How we built Pingora, the proxy that connects Cloudflare to the Internet

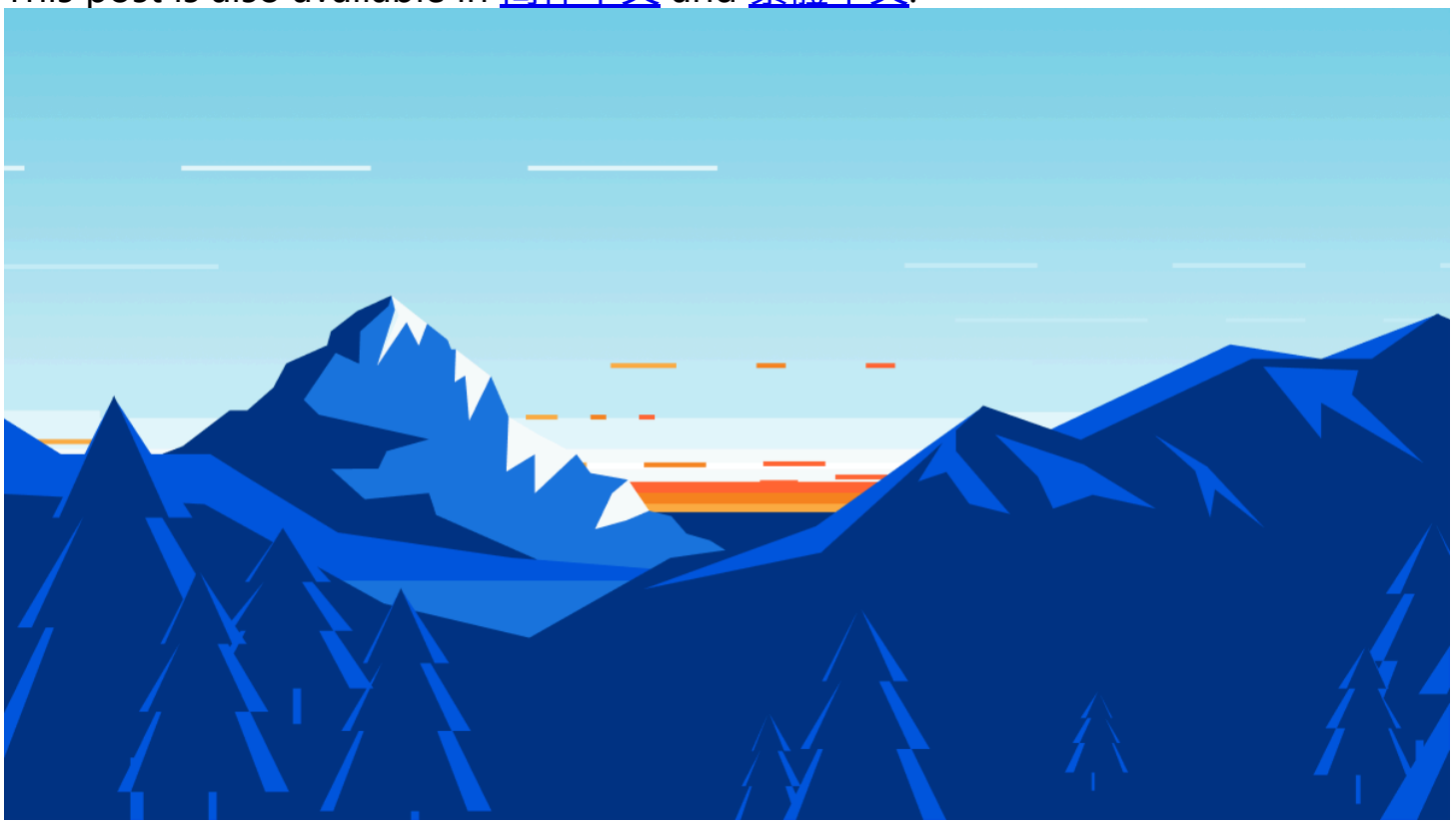
2022-09-14

 Yuchen Wu

 Andrew Hauck

8 min read

This post is also available in 简体中文 and 繁體中文.



Introduction

Today we are excited to talk about Pingora, a new HTTP proxy we've built in-house using Rust that serves over 1 trillion requests a day, boosts our performance, and enables many new features for Cloudflare customers, all while requiring only a third of the CPU and memory resources of our previous proxy infrastructure.

As Cloudflare has scaled we've outgrown NGINX. It was great for many years, but over time its limitations at our scale meant building something new made sense. We could no longer get the performance we needed nor did NGINX have the features we needed for our very complex environment.

