Hello Huntregs team,

Thanks again for bringing me on to look at your setup. Overall, we've got a very simple app that needs a little "operational" love as far as improving reliability and robustness to changes in usage and load.

The app itself is extremely simple: just 10,000 lines of Ruby. That's great! Most apps I have to work with are far more complex, and for the limited size of your full-time team, I think having the app be a reasonable size like this is an incredible asset. Most teams would be buried under ~5 years of tech debt and ~100k+ lines of app by this point, and you're not. Awesome!

Your load is also very low. The Wordpress site sees 4,000 views a day on a big day, and based on Sentry and your sample rate, you're probably in the ~100 requests/minute range for the Rails application. This is also a great place to be in, as you have so much room to scale vertically and horizontally!

I view retainer engagements as a long-term focus on a particular set of metrics. I said I would propose a set of metrics to be our yardsticks for this effort. Here they are:

- **Web request queue time SLO: >99.9% p95 500ms or less**. Web request queue time is how long a request spends waiting (queueing) for a free Puma thread to start processing it. This queueing occurs when a request is routed to an ECS task but is not immediately picked up by a free Sidekiq thread. This is the fundamental signal for all web scaling; we are trying to manage this number in response to changes in traffic.
- **Sidekiq job latency SLO: All queues meeting latency SLO >99.9% of the time**. All jobs spend some amount of time waiting in a Redis queue. Each job has, de facto, some level of acceptable amount of queue latency. I'll discuss this more later, but we will assign an SLO to every job (actually to every queue), so our metric will be what % of the time we are achieving those SLOs.
- **99.9% uptime/success rate SLO**. Probably your biggest danger when scaling up transactions/sec is that an underlying 3rd-party explodes. To capture that dependency, I'd like to focus on keeping the overall error rate for the Rails service below 0.1% of requests.

We would track these SLOs on Cloudwatch.

There are other, secondary metrics which kind of by definition have to be "good" in order for these three metrics to succeed, but these high-level metrics "contain" those secondary metrics within them. For example, we don't need a separate database uptime metric, because if your database goes down for 3 hours straight both of these SLOs will be toast for the quarter.

I arrange my recommendations here based on which of those SLOs it best applies to.

Web

Your web response times in general look great. The app is simple enough that you don't need major overhauls to get lower response times. My recommendations here focus on other issues as a result.

Recommendation: Upgrade to Ruby 3.2+ Ruby 3.2 and up are where YJIT started to get really effective, and YJIT is just a free 20-30% speed improvement for most Rails apps. If you use Ruby 3.2+ with Rails frameworks defaults at 7.1 and up, you get YJIT turned on automatically with no other changes.

The app is so small that I don't anticipate this being a difficult upgrade.

Recommendation: Remove most gem constraints. Puma, pg, redis, are major behind. The app has a lot of very strict pessimistic gem version requires that have resulted in a number of dependencies getting very, very far out of date.

Here are the most important ones that I recommend upgrading immediately for performance reasons:

- Puma version 3.6 is 5 years old. I personally have shipped a lot of improvements to Puma in 5 years, namely a lot of important security patches too!
- Redis gem version is 2 years old. Byroot has done a lot of good work on this gem in that time.
- Pg version (the critical gem for interacting with your db) is 5 years old.

Here's a remaining list of all dependencies where your current version is 3 years old versus the latest version available (or older):

Name	Current Version	Latest Version	Age
ast	2.4.2	2.4.3	4y55d
bootstrap	4.3.1	5.3.5	6y68d
diff-lcs	1.5.0	1.6.2	3y140d
et-orbi	1.2.7	1.4.0	3y203d
faraday-follow_redirects	0.3.0	0.4.0	3y183d
ffi-compiler	1.0.1	1.3.2	7y251d
http-2	0.11.0	1.1.1	4y100d
http-cookie	1.0.5	1.1.0	3y125d
humanize	2.5.1	3.1.0	3y129d
jbuilder	2.11.5	2.14.1	3y235d
job-iteration	1.3.6	1.11.0	3y127d
jquery-ui-rails	6.0.1	8.0.0	8y151d
llhttp-ffi	0.4.0	0.5.1	3y184d
matrix	0.4.2	0.4.3	4y0d
multi_xml	0.6.0	0.7.2	8y147d

