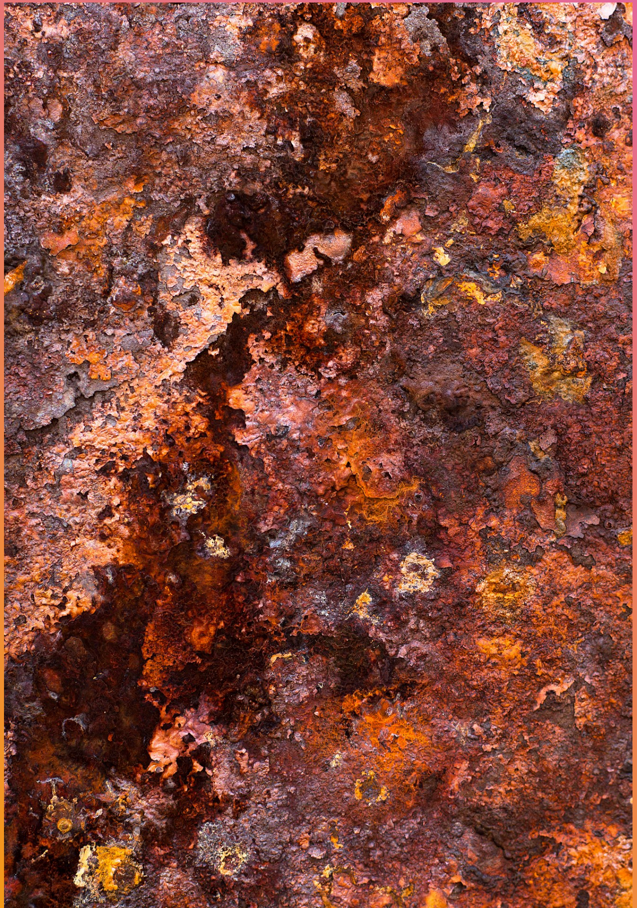

Rust Build Iteration

Strategies to speed it up

or dynamic hot reloading?

François Mockers

- QA Lead @ **PayLead**
- Maintainer @ **Bevy**
- **Vleue** for all your consulting needs on CI, Testing, Build and Bevy

https://bsky.app/profile/francois.mock.rs

https://github.com/mockersf

Disclaimer

- No numbers
  - Depends on too many factors specific to your project
  - Hardware, development environment, kind of project, shape of dependency tree
- No silver bullet
  - If there was one, the rust project would do it
  - In fact they did and continue to do so!
- Recommendations of things to look into
- Measure on your use case!

Rust

It’s Slow And It’s OK

The Rust compiler does a lot of things

Step	Output	Checks
Lexing and Parsing	AST	Syntax, Macro Expansion, Name Resolution
AST Lowering	HIR	Type Inference, Trait Solving, Type Checking, Desugaring
MIR Lowering	MIR	Borrow Checker, Optimizations, Monomorphization, Desugaring, Const Evaluation
Code Generation	LLVM-IR	Optimizations
Compilation	Machine Code	
Linking	Executable	

https://rustc-dev-guide.rust-lang.org/overview.html

Some Definitions

Acronym	Meaning
AST	Abstract Syntax Tree
HIR	High-level Intermediate Representation
THIR	Typed High-level Intermediate Representation
MIR	Mid-level Intermediate Representation
LLVM-IR	LLVM Intermediate Representation

Is It Slow?

- How to measure global compilation time?
  - `hyperfine --prepare 'cargo clean' 'cargo build'`
  - `hyperfine --prepare 'touch src/lib.rs' 'cargo build'`
  - https://github.com/sharkdp/hyperfine
  - Run the command multiple times with optional setup / warmup, and compare different runs
- How to measure each dependency compilation time?
  - `cargo build --timings`
  - https://doc.rust-lang.org/cargo/reference/timings.html
  - Reports each crate time, the dependencies and order, and the threads usage
- How to measure each `rustc` step?
  - `RUSTFLAGS="-Zself-profile" cargo +nightly build`
  - https://rustc-dev-guide.rust-lang.org/profiling.html
  - Self profiling of the rust compiler

Is It Big?

