Speedshop Tune: Easol

@nateberkopec, with review by @yuki24

Prepared with care

@nateberkopec, with review by @yuki24

2024-10-31

Contents

Hello Easol!

Thanks for having me take a look at your application. I've identified several areas for investment in performance. Some of the things I've discussed in this report are simple configuration changes. Others are more long-term projects and skills that you can improve on over the next six months.

At the top level of this document, you'll find our desired Outcomes. These are my metric-driven goals for your performance improvements over the next six months. Underneath each Outcome are specific Recommendations; our methods of achieving the Outcome. Each Recommendation has an associated cost and benefit. I rate them, subjectively, on a 5 point scale.

At the end, you'll find the Outcomes and Recommendations without commentary. This is a quick reference to help you turn this document into action and assist during planning your upcoming work.

I hope you enjoy this report. You'll find it a useful guide for the next 6 months of performance work on your application.

[illegible]

Nate Berkopec, owner of The Speedshop

Outcome: Improve your observability

I enjoyed taking a look at some of the dashboards you've created already. Some of them are quite good at solving a specific problem (like that timeout dashboard). However, I think there's a big opportunity to build something which is useful not just for Vanguard, but for the entire organization.

Performance is engineering: the science of meeting **requirements** while staying within **constraints**. Our measurements must reflect each of the three parts of engineering:

1. **Requirements:** How fast does this have to be? How many req/sec does it need to serve, at what level of service? In SRE terms, we can call this a Service Level Obligation.
2. **Results:** For each requirement, what is the actual state of the app? If we say it needs to respond within 100 milliseconds, what are we actually doing over time? In SRE terms, we call this a Service Level Indicator.
3. **Constraints:** In the modern era of the cloud, only two constraints: cost and time. All projects have limits on the time and the money we can devote to performance.

Unlike feature work, one can master performance work with a numbers-driven Taylorist approach. What gets measured gets managed. At almost every new client we've had, not enough is being measured. Often, they're measuring the wrong things.

Most teams do a drunkard's search. They base their performance work on the numbers that their APM shows by default.

We reject allowing our APM vendor to decide our "important numbers" on our behalf. Instead, we proceed from first principles. What kinds of numbers matter to performance?

There are three domains of performance that every app will care about, although to different degrees:

1. **Experience**: The latency that customers feel as they use the site.
2. **Scalability**: Resource utilization and response to changes in load.
3. **Reliability**: Global uptime and health of underlying resources (databases).

Our metrics must cover each of these three domains. For each unique business, some domains are more important than others. But we must measure all of them.

Recommendation: Implement the Speedshop Standard Dashboard

That's how Speedshop arrived at the Standard Dashboard. It is a set of 25 charts. They communicate Requirements, Results, and Constraints in the three domains: Experience, Scalability, and Reliability.

This is the dashboard:

- Experience
  - Page load time (all loads)
    - Page load time (initial/cold load)
    - Page load time (hot SPA route changes)
  - Time for interactions (i.e., time spent waiting on DOM/network for clicks that don't change the URL)
  - Time to execute customer-blocking background jobs. For any background job where a customer is actively waiting on the result and is blocked until that job completes (password reset email), tracks total time from enqueued_at until completion.
  - % of responses which took longer than 500ms, organized by controller action.
- Scalability
  - Web utilization
    - Total Puma process count
    - Concurrent request load (average req/sec * sec/req)
    - Process count / load
  - HPA/scaler status (web and workers)

        - current, min, max
    - Web request queue timing (p75,p95,pmax)
    - Worker latency
        - For each queue, show queue latency (and SLA for that particular queue)
- Reliability
    - Database, cache DBs, and Redis DBs
        - CPU (load and utilization)
        - IOPs (if limited)
        - Read/write latency
        - Error rates
        - Hit-rate (if cache)
    - Error rates
        - Web, worker
    - www.*.com uptime

We've already implemented the beginning of this dashboard for you here. There is some work to be done still to "fill in" the remaining parts:

1. **RUM.** See recommendation: "Install RUM"
2. **Creating a customer-blocking job queue.** See the recommendation: "Create a within_0_seconds queue"
3. **Create a reminder for the "controller action SLO".** The "% of responses which took longer than 500ms, organized by controller action" is an important chart. However, it's not really appropriate to "alert" on. No one needs to be woken up at 2am to deal with this. It's more of a long-term inventory or backlog. Add this to your "process" to review at regular intervals and scope/assign work to fix.