Name	Current Version	Latest Version	Age
pdf-core	0.9.0	0.10.0	3y132d
pg	1.2.2	1.6.2	5y238d
prawn	2.4.0	2.5.0	3y64d
puma	3.12.6	7.1.0	5y151d
rails-assets-jquery	3.7.1	2.2.4	unknown
rb-inotify	0.10.1	0.11.1	4y147d
roo	2.8.3	3.0.0	5y241d
rubyzip	2.3.2	3.2.0	4y101d
semantic_range	3.0.0	3.1.0	3y242d
spring	2.1.1	4.4.0	4y348d
spring-watcher-listen	2.0.1	2.1.0	5y358d
ttfunk	1.7.0	1.8.0	3y66d
uglifier	4.2.0	4.2.1	4y363d

Again, your app is so small that I think you should be more on top of these dependency updates than you are. Upgrading these will only get more difficult if/as the app grows.

Recommendation: Measure web request queue time and report to Cloudwatch This is a good fit for Speedshop to implement.

You should be measuring the time between nginx seeing a request and when it first gets processed by Puma. This queue time is the time that will increase if your ECS task count is too low. It should be your primary scaling signal.

Recommendation: Move Puma to cluster mode and run fewer, bigger web tasks You will get better performance out of 1 ECS task with a Puma in cluster mode with 4 workers and 4 CPU than 4 tasks with 1 worker/1 CPU each. The reason is based in queueing theory, but the upshot is that it will allow you to run fewer servers for the same load. Doing this will allow you realize lower request queue times for the same server spend.

My recommendation is to run 4 workers per task, and allocate 4096 CPU to each task. Minimum task count can be 2.

Recommendation: `RAILS_MAX_THREADS` is probably too high (should be 3) The correct setting for `RAILS_MAX_THREADS` is based on your workload. We don't have any data for your app specifically as to how much time is spent waiting on I/O, but, having done extensive testing on this, I can tell you that most Rails apps should be using a setting of 3 (yours is 10).

Having this set too high leads to overly slow response times under load. Having it set too low would lead to higher server costs. 3 is the right compromise for 99% of apps.

Recommendation: WebReportsController#create - Turn on Sentry profiling to understand what this controller is doing that takes ~2 seconds This endpoint is one of the few truly poor-performing endpoints in the entire app.

We need to turn on profiling in Sentry (using Vernier) to understand why some of these traces take so long. A profile will tell us exactly what the stack looks like for these ~2 second requests you see so frequently. It could be anything. A profile will tell us exactly what it is.

Recommendation: Background-ize or parallelize the IO in Api::V1::JournalsController#upload_photo From a latency perspective, this controller has a problem with making three somewhat slow HTTP requests to external providers, serially:

1. A reverse search for geocoding
2. An upload PUT
3. Another reverse search

It may be possible to parallelize these using threads, but an even better approach (which has some benefits for scalability, discussed later) would be to move them into background jobs. This controller is one of the slower ones in the app (slower even than the Journals create endpoint), so the cost/benefit feels worth it to me.

Sidekiq

I tend to focus on queue times, because for Sidekiq, service time (time to actually run the job) is not difficult to manage. Simply look at Sentry, look for jobs which routinely take ~1 minute or more to execute, and if they do, just split them up into many smaller jobs (or use Sidekiq::Iterable). Not too hard!

Queue times are the big thing to manage. When your app suddenly get a bunch of traffic, queue times easily balloon out of control. This is the thing you need visibility on.

Recommendation: Report queue latency to Cloudwatch This is a good issue for Speedshop to implement.

You currently don't have a way to look back historically on queue latency. You can look at the Sidekiq dashboard to understand it moment to moment, but what about when no one's looking? You should report this to Cloudwatch so you can understand how frequently your queues are violating their expected latencies.