- There is a relation between binary size and compilation time
- Find out what contributes the most to binary size
  - https://github.com/RazrFalcon/cargo-bloat
  - https://github.com/AlexEne/twiggy in Wasm
  - https://github.com/dtolnay/cargo-llvm-lines
- Strings created at compile time
  - https://linux.die.net/man/1/strings

The Easy Answer

- Buy a faster computer
- This is the end of this talk, thank you

Rust Project Goal 1/2

https://rust-lang.github.io/rust-project-goals/2025h2/index.html#flexible-faster-compilation

- Better parallelization
  - https://blog.rust-lang.org/2023/11/09/parallel-rustc/
  - Front end (down to MIR) parallelization
- Cranelift for development use
  - https://cranelift.dev
  - Faster compilation for less optimized binaries

Rust Project Goal 2/2

https://rust-lang.github.io/rust-project-goals/2025h2/index.html#flexible-faster-compilation

- Custom `std` for specific use cases
  - With a subset of `std`
  - With different optimization settings
- Avoid rebuilds when only relinking is needed
  - If a crate interface doesn’t change, its dependents don’t need to be rebuilt

Project Configuration

Cargo Profiles - dev

Cargo profiles control compilation settings

- `dev` profile
    - Fast compilation, slower runtime
    - Incremental compilation enabled
    - Debug symbols included
- Custom profiles for dev
    - Can override optimization level for all dependencies or for specific ones

```toml
[profile.dev.package."*"]
opt-level = 3
```

Cargo Profiles - release

Cargo profiles control compilation settings

- `release` profile
  - Slow compilation, fast runtime
  - Full optimizations enabled
- Custom profiles for release
  - Can make compilation even slower
  - `codegen-units`: parallel codegen units, the more you use the less knowledge each has
  - `lto`: link time optimization, default to `false`

Replace part of the toolchain

- The OS-provided linker is often slow, there are alternatives
  - `lld` https://lld.llvm.org (Windows, Linux)
    - Default on Linux since rust 1.90 https://blog.rust-lang.org/2025/09/01/rust-lld-on-1.90.0-stable/
  - `mold` https://github.com/rui314/mold (Linux)
  - `wild` https://github.com/davidlattimore/wild (Linux)
- Rust compilation is backed by LLVM
  - Cranelift can be faster
  - Goal is to be usable as a Just-In-Time compiler
  - https://cranelift.dev

Incremental Compilation

`rustc` tries to cache things that don't change between rebuilds

- Enabled by default for the `dev` profile!
- During compilation:
  - Build the HIR
  - Query the HIR
  - Build of graph of queries and their dependencies
- Cache query results for HIR that didn't change
- But
  - Caching is hard
  - Relies on disk

https://rustc-dev-guide.rust-lang.org/queries/incremental-compilation-in-detail.html

Shared Target Folder

- Share a common target folder between projects
  - Creating a workspace
  - Or using `target-dir`
- Avoid recompiling everything if you share dependencies between projects
- But
  - Projects can’t be built in parallel
  - Dependencies are unified (in a workspace)
  - Different features trigger rebuilds (with `target-dir`)
  - Can’t clean an individual project

Shared Compilation Cache

- `sccache` wraps `rustc` and caches compilation
- Cache can be shared between computers
  - Hosted on S3, R2, Redis, ...
- Build can be distributed
- But
  - Doesn’t work in some cases (build script, crates using the linker), fallbacks to `rustc`

https://github.com/mozilla/sccache

Why a Rebuild is Happening

```
CARGO_LOG=cargo::core::compiler::fingerprint=info cargo build
```

Common causes:

- Build scripts
- Different feature sets between builds
- File timestamp issues
- Another build or background process modified or deleted build artifacts

Not possible to debug after the fact why a build happened

https://doc.rust-lang.org/cargo/faq.html#why-is-cargo-rebuilding-my-code

rust-analyzer

- `rust-analyzer` shares the target folder with `cargo`
  - Can hold a lock
  - Running `cargo` quickly after a change will wait on `rust-analyzer`
- Configure different target directories
  - `rust-analyzer.cargo.targetDir`
  - But will use more disk space

https://rust-analyzer.github.io/book/faq.html#rust-analyzer-and-cargo-compete-over-the-build-lock

Replace cargo

- `bazel`, `buck2`, ...
- Hermeticity
- Reproducibility
- Complex setup

Side Note: CI

- Don’t use the same settings
  - Disable incremental builds
  - Disable debug info
- Why?
  - Avoid writing to disk
  - Smaller files to cache

Side Note: CI

- Use a build cache between runs
- Avoid redownloading from crate repository
- Avoid rebuilding everything from scratch
- Can be CI platform specific (GitHub Actions, GitLab Pipelines, ...)
- Can be `sccache`

Side Note: OS Specific

- macOS
  - code signing https://nnethercote.github.io/2025/09/04/faster-rust-builds-on-mac.html
  - On Apple Silicon, ensure everything is arm
- Windows
  - Anti virus checks on the target folder can slow down compilation