Cost: 2, Benefit: 4: A good picture of what's happening is the foundation of future work. The remaining work to "fill out" the dashboard is nearly done.

Recommendation: Terraform your Datadog monitors and SLOs

We talked about your past forays with Infrastructure as Code, and it seems like it didn't work out. That's fine. I'm not asking your to IaC your entire infrastructure with this recommendation, but I do think you'd find it useful to IaC your Datadog setup.

Almost every team I work with eventually goes this route. It just becomes *far* easier to manage X number of SLOs for your queues when adding a queue to the setup is as simple as adding one line to a terraform file and hitting "merge", with everything auto-applied by a Github Action. It also adds a layer of accountability, history and "why did we change that?" support.

I'd start by Terraforming the performance dashboards, monitors and SLOs.

Cost: 2, Benefit: 2. Mostly the benefit is for the future when any of this needs to be changed, but it's a low-effort project.

Recommendation: Install RUM, ditch WebVitalsController

It's tempting to think of ourselves as "Rails application developers", but what we really are is "browser experience developers". Unless your job is to ship a JSON API to non-human users (e.g. Stripe), then you're trying to make human users happy with an experience delivered by a web browser. Shipping the browser some functional HTML or JSON is just Step 1 in Steps 1-10 of "make the web page do the thing".

Without performance telemetry coming from the browser itself, we're not measuring steps 2-10. We really have no idea what the user is actually experiencing. This becomes particularly more important when SPAs start to get involved (which they're not here yet, but eventually they will be).

You're already on Datadog, and using it quite well. I recommend shelling out $500 a month on Datadog RUM with Session Replay, sample it down until you're spending about that much. I've used Datadog RUM extensively, it's a good product. It'll integrate well with your existing dashboards, SLOs, and overall approach.

You've already ditched WebVitalsController as well, which is good. It was contributing a non-trivial (though still small) amount of load. But we can't just ignore Web Vitals altogether! It's just better to use someone else's prebuilt product rather than "inventing it here".

Cost: 1, Benefit: 3. A browser-first performance mindset is really transformational. It puts everything in perspective: does 100ms faster time-to-first-byte really matter when my LCP is 5 seconds?

Recommendation: Post Launch Events to Datadog as an Event

You already have an ICS file somewhere with launch events.

It feels like a 1-to-4 hour project to use some AI-codegen to create a Github Action that will take that ICS file, and upsert all the events in the calendar to Datadog as Events.

Once they're in Datadog as Events, you can overlay them onto pretty much any chart in the entire app.

So much of your current performance problem set (and really, so much of the *business*) revolves around these launch events, that I think it would be supremely useful to have this embedded in Datadog as a "first class citizen".

Cost: 1, Benefit: 1: You're a launch-driven company, this helps you act like it.

Recommendation: (COMPLETED) Send Heroku Postgres, Redis logs to Datadog

We worked on finishing this one together, so I don't have to expand too much on it as it's 100% done. I got the initial version up but y'all took it over the line and made it much nicer - thanks!

This one was important because Postgres/Redis outages are going to be a major cause/failpoint for future launches. Having that information in Datadog is critical.

Cost: N/A, Benefit: 3: All done.

Recommendation: Ditch Datadog Profiling

You've currently got the Rails app set up to do continuous profiling in Datadog, which is here.

I find Datadog's continuous profiler to be completely useless for Rails.

The problem is that a continuous profiler doesn't have any concept of a "request" starting or ending. It just profiles your code for, say, an hour, and then makes aggregations over what happened for the previous hour. The problem is that web applications (and your background job processing) can be doing any one of a thousand different high-level transactions (jobs or controller actions) and each of those works completely differently.

Because of this, continuous profiling can really only catch *blindingly* obvious issues which involve added overhead to *every* transaction. This kind of problem is incredibly rare, and it's also easy to catch with the trace tooling.