Recommendation: Change Sidekiq concurrency to 10 Similar to my comment about `RAILS_MAX_THREADS` in web, we believe the best default for Sidekiq regarding thread count is 10. You're currently running it at only 2, and

it seems like `RAILS_MAX_THREADS` is not being read by the Sidekiq start script you're using. I'd drop the `concurrency` setting from `sidekiq.yml` and use `RAILS_MAX_THREADS` to configure Sidekiq concurrency. I'm also slightly worried about your database pool size is being set to in Sidekiq currently (the setup seems to assume concurrency is configured with `RAILS_MAX_THREADS` and not manually via config files).

Recommendation: Split Sidekiq into two queues and two task types: ASAP and `within_10_minutes` Sidekiq workloads are defined by how quickly they need to be started. Currently, you're operating with just one queue. That's fine. However, I'm sure you've got at least one job in there which cannot wait. That means the SLO of your entire queue is effectively 0, or put another way "the default queue is the ASAP queue".

That's actually a great place to start, but I think you've probably also got work that can wait up to 10 minutes or so to be started. This is useful, because you can autoscale to meet that kind of demand. An ASAP queue needs capacity *already online* to meet its SLO goal, which means it needs high minimum task counts. That's expensive!

As you grow, you will continue to add more queues and SLO buckets (within an hour, within 24 hours, etc).

So, concretely:

- Rename default to ASAP or `within_0_seconds`
- Create a new queue and process type called `within_10_minutes`. Move some jobs to this queue.

Recommendation: 1024 CPU for Sidekiq workers Sidekiq can't use more than 1 CPU core at a time. You currently have 2 cores allocated to these tasks, which is simply wasted.

Errors

Recommendation: Move geocoding to background everywhere, separate endpoints for fetching that data later This was already discussed in a Shortcut issue.

One of the few ways that Huntregs can fail is due to the failure of an external service provider. This may be for any number of reasons - maybe they're just having a bad day and decide to rate limit you for no good reason. You need to be resilient to 3rd-party failure of your two non-AWS external APIs: weather and geocoding.

The best possible path forward for resiliency would be to never call these APIs during a web request. Always do it in a background job. That leads your frontend to the following pattern:

1. POST to a web endpoint.
2. The endpoint returns a URL and says "poll this URL until the result appears there".
3. Client polls the endpoint until the geocoding job is done, and the data they need appears at the URL.

Polling *sounds* old fashioned but in fact it can scale extremely well at this size of app. Each polling request is extremely cheap (10ms) because there's no data there to respond with!

Recommendation: Geocoding in the DB is not necessarily better There's also a Shortcut story around "moving more geocoding calculation from Ruby to the database". I want to disagree with this idea.

Your database is not (easily) horizontally scalable. Your Ruby compute is. Moving these CPU-bound calculations from Ruby to the DB is making them less easy to scale.

They may be faster on Postgres, but I'd want to see a really, really good benchmark result before I started moving in this direction.

Recommendation: Do not process uploads directly in the Rails app, upload directly to S3 Rails and Ruby in general don't really need to be involved in uploading assets to S3. You should use token-based workflows to just let the client upload directly to S3 and cut the Rails app out of it entirely. Big uploads of ~5mb+ pictures and other assets are no fun to scale and deal with. Let Amazon do that for you.

Recommendation: Geocoding and Weather APIs should use circuit breakers When you have a critical dependency on an external API provider like this, you need to use a circuit breaker pattern to decouple yourself from that provider should they go down.

A circuit breaker allows you to make the "error" path when the entire provider goes down *much* faster, which means that if your weather API provider goes down, your site doesn't go down.

As a note, I prefer the Meteo weather API for reliability.

Frontend

One of the reasons I was interested in working with you is because I think your userbase is so interesting: primarily mobile devices, often operating on poor network conditions.

