Optimizing Compilation Times for an Elixir Monolithic Codebase

Lívia Almeida Barbosa

Remote Technology, Inc
Wilmington, DE – United States

livia@remote.com

Abstract. *An inefficiently compiled codebase can harm the developer experience, the continuous integration process, and increase organizational costs. Addressing this problem may be challenging, requiring expertise and time to identify, resolve, and prevent the concerns. This paper presents how Remote manages the compilation time of its large-scale monolithic codebase.*

1. Introduction

Remote is a global employment company, where customers can enroll candidates, handle payroll, and manage international teams, reducing barriers and creating more opportunities for both employers and employees across 193 countries. Remote's team also works remotely around the world, ensuring global coverage in all departments.

The company has over 300 software engineers working with a monolithic Elixir codebase containing more than 19,000 files. Elixir, created in 2012, is a functional programming language built on top of the BEAM, Erlang's virtual machine. It leverages Erlang concepts and ecosystem to build reliable, fault-tolerant, and distributed applications. MIX,[1] a build tool built-in in Elixir, provides tasks for creating, compiling, testing, managing project dependencies, and more. One can run `mix compile` and the Elixir files will be compiled into BEAM bytecode files [The Elixir Team 2025b].

Given the large number of files, it is expected that developers face compilation challenges. To address them, in October 2022, the backend team created a dedicated Slack[2] channel and wrote wiki documents to asynchronously discuss possible solutions and coordinate efforts. According to employees' feedback, the initiatives reduced by at least a third the compilation time and improved knowledge sharing. In the following sections, we discuss and share the actions taken and the challenges faced throughout the process. They were not performed in a particular order, and many started in parallel.

2. Moving compile-time environment configurations to runtime

Elixir provides two configuration entry points. The first configuration file (`config/config.exs`) is read at build time, before compiling and loading dependencies. The second one (`config/runtime.exs`) is read after the application and dependencies are compiled, and it configures how the application works at runtime [The Elixir Team 2025a]. Initially, the team heavily relied on the configuration file read at build time to place configuration and environment variables. Thus, any configuration

[1] https://hexdocs.pm/mix/Mix.html
[2] https://slack.com

change would trigger a full recompilation of the project. At the time, the codebase had more than 4,500 files and would take around 3 minutes to be recompiled locally, as reported by employees. After discussions, the team noticed there was no need to control how most configurations were compiled, moving them to be runtime configurations.

3. Removing stale files

Refactors, feature flags, and one-off scripts often leave behind stale files and unused code paths, which leads to unnecessary compilation. The team regularly works on cleanup tasks to remove such code. An automated approach to identify unused code paths with MIX_UNUSED[3] was tested; however, this resulted in a high number of false positives. Ultimately, the team relies on the developers' commitment to maintain a clean codebase, as this is a continuous task.

4. Handling compile-time dependencies

Transitive compile-time dependencies have been identified as the biggest pain point for the team, leading to unnecessary recompilation of files that don't directly depend on changed files. *Cyclic* dependencies are also carefully observed, as these can increase recompilation time and the likelihood of having transitive compile-time dependencies.

To identify *transitive* dependencies, one can use the `mix xref graph` command to get the project's dependency graph. Next, evaluate if the dependency is necessary. If a dependency must be available at compile time or if moving it to runtime significantly impacts performance, it is advisable to retain it as a compile-time dependency. A recommended approach is to create separate modules, extracting the information that causes the transitive dependency, and having a direct dependency instead. Otherwise, if there are no performance penalties, dependencies can be replaced with runtime calls.

To manage *cyclic* dependencies, one can either replace the cyclic call with an alternative function or move it to a separate module that breaks the cycle, similar to the approach taken to address transitive dependencies.

Applying these changes can be difficult and time-consuming. At Remote, they are still in progress. To prevent new transitive or cyclic dependencies in the project, the team sets code analysis rules, educates team members, and shares the reasoning behind the endeavor and the intermediary results to motivate other developers.

5. Introducing domain separation

In 2022, with an already substantial codebase, the backend team began enforcing domain separation using the BOUNDARY[4] library. The goal was to align the code structure with the respective team responsibilities, encourage interactions through top-level public APIs, and identify and reduce cross-domain references.

Approximately 2 years later, the team began to express concerns regarding increased friction in the development process. The changes implemented resulted in higher compilation times and prejudiced developer productivity, as team members needed to

[3] https://hexdocs.pm/mix_unused

[4] https://hexdocs.pm/boundary/Boundary.html

learn new guidelines, update existing code, and address violations that emerged following the addition of the library. Team members even expressed that "*we didn't have the benefits of a monolith, nor the benefits of a microservice architecture.*"

In 2025, engineers revisited the constraints around the library usage. Existing BOUNDARY related code remained, but its enforcement has been relaxed, and new domains should not use the library unless justified and necessary. Self-contained domains still benefit from the isolation and are recommended to leverage the BOUNDARY library, whereas the majority of other domains are intertwined in some fashion and will take advantage of the recent instruction. Teams are supposed to manage boundaries via API design, static code analysis tools, code reviews, and documentation.

Efforts are underway to shift the application toward a modular domain-driven monolith. They include establishing well-defined domain boundaries, migrating code and isolating domains, decoupling modules using hooks and events for intercommunication, and achieving data isolation. However, defining domains can be complex, the adoption can be challenging, and applying changes might be slow due to the size of the codebase.

6. Seeking expert guidance

If little improvement is shown despite optimization efforts, it is advisable to consult with experienced professionals. The Elixir community is known for being welcoming and supportive, providing valuable resources. For instance, a few blog posts inspired the initiatives at Remote.[5][6][7] Moreover, one of Remote's employees identified an optimization opportunity that reduced the compilation time and contributed to the Elixir language codebase,[8] receiving positive recognition from the language's creator.

Besides keeping the language up-to-date to benefit from improvements provided by the Elixir Core Team, alternative solutions include training or hiring an engineer for dedicated assistance, engaging with specialized services, or consulting with an advisor.

7. Conclusion

In this paper, we assessed how Remote engineers maintain adequate compilation times for its large codebase. Moving compile-time configuration to be read at runtime, removing stale files to avoid unnecessary compilation, breaking transitive compile-time and cyclic dependencies, trying to keep separated domains, and investing in high-quality guidance for the team are the steps that resulted in positive outcomes in the past years. With this work, we hope to generate insights for further academic investigations.

Acknowledgments

This paper reflects collaborative efforts by Remote software engineers, with special thanks to Staff Engineer Alex Naser, who led most of the aforementioned initiatives.

[5] http://dashbit.co/blog/how-to-debug-elixir-erlang-compiler-performance

[6] https://medium.com/multiverse-tech/how-to-speed-up-your-elixir-compile-times-part-1-understanding-elixir-compilation-64d44a32ec6e

[7] https://szajbus.dev/elixir/2020/04/14/understanding-and-fixing-recompilation-in-elixir-projects.html

[8] https://github.com/elixir-lang/elixir/pull/12287

References

The Elixir Team (2025a). Configuration and releases. https://hexdocs.pm/elixir/config-and-releases.html#configuration. Accessed: 2025-06-03.

The Elixir Team (2025b). Modules and functions. https://hexdocs.pm/elixir/modules-and-functions.html#compilation. Accessed: 2025-06-03.