The thing is, you really only need Datadog's `APM` product ($36/host/month on demand). This continuous profiling thing comes from the `APM Enterprise` product, which is $12/host/month more.

You average about 4-5 dynos, so that's 48-60 dollars per month you didn't need to spend. I'd just ditch it. That number is low, but when you start running 100 dynos to deal with launch traffic, it might add up (how does Datadog deal with a host that runs only for 5 minutes a month? IDK).

Cost: 0, Benefit: 1: Easy enough of a switch to flick in a Datadog admin dashboard somewhere.

Recommendation: (COMPLETED) Add Datadog database monitoring

While I find the continuous profiling product mostly useless, the database monitoring product is great! I'm glad we got this set up, I actually had no idea you could do it on Heroku.

I added 'wait by app' to the dashboard, which I think is a supremely useful way for separating "db load" over "sidekiq vs web" dimensions.

Cost: N/A, Benefit: 2: It's really the best way for sussing out where database load is coming from.

Recommendation: Use rack-mini-profiler in all environments

Unfortunately you got bit by Redis when deploying this. I'm not sure what the blocker is right now to re-applying this.

One of your biggest issues is working out how to more efficiently load data in PagesController. When we initially got started on this engagement, I remember Kyle explaining the main source of your problems to me as "our Liquid stuff", but that's a slight misnomer. The Liquid itself is probably fine, it's the way you're loading data that goes into that Liquid template that's killing you.

In my experience, every time we transform a page from making 100+ SQL queries into one that makes 10-15 on a cold, uncached load, we completely fix all the latency issues and it's a 100ms p50 page at worst. I think the same thing can happen here.

But, that's probably going to require a massive revolution in how you're loading and passing data into Liquid templates. ActiveRecord's lazy-loading right now is killing you.

Where to start? We need stacktraces for individual queries to figure out where they're being triggered. The easiest way to get that is RMP. That's why it's so important to get installed, and it's needed in *all* environments because production is currently the only place to get realistic data, and production is the only place to reproduce specific customer issues.

Cost: ???, Benefit: 4: This is the way forward to figuring out a better data-loading pathway for your Liquid-powered stuff.

Outcome: Serve higher req/sec during launch events

With the right observability in place, we can move on to actual value-generating work. As you've communicated to me, the most important of that work is keeping the site online and performant during launch events.

In the recent past, launch events have been a 2-3x multiple on current site load, although this is worsened by the fact that launch events tend to dump load onto your poorest-performing controllers (PageController).

Throughout the last month, we've worked a lot on the causes of outages during launches. This section summarizes these findings.

Recommendation: Increase minimum dyno counts in advance of launch events using Judoscale's API

I feel like my recommendations to you weirdly include a lot of mini-services. I don't normally include this. It could be that AI-codegen has influenced me to see these sorts of things as much easier and much lower cost than they were even a year ago to develop, deploy and maintain.

Here's another:

1. At the start of each day, check the Launch Calendar ics file.
2. For each launch event, use the new Judoscale scheduler API to schedule a web dyno count increase for 15 minutes before until 1 hour after the event time.
3. (Optional) Schedule a temporary pause on 1 hour or longer queues for 30 minutes.

This could be deployed as a Github Action, no need to add anything on Heroku either.

It's simply not possible for you to autoscale adequately to meet the sudden load generated by a launch dump. Going from X req/sec to 3*X req/sec within a 5 second period will *inevitably* result in service degradation until new dynos can be brought online, which will take anywhere from 2 to 10 minutes (a number that will only grow over time as the app grows in complexity).

Instead, you have to provision capacity in advance.

The cheapest way to do this would be to overscale for brief periods around known events. Given Heroku and the Judoscale API, you have a tremendously easy way to do that.

Cost: 2, Benefit: 3 Easy to do, with a nice payoff.

Recommendation: Consider permanent over-scaling on web by setting the next day's minimum to be the same as the previous week's maximum.

The same Github Action in the previous recommendation could also do the following:

1. Look at the last calendar week's maximum web dyno count. This represents the maximum that we scaled to during all launch events.
2. Set that number (or some multiple of it) as the *minimum* dyno count.