Recommendation: prod.huntregsapp.com should be behind any CDN A CDN is an immensely useful thing when network conditions get bad. By

making the "first hop" on a poor network connection as close to the user as possible, you minimize the effect of that poor connection.

You're on AWS, so Cloudfront is an option, but for a number of reasons, Cloudflare is my preferred vendor. They have a wide and useful product offering, and the CDN product is just better than AWS' in a number of ways (more PoPs, HTTP/3, lots of nice image manipulation features, brotli, etc etc)

Recommendation: HTTP cacheable responses should be HTTP cached In conjunction with the previous recommendation, you have a number of responses that *could* be HTTP cacheable. For example, the main web regulation search flow uses `https://prod.huntregsapp.com/regulations/species?state_id=41`, which is a classic example of an HTTP cacheable response. There's no user data required for this request, I can just run `curl https://prod.huntregsapp.com/regulations/species?state_` and it just works!

HTTP caching would give you some really nice stuff:

1. The CDN will now serve those requests, taking load off of you.
2. The CDN doesn't have to talk to your Rails app, which means all your responses just got at least 100ms faster.

HTTP caching is so useful but so rarely allowed, but you have a great usecase here.

Recommendation: Consider prefetching HTTP cacheable responses If you do find you're able to HTTP cache a lot of your GET requests, and those request are coming from a CDN, you can suddenly get way more aggressive with prefetching stuff on your clients.

For example, on the main regulation search page, you could just prefetch every state! It "Just Works" if you set the headers correctly, and since *you're* not serving that request (your CDN is) you're not meaningfully increasing your own load at all!

It means your users no longer have to wait for anything to load, because the data's already there.

You could, of course, do this in your mobile clients as well. Again, the nice thing is since this is all just HTTP, the semantics of how caching works is already built for you. You just have to mark your requests as HTTP cacheable, the underlying client libraries should do all the rest for you!

Recommendation: Monitor the size of the payloads you return, and compress them Another reason to use CDNs is that they will automatically compress responses. If you're shipping JSON to mobile clients, a brotli-capable CDN could reduce payload sizes by 50% or more.

I can also tell by the stuff you're returning on your API that you might have some especially large responses. I think you should pay attention to this as it's definitely going to be a cause of latency for mobile clients.

I can't be super concrete on this one because I think it will depend on what CDN you choose. Cloudfront and Cloudflare would both have different tools for monitoring response body sizes. I would pick one and keep track of how many response bodies I was sending that were larger than ~100kb.

Recommendation: Drop webfonts (issues: fontawesome is late, just not worth the payoff) You have an issue with your webfonts on prod.huntregsapp.com - fontawesome and Google Fonts are downloading late (they're blocked on something else, I'm not quite sure what).

I don't think webfonts are worth it for Huntregs, outside of the marketing site. If your clients are mostly and mobile and potentially mostly on poor connections, any branding benefit here is outweighed by the ~500ms-1 sec it takes it download a webfont for the first time.

I would browse through https://modernfontstacks.com/ and find something I liked and just use that instead.

If you must keep webfonts, let's host them first-party from huntregsapp.com to improve performance, and I'll investigate the late-download issue.

Recommendation: Firstparty all js you can on https://prod.huntregsapp.com/regulations? You have several third-party CDNs serving javascript on your hunting regulation pages. This isn't necessary and it's much slower than just hosting everything yourself on prod.huntregsapp.com.

Javascript is on the "hot path" of your page load. By putting JS on a 3rd party domain, we now add the time it takes to connect to that 3rd party domain on the "hot path" and slow down our pageload by at least that amount. It's one of those "100 ms problems that adds up".

Recommendation: Consider Turbo Drive The HTML pages served by Basecamp are pretty simple and there's just a few screens. I would consider using Turbo Drive to speed up the page transition and make it feel almost instant when you go between them.

The difference between a full page load and a Turbo Drive pageload will be something like a 10x difference. The other alternative to get that kind of performance boost is a React app, but that would add significant complexity.