Many Cloudflare customers and users use the Cloudflare global network as a proxy between HTTP clients (such as web browsers, apps, IoT devices and more) and servers. In the past, we've talked a lot about how browsers and other user agents connect to our network, and we've developed a lot of technology and implemented new protocols (see QUIC and optimizations for http2) to make this leg of the connection more efficient.

Today, we're focusing on a different part of the equation: the service that proxies traffic between our network and servers on the Internet. This proxy service powers our CDN, Workers fetch, Tunnel, Stream, R2 and many, many other features and products.

Let's dig in on why we chose to replace our legacy service and how we developed Pingora, our new system designed specifically for Cloudflare's customer use cases and scale.

Why build yet another proxy

Over the years, our usage of NGINX has run up against limitations. For some limitations, we optimized or worked around them. But others were much harder to overcome.

Architecture limitations hurt performance

The NGINX worker (process) architecture has operational drawbacks for our use cases that hurt our performance and efficiency.

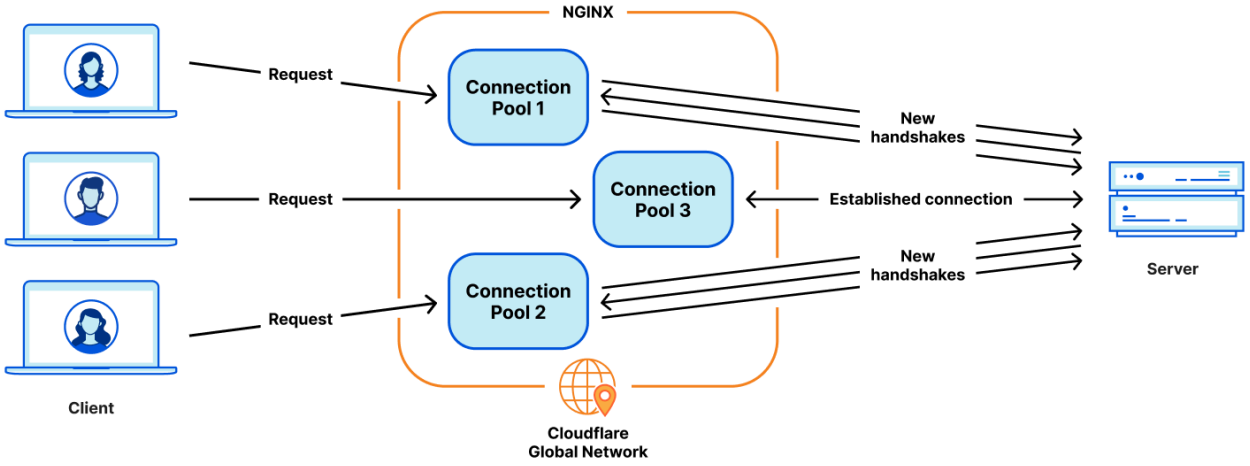
First, in NGINX each request can only be served by a single worker. This results in unbalanced load across all CPU cores, which leads to slowness.

Because of this request-process pinning effect, requests that do CPU heavy or blocking IO tasks can slow down other requests. As those blog posts attest we've

spent a lot of time working around these problems.

The most critical problem for our use cases is poor connection reuse. Our machines establish TCP connections to origin servers to proxy HTTP requests. Connection reuse speeds up TTFB (time-to-first-byte) of requests by reusing previously established connections from a connection pool, skipping TCP and TLS handshakes required on a new connection.

However, the NGINX connection pool is per worker. When a request lands on a certain worker, it can only reuse the connections within that worker. When we add more NGINX workers to scale up, our connection reuse ratio gets worse because the connections are scattered across more isolated pools of all the processes. This results in slower TTFB and more connections to maintain, which consumes resources (and money) for both us and our customers.



As mentioned in past blog posts, we have workarounds for some of these issues. But if we can address the fundamental issue: the worker/process model, we will resolve all these problems naturally.

Some types of functionality are difficult to add

NGINX is a very good web server, load balancer or a simple gateway. But Cloudflare does way more than that. We used to build all the functionality we

needed around NGINX, which is not easy to do while trying not to diverge too much from NGINX upstream codebase.

For example, when retrying/failing over a request, sometimes we want to send a request to a different origin server with a different set of request headers. But that is not something NGINX allows us to do. In cases like this, we spend time and effort on working around the NGINX constraints.

Meanwhile, the programming languages we had to work with didn't provide help alleviating the difficulties. NGINX is purely in C, which is not memory safe by design. It is very error-prone to work with such a 3rd party code base. It is quite easy to get into memory safety issues, even for experienced engineers, and we wanted to avoid these as much as possible.