The intent of this is to cover you for all the "unscheduled" launch events for the coming week. Again, it is not physically possible to bring on enough capacity quickly enough to catch up with these events.

Shopify used to have so much extra capacity installed that they ran at 5% CPU utilization. You won't have to be that extreme, but some degree of over-provisioning is probably required.

Cost: 1, Benefit: 2: It's gonna cost more, but may be the only way to achieve a 4 9's SLO for request queue time.

Recommendation: Move the rack-attack memcache server to a separate machine

You currently run a single Memcachier addon with 5GB of storage. The problem here is that it's used as cache *and* as a store for rack-attack.

rack-attack has completely different access patterns to a cache. rack-attack is creating new keys for every IP address, which is a super high cardinality check, and may involve lots of bots making 1 request and never coming back. It's going to use loads of memory, *particularly during launch events*, and it's not even all that effective (WAF-level protection at, say, Cloudflare is far more useful).

You should split rack-attack into it's own Memcache instance, and wargame a failure of that database. A sudden influx of traffic shouldn't cause rack-attack to insert a bunch of new keys and push out potentially *actually useful* cache keys generated by the Rails app.

Cost: 1, Benefit: 2: A low-likelihood footgun, perhaps, but definitely the last thing you want failing during a launch.

Recommendation: (COMPLETED) Change WEB_CONCURRENCY to 8

I don't have the number this was set to previously, but I recommended you change it and you did.

This is particularly important for your setup. You need an accurate auto-scaling signal.

Let's say you ran way over-subscribed, like at WEB_CONCURRENCY=100. During a launch dump, you wouldn't see request queueing increase that much, but service latency would go through the roof.

Unfortunately, that means your autoscaling would never trigger!

Running without "oversubscription" of processes-to-cores is the best way to go for reliability.

Cost: N/A, Benefit: 1: Another low-likelihood footgun, easily fixed.

Recommendation: Create automated warnings on timeouts, SLO violations for web requests, and build a postmortem-driven incident culture.

Performance work is engineering. We set the requirements, and we either meet them, or we don't.

There are two ways that requirements which haven't been managed will show up:

1. Complaints. People will complain about the site going down or being too slow.
2. Fires. Any time something comes up in the #fire channel, it's a sign that a requirement wasn't managed adequately.

One of my favorite ideas from the non-software world is the idea of an **andon cord**. On Toyota production floors, there's a cord that runs along past every workstation. Any worker, at any time, can pull this cord to indicate "I have a problem". What happens next though is pretty special.

When someone pulls the andon cord, **the entire line stops**. The entire factory halts. Then, everyone (in theory. In practice, it's the senior plant managers) comes by to help and figure out what went wrong.

There are a number of important ideas here:

1. Anyone, from the lowest junior the highest staff engineer, should feel like they can direct the entire company's attention to any problem which prevents "the line" from working (in our case: breaks any performance SLO).
2. If there is a problem, we stop the entire company to fix it. This may feel inefficient at the time (lots of people standing around), but it pays off in the long run, because we will solve the problem permanently and "the line" will continue to run smoothly in the future.

You could think about this a radical approach to certain kinds of technical debt: a zero-tolerance policy for accumulation.

Technical debt is often quite a powerful lever. It can allow us to make certain business outcomes possible on short timelines. But in the case of a known requirement, accumulating "debt" (i.e. allowing the normalization of deviance from that requirement) can be incredibly toxic.

What does this all look like in Easol's context?

There is currently **not enough organizational attention** devoted to #fire and violations of existing SLOs. I'm not seeing a sufficient level of time and energy spent on these issues to be in alignment with your goals on serving larger and larger launches.

1. **Automated monitors on all "failure" conditions.** I think we're getting there, but automated monitors which can drive the creation of incidents, even for short outages (between 1 and 5 minutes) is important for Easol.
2. **A post-mortem-driven culture** around launch events. Incidents in #fire lack serious follow-up. Every incident with customer impact deserves a post-mortem meeting.
3. **SLOs everywhere**. Monitors are the "get up at 3am and fix it now" component, SLOs are how we hold ourselves accountable over the long-term. Anything in the performance and reliability domain that we care about in the long-term deserves an accompanying SLO.