Recommendation: Consider 1 big box versus AWS This may be too late, but as I was looking through this, I thought: Huntregs is kind of the poster child for the "new" style of Rails deployment that DHH is pushing with Rails 8:

1. Run on one, big, ~64 core box on a tier-two cloud like Hetzner, OVHCloud, ReliableSite Hosting for ~200/month.
2. Run the database, cache, Redis, Puma and Sidekiq all on the same machine.
3. Optionally, use Sqlite.

That kind of setup can easily handle ~500 requests per second. You're currently doing about 2. You could serve your load for the next ~3 years for $200/month!

The drawback is that all the data stuff is much harder: backup, recovery, snapshotting, is all "taken care of" for you by RDS, and you have to reinvent the wheel. But, something to consider.

Recommendation: Tighten timeouts, both for 3rd parties and yourself When you get circuit breakers in place, you should reduce the read and open timeouts for your connections to third parties. If OpenStreetMap or your weather API doesn't respond in 2 seconds, they're not gonna respond if you wait another 28 seconds.

The default read/open timeouts in Net::HTTP are 30 seconds.

Lax timeouts just lead to worse behavior when third parties go down. I would also consider timing out your own responses on a tight leash: something like 10 seconds. It's easy to do at this stage of an application when it's this small, and much harder to do 3 years from now.

Recommendation: ASG based on request queue times, esp. Sidekiq Your AWS auto-scaling group is currently based on CPU autoscaling. It's better to use request queue times (which, as I've recommended in other parts here, you need to report to Cloudwatch anyway).

The reason is that it's trivial, especially in your app, to construct a workload that 100% utilizes your servers but not your CPU.

Consider what happens if your Weather API gets 2x slower. Let's say every request now takes 1 second, and 0.7 seconds of that is I/O to the weather API, waiting on a response.

In these conditions, your Puma processes will completely full and requests will time out waiting to be picked up, but your CPU usage will be just 30%, and your auto scaler threshold at 40% CPU will never be hit.

Request queue time is the only correct way to scale. We scale based on how long people wait.

Other

Here's some more random things I noticed while looking around.

Recommendation: Your setup is extremely good, but try mise! Your overall monorepo structure and the onboarding process is easily the best I've ever seen for a company of this stage and size, credit to the people who wrote it.

I've really been enjoying `mise` for tool version management. I would encourage you to adopt it officially in the monorepo and enforce/set it up for devs in the automated process you've got.

Recommendation: Turn on profiles in Sentry Speedshop can do this.

Sentry isn't configured to do profiling. It costs a small amount extra (like in the $50/mo range IIRC) but it's incredibly, incredibly useful (mentioned in another recommendation). You should turn it on.

Recommendation: I'd like to read your "performance testing report" I saw the performance testing report on Shortcut, but I don't have access to the repo. I'd like to read it now that I've done my own report!

Recommendation: Marketing site: rebuild it yourself, static site, CDN. The marketing site on Wordpress is kind of hopeless in my opinion. I'd like to hear more about what your requirements are for this site.

The load time for the marketing site, from Tokyo, is ~2-3x worse than the app itself.

The way I see it:

You're a tech company. You can be technical and solve problems in a technical way. You don't need a WYSIWYG for content editing because you've got full time technical staff.

Your prod.huntregsapp.com site is very fast. It would be great if the marketing stuff was built on the same kind of frontend stack, because then it would also be really really fast.

Recommendation: Idea for app db: sqlite shipping weekly patch diffs (ver: 2025v1 v2 v3), split per-state. You asked my opinion on how to deal with syncing ~1GB of regulations to the mobile apps. I'm not an expert on this kind of thing but, my rough idea:

I'm assuming the database itself is Sqlite, or it probably should be.

1. Split the database into one sqlite file for each state. These may include multiple tables. Allow the user to choose which states they want to sync.
2. Ship a patch diff weekly (e.g. version number 2025vWEEK_NUMBER), keep track on the client which patch diff they most recently ran and what the most recent diff is that's available.

This is more or less how most video game companies do it. The patch diff files themselves can be HTTP cached as static binary data, which would also help download times.