Project Organization

Tests

Unit tests vs integration tests have different compilation impacts

- Unit tests (`#[cfg(test)]`)
  - Compiled with the crate
  - Quick to rebuild
- Integration tests (`tests/` directory)
  - Each file is a separate crate
  - Slower if you have many test files
- Impacts `rust-analyzer` duration
- Reorganise your integration tests
  - Single test crate, with many modules
  - https://matklad.github.io/2021/02/27/delete-cargo-integration-tests.html
- Consider using `cargo nextest` for faster test execution https://nexte.st

Examples

Examples are compiled as separate binaries

- Each example in `examples/` is checked during `cargo check`
- Can significantly increase iteration time with many examples
- Impacts `rust-analyzer` duration
- Use `required-features` to gate examples behind features
- Consider using a single multi-example binary with CLI parameters / subcommands

Workspace Setup

Split your project into smaller crates

- Allows parallel compilation
- Cargo can build independent crates simultaneously
- Only rebuild what changed
  - Example: separate `core`, `api`, `cli` crates
- Be careful not to over-split
  - Overhead per crate
  - Complexity

Dependencies as Dynamic Libraries

- Setup your dependencies as dynamic libraries

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

- Significantly faster linking during iteration
- But
  - Miss on some optimizations
  - Binaries are not portable
  - Can be complex to setup if not already supported by the crate

https://robert.kra.hn/posts/2022-09-09-speeding-up-incremental-rust-compilation-with-dylibs/

Automatic Command Execution

- Iteration loop:
  - Change your code
  - Wait for `rust-analyzer` to not report issues
  - Trigger the tests
- Remove the human in the loop, automatically run your tests on any change
  - https://dystroy.org/bacon/
  - `bacon test` / `bacon nextest`
  - Can be customized for anything

Monitoring Your Dependencies

- Avoid unused dependencies
  - https://github.com/est31/cargo-udeps
  - https://github.com/bnjbvr/cargo-machete
- Avoid duplicate dependencies
  - Cargo will unify dependencies based on SemVer compatibility
  - https://github.com/EmbarkStudios/cargo-deny
  - Can be a lot of work to update your dependencies' dependencies

Code Organization

Conditional Compilation & Features

Conditional compilation can reduce build scope

- Use feature flags to gate expensive code

```rust
#[cfg(feature = "expensive")]
mod expensive;
```

- Also gate dependencies only used behind gated code
- Only enable features you're actively working on
- Example: gate database backends, UI frameworks
- Disable default features of your dependencies and only enable what you need
- Keep in mind:
  - Features are additive, and must not cause issues when enabled together
  - It can explode complexity of tests as it's a combinatorial issue

Declarative Macros

Macros run at compile time

- `macro_rules!`: apply a template
- Can generate more code
- Recursive declarative macros

```
macro_rules! smaller_tuples_too {
  ($m: ident, $ty: ident) => { $m!{$ty} };
  ($m: ident, $ty: ident, $($tt: ident),*) => {
      $m!{$ty, $($tt),*}
      smaller_tuples_too!{$m, $($tt),*}
  };
}
smaller_tuples_too!(other_macro, F, E, D, C, B, A)
```

  - or worse: https://lukaswirth.dev/tlborm/decl-macros/patterns/tt-muncher.html
- Consider expanding macros

Procedural Macros

Procedural macros can do anything

- Three types
  - `#[derive( … )]` : add code, mostly used to implement a trait
  - `#[my_own_attribute]` : replace code, often wrapping the original code
  - `custom_macro!` : call a function with side effects at compilation time
- Execute arbitrary Rust code on the `TokenStream`
  - Validate SQL queries against remote database
- Consider:
  - Deriving only what you need
  - Manual implementations for hot paths
  - Avoid calling macros in macros
  - Macros with side effects should have an "offline" mode to reduce external calls

Build Scripts

Build scripts and code generation

- `build.rs` runs before compilation
- Can generate code, compile C libraries, ... anything
- Heavy build scripts slow down every build and are hard to cache
- Use change detection to avoid unnecessary reruns
  - https://doc.rust-lang.org/cargo/reference/build-scripts.html#change-detection
- Consider pre-generating code and checking it in

Static VS Dynamic Dispatch

- Polymorphism
  - Functions that accepts any parameter that implements a trait
- Static dispatch (`impl Trait`, trait bounds, generics)
  - Faster runtime: monomorphization
  - Slower compilation: code generated for each type
- Dynamic dispatch (`dyn Trait`)
  - Faster compilation: single implementation
  - Runtime overhead: use the object `vtable`
  - Dyn Compatibility, Object Safe, Trait Object