Cost: ???, Benefit: 3 Hard to put a number on a cultural shift.

Recommendation: Fix /cart/stripe_card_payment?order_id double Stripe hit

This is a low-likelihood footgun.

All views which render `checkouts/_summary.html.erb` (by my count: StripeCardPaymentsController#update, PaymentMethods#show, StripeSetupIntents#update, Checkouts#wizard) are potentially making two of the same request in a row to Stripe.

`_stripe_payment_intent_confirmation.html.erb` causes a hit to the `payment_intents` API, and then renders `_stripe_payment_method.html.erb` and immediately hits the same API again. This second call should be unnecessary.

In some contexts, both calls may be unnecessary as you're often `POST`ing the Payment Intent right before calling `GET` twice here.

Stripe is probably the most important 3rd-party API on the whole site, so keeping your calls neat, tidy and low in number is an important way to make sure you're resilient.

Cost: 2, Benefit: 1 The main benefit is in reducing time to render by ~200ms, but it also feels important to keep Stripe call count as low as possible.

Recommendation: (COMPLETED) Create a launch-focused dashboard

I went ahead and created this for you along the way this month.

The intended use of the dashboard is:

- For the last 2-3 days, scan the requests per second graph and look for spikes.
- For each spike, zoom in and look at whether or not request queue time and response time spiked.
- If it did spike, use the other graphs to quickly diagnose a cause.

You could do this on a regular cadence and call for a postmortem if any of your request queue/response time thresholds were violated.

As of writing this (Oct 30), I saw only one post-mortemable incident. A sudden dump of traffic during Oct 29, 10:13 am – Oct 29, 10:18 am UTC for coffee hospitality expo resulted in the p99 request queue time briefly shooting up, simply due to a large number of requests being dropped at once on the app. This issue would be addressed by my "scale dyno minimums to last week maximums" recommendation, but it would also be good for you to have your own postmortem of such events and own the solution yourselves.

Cost: N/A, Benefit: 2: This is a critical step in building a "safety culture" around launches.

Recommendation: Reduce resources used for each launch

This was something I covered a number of times on calls, so I don't think I need to belabor the point here.

Scaling is just a numbers game: serve more with a given level of resources. However, we can also keep the request rate the same and try to serve it with fewer resources. It's the same thing: serve more with less, we're just changing which side is the independent variable.

You have a limited set of resources which must change horizontally or vertically when traffic changes:

1. Dyno counts.
2. DB sizes (Postgres, Memcache, Redis)
3. That's it.

I would be trying to run each of these at lower and lower sizes or numbers over time. Doing so means that when a real event comes along (Vibee), I don't need to provision nearly as many resources to serve that event.

Cost: 2, Benefit: 3

Recommendation: Prior to Vibee launch, thug it out and crank that DB plan as high as it will go

This wasn't really my idea, more of something either Tom or Theo brought up (can't remember, sorry lads!) during a call, but it's just to say that databases can be scaled up and down on a ~24 hour cadence pretty easily, which is fine to do for a massive launch event that 100xs your current scale, such as Vibee.

Cost: 1, Benefit: 3. I still believe that your main bottlenecks are resource-based, so this goes a long way to ensuring that Vibee goes well.

Recommendation: Prior to Vibee launch, drop a load-test nuke on your payment flow and try to smoke out any other resource contention

On the same call, we talked about the role of load testing in preparation for events like this.

What you **cannot do** is say:

1. We need to serve 500 requests/second.
2. We come up with a load test that does that, run it, and it goes fine, therefore…
3. … we'll be fine on launch day.

That's not how it works, primarily due to variations in request arrival rate. When you do a load test, usually what happens is you come up with a specific number of concurrent users and tune that up or down until you get a requests/second number that feels "OK". The problem is that this "closed" traffic model is completely irrelevant to how traffic actually works.

K6's docs on this distinction are quite good.

