHub Actions ran by the Rust CI. The Rust project does not need more features for their build pipeline as they have built custom tooling for the necessary parts instead, sacrificing DevOps velocity offered by this available platform to push for higher build-pipe throughput, reliability and security.

3 Wrap Up

Having escaped the wonderland of rust, what did we learn?

The most unique feature of the rust project might be that most of its CI, although running through GitHub PRs, is mostly independent. First of all, it's run in its own repository, whose master is a bot. Secondly, all the interactions with the CI happen through commands sent to a bot either in form of comments or its web interface. This extends into this bot bors performing all the heavy lifting merging PRs. And finally, although running on GitHub's infrastructure, the actual actions are entirely implemented as part of the project.

These features constitute a unique integration of custom tooling and GitHub and make rust one of the most extraordinary open-source projects.

References

[1] Rust Programming Language, en-US. [Online]. Available: https://www.rust-lang.org/ (visited on 04/14/2021).

[2] J. Goulding, What is Rust and why is it so popular? en-US, https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/, Jan. 2020.

[3] The Importance of the DevOps Pipeline + How to Build, en-US, https://phoenixnap.com/blog/devops-pipeline, Jul. 2020.

[4] E. Katz, The essential steps to building your own DevOps pipeline, en, https://blog.exigence.io/build-devops-pipeline.

[5] Triagebot - Rust Forge. [Online]. Available: https://forge.rust-lang.org/platforms/zulip/triagebot.html (visited on 04/19/2021).

[6] Rust-lang/rust CI - Rust Forge. [Online]. Available: https://forge.rust-lang.org/infra/docs/rustc-ci.html#which-branches-we-test (visited on 04/14/2021).

[7] Rust's CI is moving to GitHub Actions — Inside Rust Blog, en. [Online]. Available: https://blog.rust-lang.org/inside-rust/2020/07/23/rust-ci-is-moving-to-github-actions.html (visited on 04/20/2021).

[8] Rust-lang-ci/rust, en. [Online]. Available: https://github.com/rust-lang-ci/rust (visited on 04/26/2021).

[9] Custom GitHub Actions runners - Rust Forge. [Online]. Available: https://forge.rust-lang.org/infra/docs/gha-self-hosted.html (visited on 04/23/2021).

[10] Rollup Procedure - Rust Forge. [Online]. Available: https://forge.rust-lang.org/release/rollups.html (visited on 04/23/2021).

[11] Stop re-writing pipelines! Why GitHub Actions drive the future of CI/CD, en-US, https://blog.codecentric.de/en/2021/03/github-actions-nextgen-cicd/, Mar. 2021.

[12] Rust-lang/rust, en. [Online]. Available: https://github.com/rust-lang/rust (visited on 04/21/2021).