The other language we used to complement C is Lua. It is less risky but also less performant. In addition, we often found ourselves missing static typing when working with complicated Lua code and business logic.

And the NGINX community is not very active, and development tends to be "behind closed doors".

Choosing to build our own

Over the past few years, as we've continued to grow our customer base and feature set, we continually evaluated three choices:

1. Continue to invest in NGINX and possibly fork it to tailor it 100% to our needs. We had the expertise needed, but given the architecture limitations mentioned above, significant effort would be required to rebuild it in a way that fully supported our needs.

2. Migrate to another 3rd party proxy codebase. There are definitely good projects, like envoy and others. But this path means the same cycle may repeat in a few years.

3. Start with a clean slate, building an in-house platform and framework. This choice requires the most upfront investment in terms of engineering

effort.

We evaluated each of these options every quarter for the past few years. There is no obvious formula to tell which choice is the best. For several years, we continued with the path of the least resistance, continuing to augment NGINX. However, at some point, building our own proxy's return on investment seemed worth it. We made a call to build a proxy from scratch, and began designing the proxy application of our dreams.

The Pingora Project

Design decisions

To make a proxy that serves millions of requests per second fast, efficient and secure, we have to make a few important design decisions first.

We chose Rust as the language of the project because it can do what C can do in a memory safe way without compromising performance.

Although there are some great off-the-shelf 3rd party HTTP libraries, such as hyper, we chose to build our own because we want to maximize the flexibility in how we handle HTTP traffic and to make sure we can innovate at our own pace.

At Cloudflare, we handle traffic across the entire Internet. We have many cases of bizarre and non-RFC compliant HTTP traffic that we have to support. This is a common dilemma across the HTTP community and web, where there is tension between strictly following HTTP specifications and accommodating the nuances of a wide ecosystem of potentially legacy clients or servers. Picking one side can be a tough job.

HTTP status codes are defined in RFC 9110 as a three digit integer, and generally expected to be in the range 100 through 599. Hyper was one such implementation. However, many servers support the use of status codes between 599 and 999. An issue had been created for this feature, which explored various sides of the debate. While the hyper team did ultimately accept that change, there would have been valid reasons for them to reject such an ask, and this was only one of many cases of noncompliant behavior we needed to support.

In order to satisfy the requirements of Cloudflare's position in the HTTP ecosystem, we needed a robust, permissive, customizable HTTP library that can survive the wilds of the Internet and support a variety of noncompliant use cases. The best way to guarantee that is to implement our own.

The next design decision was around our workload scheduling system. We chose multithreading over multiprocessing in order to share resources, especially connection pools, easily. We also decided that work stealing was required to avoid some classes of performance problems mentioned above. The Tokio async runtime turned out to be a great fit for our needs.

Finally, we wanted our project to be intuitive and developer friendly. What we build is not the final product, and should be extensible as a platform as more features are built on top of it. We decided to implement a "life of a request" event based programmable interface similar to NGINX/OpenResty. For example, the "request filter" phase allows developers to run code to modify or reject the request when a request header is received. With this design, we can separate our business logic and generic proxy logic cleanly. Developers who previously worked on NGINX can easily switch to Pingora and quickly become productive.

Pingora is faster in production

Let's fast-forward to the present. Pingora handles almost every HTTP request that needs to interact with an origin server (for a cache miss, for example), and we've collected a lot of performance data in the process.

First, let's see how Pingora speeds up our customer's traffic. Overall traffic on Pingora shows 5ms reduction on median TTFB and 80ms reduction on the 95th percentile. This is not because we run code faster. Even our old service could handle requests in the sub-millisecond range.

The savings come from our new architecture which can share connections across all threads. This means a better connection reuse ratio, which spends less time on TCP and TLS handshakes.

Across all customers, Pingora makes only a third as many new connections per second compared to the old service. For one major customer, it increased the connection reuse ratio from 87.1% to 99.92%, which reduced new connections to their origins by 160x. To present the number more intuitively, by switching to Pingora, we save our customers and users 434 years of handshake time every day.

More features

Having a developer friendly interface engineers are familiar with while eliminating the previous constraints allows us to develop more features, more quickly. Core functionality like new protocols act as building blocks to more products we can offer to customers.

As an example, we were able to add HTTP/2 upstream support to Pingora without major hurdles. This allowed us to offer gRPC to our customers shortly afterwards. Adding this same functionality to NGINX would have required significantly more engineering effort and might not have materialized.