A closed traffic model is one where the response time influences how much traffic you throw at the app. So, if response time goes up, we send fewer requests/sec. Wow, I sure wish actual web traffic worked that way. You can maybe see the problem already here: if our app slows down, it serves fewer requests/sec (the load tester is sending a new request AFTER each previous request has been served).

An open traffic model on the other hand sends a certain amount of request/sec regardless of the response time (or even if the responses were successful). This is NOT how most load testing tools work by default (but you can do it with k6, your previous choice).

Your previous load tests used the `constant-vus'` executor in k6. This is a closed model. It does not accurately reflect how queue-it works.

With this executor, if your response times increase by 2x, the request rate will decrease by 2x. This is not reality.

The `'constant-arrival-rate'` executor reflects how queue-it works: a constant rate of requests dumped on an endpoint, which does not change regardless of how fast or slow the service responds.

Now, even that is still not the most accurate thing in the world. Real world traffic is not a constant arrival rate, it's constantly shifting and changing, and

400 req/sec one minute becomes 800 req/sec the next. Request arrival rates are *statistically distributed*, and this can make a huge difference in our service level.

Instead, what you *can* do is this:

1. Use a ramping arrival rate.
2. Note the service quality metric during the test. This is request queue time. When request queue time exceeds an acceptable level, the test has failed.
3. At the point the test failed, figure out why. Which component died? The database? Which one? How?

You can't translate the number at which it failed back into a real-world number, but you *can* accurately say that for this traffic pattern, the first component to fail will be the same as in the load test. Fix that bottleneck and then try again to find a new one.

Cost: 3, Benefit: 3 I find accurate load testing very hard to set up and maintain. You do have some previous experience though, and I like k6.

Recommendation: (COMPLETED) Change DB vendors

You're already down the pipe on this one. I hope CrunchyData works out well for you. They've got a good team and my Slack seems to have good experiences with them.

Cost: N/A, Benefit: 3

Recommendation: Do not run high-SLO queues at maximum concurrency all the time, use 2x dynos rather than L.

This is one I've brought up in the chat a few times now.

You are currently running all of your SLO queues on a single worker using a "tiered" setup.

That is, you have a single default process type, which can pull from any queue, in order (first the lowest SLO queue, then the next highest, and so on). You are currently running this worker with 8 processes (via `sidekiqswarm`) and 4 threads per process, for a total concurrency of up to 32. That's a substantial amount of firepower!

The problem is that this firepower of 32 concurrent jobs is often misplaced with 20 minute SLOs or larger. You've had a number of times where long-SLO queues (20 minutes or more) were running at high concurrency during a launch, placing load on the DB at a time where that load must be kept low.

The great thing about high-SLO queues is that you *don't need* ultra high concurrency. It doesn't take a lot of firepower to make sure a few jobs on the 20 minute queue get executed within the next 20 minutes.

It also means that you can easily turn off high-SLO queues during launches (i.e. using the Judoscale schedule API to scale them to 0 for 15-20 minutes).

So, I recommend again that you **remove the worker process type** and just let each process type consume a single queue.

Lower concurrency provides a level of safety, particularly regarding database resources, and there's no "prize" for executing 1 hour SLO jobs within 2 seconds.

Additionally, I don't think sidekiqswarm is really helping you much. You're paying $500/month for 8 Sidekiq processes, or $62/process. You could just run each process on a 2x dyno for $50/month. This would have the added benefit of giving you more granularity to scale up or down (because now anywhere between 1 to 8 processes is possible, where before you could have 0 or 8). This is particularly nice for the high-SLO queues, like within_20_minutes and higher. With jemalloc, you probably now no longer have to worry about memory limits (this would be the one advantage of Swarm).

Cost: 1, Benefit: 2 More safety for a little config-swapping.

Recommendation: Chart out your napkin math

We more or less did this on a call but I'll write it down to re-iterate the Vibee math:

1. You have 30,000 tickets to sell.
2. At a conversion rate of 10%, you'll need to serve 300,000 visitors to sell these tickets.
3. Let's say you want to sell them all within an hour (corollary: this means the average time waiting in queue post-launch-time is 30 minutes). That's 5,000 requests per minute, or 83 requests per second.

