Tuning Performance for Deployment

This guide covers performance and concurrency configuration for deploying your production Ruby on Rails application.

After reading this guide, you will know:

- Whether to use Puma, the default application server
- How to configure important performance settings for Puma
- How to begin performance testing your application settings

This guide focuses on web servers, which are the primary performance-sensitive component of most web applications. Other components like background jobs and WebSockets can be tuned but won't be covered by this guide.

More information about how to configure your application can be found in the Configuration Guide.

This guide assumes you are running MRI, the canonical implementation of Ruby also known as CRuby. If you're using another Ruby implementation such as JRuby or TruffleRuby, most of this guide doesn't apply. If needed, check sources specific to your Ruby implementation.

1 Choosing an Application Server

Puma is Rails' default application server and the most commonly used server across the community. It works well in most cases. In some cases, you may wish to change to another.

An application server uses a particular concurrency method. For example, Unicorn uses processes, Puma and Passenger are hybrid process- and thread-based concurrency, and Falcon uses fibers.

A full discussion of Ruby's concurrency methods is beyond the scope of this document, but the key tradeoffs between processes and threads will be presented. If you want to use a method other than processes and threads, you will need to use a different application server.

This guide will focus on how to tune Puma.

2 What to Optimize for?

In essence, tuning a Ruby web server is making a tradeoff between multiple properties such as memory usage, throughput, and latency.

The throughput is the measure of how many requests per second the server can handle, and latency is the measure of how long individual requests take (also referred to as response time).

Some users may want to maximize throughput to keep their hosting cost low, some other users may want to minimize latency to offer the best user experience, and many users will search for some compromise somewhere in the middle.

It is important to understand that optimizing for one property will generally hurt at least another one.

2.1 Understanding Ruby's Concurrency and Parallelism

CRuby has a Global Interpreter Lock, often called the GVL or GIL. The GVL prevents multiple threads from running Ruby code at the same time in a single process. Multiple threads can be waiting on network data, database operations, or some other non-Ruby work generally referred to as I/O operations, but only one can actively run Ruby code at a time.

This means that thread-based concurrency allows for increased throughput by concurrently processing web requests whenever they do I/O operations, but may degrade latency whenever an I/O operation completes. The thread that performed it may have to wait before it can resume executing Ruby code. Similarly, Ruby's garbage collector is "stop-the-world" so when it triggers all threads have to stop.

This also means that regardless of how many threads a Ruby process contains, it will never use more than a single CPU core.

Because of this, if your application only spends 50% of its time doing I/O operations, using more than 2 or 3 threads per process may severely hurt latency, and the gains in throughput will quickly hit diminishing returns.

Generally speaking, a well-crafted Rails application that isn't suffering from slow SQL queries or N+1 problems doesn't spend more than 50% of its time doing I/O operations, hence is unlikely to benefit from more than 3 threads. However, some applications that do call third-party APIs inline may spend a very large proportion of their time doing I/O operations and may benefit from more threads than that.

The way to achieve true parallelism with Ruby is to use multiple processes. As long as there is a free CPU core, Ruby processes don't have to wait on one another before resuming execution after an I/O operation is complete. However, processes only share a fraction of their memory via copy-on-write, so one additional process uses more memory than an additional thread would.

Note that while threads are cheaper than processes, they are not free, and increasing the number of threads per process, also increases memory usage.

2.2 Practical Implications

Users interested in optimizing for throughput and server utilization will want to run one process per CPU core and increase the number of threads per process until the impact on latency is deemed too important.

Users interested in optimizing for latency will want to keep the number of threads per process low. To optimize for latency even further, users can even set the thread per process count to `1` and run `1.5` or `1.3` process per CPU core to account for when processes are idle waiting for I/O operations.

It is important to note that some hosting solutions may only offer a relatively small amount of memory (RAM) per CPU core, preventing you from running as many processes as needed to use all CPU cores. However, most hosting solutions have different plans with different ratios of memory and CPU.

Another thing to consider is that Ruby memory usage benefits from economies of scale thanks to copy-on-write. So `2` servers with `32` Ruby processes each will use less memory per CPU core than `16` servers with `4` Ruby processes each.

3 Configurations

3.1 Puma

The Puma configuration resides in the `config/puma.rb` file. The two most important Puma configurations are the number of threads per process, and the number of processes, which Puma calls `workers`.

The number of threads per process is configured via the `thread` directive. In the default generated configuration, it is set to `3`. You can modify it either by setting the `RAILS_MAX_THREADS` environment variable or simply editing the configuration file.

The number of processes is configured by the `workers` directive. If you use more than one thread per process, then it should be set to how many CPU cores are available on the server, or if the server is running multiple applications, to how many cores you want the application to use. If you only use one thread per worker, then you can increase it to above one per process to account for when workers are idle waiting for I/O operations.

You can configure number of Puma workers by setting the `WEB_CONCURRENCY` environment variable.

3.2 YJIT

Recent Ruby versions come with a Just-in-time compiler called `YJIT`.

Without going into too many details, JIT compilers allow to execute code faster, at the expense of using some more memory. Unless you really cannot spare this extra memory usage, it is highly recommended to enable YJIT.