More recently we've announced Cache Reserve where Pingora uses R2 storage as a caching layer. As we add more functionality to Pingora, we're able to offer new products that weren't feasible before.

More efficient

In production, Pingora consumes about 70% less CPU and 67% less memory compared to our old service with the same traffic load. The savings come from a

few factors.

Our Rust code runs more efficiently compared to our old Lua code. On top of that, there are also efficiency differences from their architectures. For example, in NGINX/OpenResty, when the Lua code wants to access an HTTP header, it has to read it from the NGINX C struct, allocate a Lua string and then copy it to the Lua string. Afterwards, Lua has to garbage-collect its new string as well. In Pingora, it would just be a direct string access.

The multithreading model also makes sharing data across requests more efficient. NGINX also has shared memory but due to implementation limitations, every shared memory access has to use a mutex lock and only strings and numbers can be put into shared memory. In Pingora, most shared items can be accessed directly via shared references behind atomic reference counters.

Another significant portion of CPU saving, as mentioned above, is from making fewer new connections. TLS handshakes are expensive compared to just sending and receiving data via established connections.

Safer

Shipping features quickly and safely is difficult, especially at our scale. It's hard to predict every edge case that can occur in a distributed environment processing millions of requests a second. Fuzzing and static analysis can only mitigate so much. Rust's memory-safe semantics guard us from undefined behavior and give us confidence our service will run correctly.

With those assurances we can focus more on how a change to our service will interact with other services or a customer's origin. We can develop features at a higher cadence and not be burdened by memory safety and hard to diagnose crashes.

When crashes do occur an engineer needs to spend time to diagnose how it happened and what caused it. Since Pingora's inception we've served a few hundred trillion requests and have yet to crash due to our service code.

In fact, Pingora crashes are so rare we usually find unrelated issues when we do encounter one. Recently we discovered a kernel bug soon after our service started crashing. We've also discovered hardware issues on a few machines, in the past ruling out rare memory bugs caused by our software even after significant debugging was nearly impossible.

Conclusion

To summarize, we have built an in-house proxy that is faster, more efficient and versatile as the platform for our current and future products.

We will be back with more technical details regarding the problems we faced, the optimizations we applied and the lessons we learned from building Pingora and rolling it out to power a significant portion of the Internet. We will also be back with our plan to open source it.

Pingora is our latest attempt at rewriting our system, but it won't be our last. It is also only one of the building blocks in the re-architecting of our systems.

Interested in joining us to help build a better Internet? Our engineering teams are hiring.

Discuss on X

Discuss on Hacker News

Discuss on Reddit

Rust NGINX Performance Pingora

Follow on X

Cloudflare | @cloudflare

RELATED POSTS

December 18, 2025 6:00 AM

Announcing support for GROUP BY, SUM, and other aggregation queries in R2 SQL

Cloudflare's R2 SQL, a distributed query engine, now supports aggregations. Explore how we built distributed GROUP BY execution, using scatter-gather and shuffling strategies to run analytics directly over your R2 Data Catalog....

By Jérôme Schneider, Nikita Lapkov, Marc Selwan

R2, Data, Edge Computing, Rust, Serverless, SQL

October 28, 2025 6:00 AM

Keeping the Internet fast and secure: introducing Merkle Tree Certificates

Cloudflare is launching an experiment with Chrome to evaluate fast, scalable, and quantum-ready Merkle Tree Certificates, all without degrading performance or changing WebPKI trust

relationships....

By Luke Valenta, Christopher Patton, Vânia Gonçalves, Bas Westerbaan

Post-Quantum, Research, Cryptography, Security, TLS, Chrome, Google, IETF, Transparency, Rust, Open Source, Cloudflare Workers

October 21, 2025 6:00 AM

A deep dive into BPF LPM trie performance and optimization

This post explores the performance of BPF LPM tries, a critical data structure used for IP matching. ...

By Matt Fleming, Jesper Brouer

Deep Dive, eBPF, IPv4, IPv6, Linux, Performance

September 29, 2025 7:00 AM

15 years of helping build a better Internet: a look back at Birthday Week 2025

Rust-powered core systems, post-quantum upgrades, developer access for students, PlanetScale integration, open-source partnerships, and our biggest internship program ever — 1,111 interns in 2026....

By Nikita Cano, Korinne Alpers

Birthday Week, Partners, Developer Platform, Workers Launchpad, Performance, Security, Cache, Speed, Developers, AI, 1.1.1.1, Application Security, Application Services, Bots, CDN, Cloudflare for Startups, Cloudflare One, Cloudflare Zero Trust, Cloudflare Workers