83 requests per second for 1 hour is a nice, predictable load. We can use this to make a few decisions:

1. **Increase all queue-it rates** from now until the launch to 83 requests/sec, so we can test the actual load we'll experience on vibee.
2. **Investigate what happens in production** when we start to hit around 83 requests per second.

For example, on `Oct 25, 8:55 am - Oct 25, 9:16 am` UTC, you had what looks like maybe a couple of launches at the same time, and went over ~20 req/sec for a 20 minute period on the pages controller. So, Vibee will be approximately 4x this load.

During this mini-launch, we saw:

1. DB CPU load hit 4.
2. Database latency spiked when a bunch of Sidekiq jobs were enqueued during the launch (I think PrunableWorker on the 5 minute queue).
3. IOPs hit 100% for about a 5 minute period.

4. Request queue times looked good, but process utilization approached ~50%.

Take all these numbers, multiply by 4, and you get an idea of what the Vibee launch could look like. That means your DB plan is probably still too small also by a factor of 2-4x, and you could need about 2x the amount of dynos (aka 16) if the page performance/response time is the same as this launch.

Cost: N/A, Benefit: 3 I prefer this kind of whiteboarding to load testing.

Recommendation: Reduce query counts and p50 on launch drop pages

The biggest load for you during launches isn't what happens after the initial drop into the Easol system, instead it's actually that first page.

1. These pages are often very expensive, at 500-1000 milliseconds for p50.
2. Maybe 50% of visitors bounce after this initial visit.
3. They land on this page at a constant rate, and then after that might make 1-2 requests per minute as they navigate the checkout flow.

These factors combine to make this the most important bottleneck in the system.

These pages are in dire need of a reduction in query count. 500+ queries for a page is a significant load generator, which is what creates the primary pathway for how these pages generate IOPS/DB CPU load.

Cost: 4, Benefit: 5: Fixing how DB information is provided to Liquid in a way that minimizes SQL queries is the single most important thing you could do to improve scalability of this app.

Recommendation: Delete RUBY_ GC tuning variables, MALLOC_ARENA_MAX=2.

When I was poking around the config vars, I noticed these. You probably forget you added them. Now that you have jemalloc, they are unnecessary footguns.

MALLOC_ARENA_MAX can slow down the app by 5-10%, but it also is a no-op with jemalloc, so it no longer does anything.

RUBY_ GC tuning variables are similarly not without danger. They increase GC frequency, which increases CPU use. Again, with jemalloc, that's more than enough to take care of your memory usage problem. Ditch the GC tuning and enjoy 1-2% extra free speed.

Cost: 0, Benefit: 1

Outcome: Reduce p75 response times

As of this writing (end of October 2024), there are a handful of actions for which p75 latency is noticeably high.

Another way to think about this: 25% or more of visitors to this action are getting a sub-par experience that they would *feel* a difference with, if we optimized it.

More or less every action you have which is not a low-level system action (like ActiveStorage or a 404) has a p75 of over 500ms. A handful have p75s of over 1 second.

This represents a good opportunity for improvement.

The combination of all of these recommendations creates a new "workflow" which did not exist before:

1. Something goes wrong in production, as is picked up by a monitor, SLO or by a "backlog" chart like the ">5% of responses over 500ms" table.
2. A human being translates this SLO/monitor/chart violation and turns it into a story/ticket to be worked on.
3. Using a new suite of tooling, including rack-mini-profiler, prosopite, and the production db copy, someone repros the issue and makes a fix.
4. A PR is posted with before/after logs or metrics (i.e. "this did 100 SQL queries before, now it does 10")
5. We deploy the fix, and look at the monitor or chart to see if the issue is resolved.

Recommendation: (COMPLETED) Index all _id columns, enforce this via Github Action

All done. This is just such an easy mistake to make, and an easy one to basically "outlaw" so that it never happens again.

Cost: N/A, Benefit: 2

Recommendation: Lay the groundwork for code and experience ownership

Easol's at that stage of growth where "the commons" starts to become a thing. You've got just enough code and just enough developers where the following story starts to play out, over and over:

1. A developer has to head into a section of the codebase they're unfamiliar with and never seen before. This may happen quite often, as they're expected to navigate ~185000 lines of application code.

