The Pragmatic Programmers

Rails Scales!

Practical Techniques for Performance and Growth

Cristian Planas

edited by Michael Swaine

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Introduction

This book has a superhero origin story. Let me tell it to you.

The Origin Story

I started writing Ruby in the late 2000s, while I was still in college. Those days, I was surrounded by fellow students who were becoming entrepreneurs, and soon I decided to try my luck. In 2012, I became one of the co-founders and the Chief Technical Officer of PlayfulGaming, a company that supported only one product: a game called Playfulbet. Playfulbet quickly became *very* popular. Within two years, the game had over half a million registered users.

This was a huge challenge for a 25-year-old CTO – especially considering that I was also the *only* engineer and the only technical person on the team. I took care of everything: from infra to front-end, from the Android app to database management, from writing Ruby wrappers around third-party APIs to the management of the iOS account for our iPhone app… which was also created by me.

All this was a titanic effort for one developer, only made possible by Ruby on Rails. Our web app had an extremely simple front-end and the mobile apps just rendered the mobile version of the web app. In this way, I was able to spend most of my time coding Ruby, the language that prioritizes developer productivity and happiness. And I *was* reasonably happy… and also crazily productive.

Still, Playfulbet suffered from very intense growing pains. Make that: we had a lot of trouble making it scale. The game became unplayable at peak hours, when the Rails app was just unable to respond to so many simultaneous requests. I learned a lot during that period, and many of the techniques that I will share with you in this book were used to keep Playfulbet afloat during those years of rapid growth. Still, our growth was just too spectacular: my Rails application couldn't manage it. This was the beginning of my fascination

with scalability problems, and the reason why I decided to write this book. Was it possible that Rails just didn't scale?

We Need to Talk about the Rails Learning Curve

The idea that "Rails does not scale" is, unfortunately, quite extended. A couple of months ago, while talking with the founder of a start-up about this book, he told me "The only thing I have heard about Rails is that it doesn't scale!". That this idea is around obviously does not make anybody happy in the Rails (and by extension, Ruby) community. But is it a fair assessment?

In short, my answer is no, it's not a fair assessment. Benchmarks[1] show that Rails does basically the same as similar technologies, like Django (Python) or Laravel (PHP). Rails *does* scale. In truth, Rails is a battle-tested technology that has proven itself in enabling the growth of companies with planetary-scale needs, like Shopify, GitHub, and Zendesk.

So if Rails has the demostrable capacity of scaling, where does the myth of "Rails does not scale" comes from? I have a theory.

Rails famously follows the design paradigm of "convention over configuration". With it, Rails reduces the cognitive workload in projects by providing sensible defaults. This reduces complexity and helps increase productivity. Other concepts embraced by Rails nudge in the same direction: "batteries included" means that Rails developers have access to many tools that bring a rich set of functionalities to their applications with a very low implementation and/or integration cost. On top of that, there is the famous (or infamous, depending on who you ask) Rails "magic": the fact that Rails sets a series of implicit behaviors, conventions, and dependencies that increase development efficiency by abstracting away much of the complexity – and they may also prove obscure when the time comes to debug or when you need to go against the implicit convention.

While other successful web frameworks have also followed this path, Rails has been the most firmly committed to it, and also the most successful. This success meant that many inexperienced developers (like I was when I worked in my start-up) were able to get very far – thanks to the Rails "magic", supported by Rails extremely smooth learning curve.

Until the time arrived to scale the application. At a talk in RailsConf 2022[2], I joked that what the community seemed to be asking for is `active_scale`, a gem

1. https://www.techempower.com/benchmarks/#hw=ph&test=composite&section=data-r22
2. https://www.youtube.com/watch?v=mJw3al4Ms2o

that would do the whole job for them. You just create a new initializer file and write:

```
ActiveScale::Rails.scale!(:sufficient_but_not_too_much)
```

And that's it. You can go back to writing business logic!

Regrettably, active_scale does not exist. It can't exist. There are multiple reasons for this: a lot of the scaling work is done outside of the scope of the Rails application itself, like database indexing and product design. Moreover, scaling is also highly context-specific, and the performance bottlenecks vary.

Fortunately, while it's true that we can't create active_scale there are a number of techniques that can be used to scale web applications, practices that are completely applicable to Rails applications.

Techniques for a Rails Renaissance

Getting back to my story, I eventually realized that the best way of learning how to scale would be to join a company that had managed to do it. That's why I joined Zendesk. It was a dramatic change moving from an organization in which I was the only engineer to one that has a product development team comprising over a thousand professionals.

This proved to be a smarter plan than I even realized. I now believe that an engineer needs three things to develop into a great professional. The first is a strong sense of ownership and responsibility, which you will definitely get working in a start-up. The second is being mentored by experts in different areas, which is something that can best be achieved by working in a big company. The third is time. You can be a great coder very early in your career; but you can't be an effective engineering leader without experience.

In my opinion, many debates in software architecture tend to be highly dependent on the broader social and economic context: Only experience can show you that there is not a general best solution, but only a best solution for these particular circumstances. A successful engineering leader needs not only a broad knowledge of the options to satisfy the requirements but also wisdom to separate the wheat from the chaff and pick the solution that will best fit the needs of the organization.

In the last 15 years, we have seen a great rise in the popularity of microservice architectures: while they have obvious upsides for scalability, flexibility, and autonomy, I cannot help thinking that their popularity has benefitted from the financial environment of the last decade. In other words, it might be, partly, a zero-interest rate phenomenon. Having so much money injected into

tech allowed the hiring of very big teams for comparatively simple products, which helped to cover some of the trade-offs of microservices, like the higher complexity and increased cognitive load needed to manage them. If we are headed into more frugal times, this will have to change.

Still, requirements and expectations in 2024 are different from 2012. Modern applications are expected to be able to serve more users and have better performance than they had 12 years ago. If we are going to have a Rails renaissance, we are going to have to write Rails applications in which performance is a first-class feature from day one. Hopefully, this book can help.