As for Rails 7.2, if your application is running on Ruby 3.3 or superior, YJIT will automatically be enabled by Rails by default. Older versions of Rails or Ruby have to enable it manually, please refer to the `YJIT` documentation about how to do it.

If the extra memory usage is a problem, before entirely disabling YJIT, you can try tuning it to use less memory via the `--yjit-exec-mem-size` configuration.

3.3 Memory Allocators and Configuration

Because of how the default memory allocator works on most Linux distributions, running Puma with multiple threads can lead to an unexpected increase in memory usage caused by memory fragmentation. In turn, this increased memory usage may prevent your application from fully utilizing the server CPU cores.

To alleviate this problem, it is highly recommended to configure Ruby to use an alternative memory allocator: jemalloc.

The default Dockerfile generated by Rails already comes preconfigured to install and use `jemalloc`. But if your hosting solution isn't Docker based, you should look into how to install and enable jemalloc there.

If for some reason that isn't possible, a less efficient alternative is to configure the default allocator in a way that reduces memory fragmentation by setting `MALLOC_ARENA_MAX=2` in your environment. Note however that this might make Ruby slower, so `jemalloc` is the preferred solution.

4 Performance Testing

Because every Rails application is different, and that every Rails user may want to optimize for different properties, it is impossible to offer a default configuration or guidelines that works best for everyone.

Hence, the best way to choose your application's settings is to measure the performance of your application, and adjust the configuration until it is satisfactory for your goals.

This can be done with a simulated production workload, or directly in production with live application traffic.

Performance testing is a deep subject. This guide gives only simple guidelines.

4.1 What to Measure

Throughput is the number of requests per second that your application successfully processes. Any good load testing program will measure it. A throughput is normally a single number expressed in "requests per second".

Latency is the delay from the time the request is sent until its response is successfully received, generally expressed in milliseconds. Each individual request will have its own latency.

Percentile latency gives the latency where a certain percentage of requests have better latency than that. For instance, `P90` is the 90th-percentile latency. The `P90` is the latency for a single load test where only 10% of requests took longer than that to process. The `P50` is the latency such that half your requests were slower, also called the median latency.

"Tail latency" refers to high-percentile latencies. For instance, the `P99` is the latency such that only 1% of your requests were worse. `P99` is a tail latency. `P50` is not a tail latency.

Generally speaking, the average latency isn't a good metric to optimize for. It is best to focus on median (`P50`) and tail (`P95` or `P99`) latency.

4.2 Production Measurement

If your production environment includes more than one server, it can be a good idea to do A/B testing there. For instance, you could run half of the servers with `3` threads per process, and the other half with `4` threads per process, and then use an application performance monitoring service to compare the throughput and latency of the two groups.

Application performance monitoring services are numerous, some are self-hosted, some are cloud solutions, and many offer a free tier plan. Recommending a particular one is beyond the scope of this guide.

4.3 Load Testers

You will need a load testing program to make requests of your application. This can be a dedicated load testing program of some kind, or you can write a small application to make HTTP requests and track how long they take. You should not normally check the time in your Rails log file. That time is only how long Rails took to process the request. It does not include time taken by the application server.

Sending many simultaneous requests and timing them can be difficult. It is easy to introduce subtle measurement errors. Normally you should use a load testing program, not write your own. Many load testers are simple to use and many excellent load testers are free.

4.4 What You Can Change

You can change the number of threads in your test to find the best tradeoff between throughput and latency for your application.

Larger hosts with more memory and CPU cores will need more processes for best usage. You can vary the size and type of hosts from a hosting provider.

Increasing the number of iterations will usually give a more exact answer, but require longer for testing.

You should test on the same type of host that will run in production. Testing on your development machine will only tell you what settings are best for that development machine.

4.5 Warmup

Your application should process a number of requests after startup that are not included in your final measurements. These requests are called "warmup" requests, and are usually much slower than later "steady-state" requests.

Your load testing program will usually support warmup requests. You can also run it more than once and throw away the first set of times.

You have enough warmup requests when increasing the number does not significantly change your result. The theory behind this can be complicated but most common situations are straightforward: test several times with different amounts of warmup. See how many warmup iterations are needed before the results stay roughly the same.

Very long warmup can be useful for testing memory fragmentation and other issues that happen only after many requests.

4.6 Which Requests

Your application probably accepts many different HTTP requests. You should begin by load testing with just a few of them. You can add more kinds of requests over time. If a particular kind of request is too slow in your production application, you can add it to your load testing code.

A synthetic workload cannot perfectly match your application's production traffic. It is still helpful for testing configurations.

4.7 What to Look For

Your load testing program should allow you to check latencies, including percentile and tail latencies.

For different numbers of processes and threads, or different configurations in general, check the throughput and one or more latencies such as `P50`, `P90`, and `P99`. Increasing the threads will improve throughput up to a point, but worsen latency.

Choose a tradeoff between latency and throughput based on your application's needs.

Feedback

You're encouraged to help improve the quality of this guide.

Please contribute if you see any typos or factual errors. To get started, you can read our documentation contributions section.

You may also find incomplete content or stuff that is not up to date. Please do add any missing documentation for main. Make sure to check Edge Guides first to verify if the issues are already fixed or not on the main branch. Check the Ruby on Rails Guides Guidelines for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please open an issue.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome on the official Ruby on Rails Forum.

Chapters