Monomorphization

- Generic functions are compiled once per concrete type
  - `Vec<T>` creates separate code for `Vec<u8>`, `Vec<String>`, etc.
- Can explode compile times with many type combinations
- Strategies to reduce:
  - Use dynamic dispatch for some generic parameters
  - Share implementations with trait objects
  - Be mindful of deeply nested generics
  - Reduce the use of generics in public API

Common

```rust
trait Compute { fn compute(&self) -> u32; }

struct Add {
    a: u32,
    b: u32
}
impl Compute for Add {
    fn compute(&self) -> u32 { self.a + self.b }
}

struct Sub {
    a: u32,
    b: u32
}
impl Compute for Sub {
    fn compute(&self) -> u32 { self.a - self.b }
}
```

Static Dispatch

```rust
fn compute_static_bound<T: Compute>(object: &T) -> u32 {
    object.compute()
}

fn compute_static_impl(object: &impl Compute) -> u32 {
    object.compute()
}
```

Dynamic Dispatch

```rust
fn compute_dynamic(object: &dyn Compute) -> u32 {
    object.compute()
}
```

Workarounds

Non Rust Files

- Move part of your app to files
  - Display: images, styling, ...
  - Configuration: json, toml, ron, ...
  - Scripts: Lua, Rhai, Python, shaders, ...
- Get the new behavior when rerunning
- Or unload and reload file contents during runtime
  - Use file watchers to detect changes (`notify` crate)
- But no longer checked at compilation

Dynamic hot reloading

Hot reload Rust code as dynamic libraries

- Split application into dylib plugins
- Reload libraries without restarting the app
- Wildly unsafe
- Helpers to make it normally unsafe
  - `libloading` https://docs.rs/libloading/latest/libloading/
  - `hot-lib-reloader` https://github.com/rksm/hot-lib-reloader-rs
- Requires careful API design at library boundaries
- State management across reloads is complex

Wasm modules

WebAssembly as a safe hot-reload runtime

- Compile plugins to Wasm
  - WASI preview 2: Component Model
  - WASI preview 3: Async Functions
  - Wasm Interface Type to share types and functions
- Runtime isolation provides safety guarantees
- Wasm runtime to execute the module
- Slower than native dynamic libraries but safer
- Good for user-provided plugins or untrusted code
  - Or iterating on your own code

Subsecond

Hot patching for instant iteration

- Hot patch running binaries with code changes
- Extremely fast iteration
- Load new functions in memory, then update a jumptable
- With limitations
  - Only works for code in the binary crate being run
  - Can’t change function signature

https://github.com/DioxusLabs/dioxus/tree/main/packages/subsecond

Making It Work For You

Conclusion

Trade-offs to consider

- Disk usage *vs* compilation speed
- Convenience *vs* performance
- Development experience *vs* production optimization
- Safety *vs* iteration speed
- Find the right balance for your project and workflow

Things you should do

Essential actions for faster builds

- Setup your local environment
  - Fast toolchain
  - Compilation cache
- Organize your code into workspace crates
- Configure dev/release profiles appropriately
- Consider dynamic linking for development

Things you should be aware of

Watch out for compile-time bottlenecks

- Build scripts and procedural macros run at compile time
- Heavy codegen increases build duration
- Understand where time is spent and if you can do something about it

Things you should decide on

Architecture decisions impacting build times

- API surface area: what do you expose?
  - Public APIs create more compilation dependencies
  - Smaller interfaces mean faster rebuilds
- Feature flags: how customisable is your crate?
  - Many features increase complexity
  - Optional dependencies add compile-time flexibility
- Balance between ease and compilation speed

Tool Example Outputs

Hyperfine

```
Benchmark 1: cargo build
  Time (mean ± σ):     51.604 s ±  2.397 s    [User: 200.975 s, System: 18.665 s]
  Range (min … max):   49.267 s … 57.191 s    10 runs
```

If you want to measure incremental build time instead of clean build, replace `cargo clean` by `touch src/main.rs` or another of your project rust file

Rust self-profile

Item	Self time	% of total time	Time	Item count	Incremental result hashing t
LLVM_module_codegen_emit_obj	8.55s	24.648	8.55s	256	0.00ns
LLVM_passes	5.05s	14.575	5.05s	1	0.00ns
codegen_module	4.37s	12.604	5.11s	256	0.00ns
normalize_canonicalized_projection_ty	3.18s	9.167	3.21s	15379	4.96ms
typeck	2.40s	6.932	2.92s	6102	20.02ms
codegen_select_candidate	2.02s	5.812	2.05s	29315	4.22ms
type_op_prove_predicate	1.25s	3.611	1.26s	16565	3.45ms
evaluate_obligation	803.84ms	2.319	884.90ms	126613	8.97ms

...