2. They make a change that solves *their* problem, but causes problems for other people. They're usually not aware of this, because they can't keep all that context in their head.
3. Rinse and repeat. It's particularly bad when the area being modified was written by someone who left the company, or isn't frequently touched by any one single team.

This "tragedy of the commons" manifests in the performance domain too.

It's the right time to start thinking about modularity, either via `packwerk` or some other means. No one human being can keep more than 10k-30k lines of Rails application in their personal "context window". You need to structure the app so they don't have to.

Code and experience ownership also means that every line and every experienced is *owned* by someone or some specific team. No one can say "not my problem" or "I didn't know that was our problem to fix", because everything's got an owner. As frontends get more complex, it's also important that every *URL* have an owner, as a single URL may make requests to varying APIS owned by many different teams.

I saw this implemented at Gusto via packwerk, but there are many ways to slice this. Services are also valid. I'm not an architecture expert. What I do know is that we cannot optimize what we do not understand, and you're quickly approaching the size of app and team where 1 team or person cannot understand the entire app.

Cost: 4, Benefit: 4

Recommendation: Have a seed/db pull process based on production data, particularly for pages

One of the most important things you have to do to squash an N+1 is to reproduce it. RMP in production helps with that, but it doesn't help when you want to have a tight feedback loop to fix the N+1: how do you know if it's fixed? You run the page request, check your logs, and make sure the query is gone.

This is all a million times easier if you have prod or prod-like data in your local database.

Your challenges are not based on user data. You don't have problems related to any of your tables which could contain PII. So, your seed process doesn't have to worry about this.

Maybe what you need is… another microservice!

It should:

1. pg_dump the main database.
2. Remove all tables except the ones which do not contain any PII, and let the foreign key restraints lapse.

3. It should create a SQL dump which allows loading this locally, and update this dump daily or weekly.

The devil is in the details here, but I feel like this could work. Other, more complicated ways, include RepliByte.

Cost: 3, Benefit: 4 This will be a key part of an N+1 squashing workflow.

Recommendation: If any controller has >5% of it's responses take longer than 500ms, create a ticket.

We now have a widget which sorts your top 50 controllers by traffic by the % of the time they respond in 500ms.

Picking from the top 50 controllers by traffic ensures that we're only looking at experiences which happen frequently. Focusing on a 500ms threshold means that any improvement (e.g., 500ms to 250ms) would be noticeable for the user (unlike, say, improving p50 from 100ms to 50ms).

This chart is more of a "backlog" than an "alert". Nothing happening here is "critical" or "wake someone up at 5am", but it is a drag on having a good experience on Easol sites.

I recommend integrating this chart into your development workflow somehow. This is probably the time and place for Vanguard to be involved - picking work off of this queue and either doing it themselves or assigning/working with other teams.

Cost: 0, Benefit: 2: A low-cost process change.

Recommendation: Ditch bullet, move to a fail-hard Prosopite-driven workflow

After looking at hundreds of Rails apps, I have a strange observation: apps with `bullet` installed are generally slower than the ones without it.

I don't think this is because bullet makes apps slower. I think it's a confounding variable: teams which install bullet are more likely to be in a performance hole in the first place, and they don't know how to get out of it, so they install a tool that they *think* provides full coverage of *all* the N+1 errors in their app, but it doesn't.

Bullet, as a tool, has three major issues:

1. It's easily ignored. In my experience, developers do not read their logs.
2. The underlying N+1 detection is not that great. A good degree of false positives *and* negatives.
3. It has enough false negatives that a significant "inventory" of N+1 issues are left over even if you make all of bullet's suggestions.

This is because `bullet` tries to tie the *problem* (repeated queries) to a particular solution (preload/eager_load). But not all N+1s are fixable this way. Consider:

```ruby
def some_method
  user.posts.where(created_at: Date.now..)
end
```

This method, if called in a loop, will N+1. And you cannot `preload` or `eager_load` to make that go away. A larger refactor is needed. So, bullet doesn't say anything.

`prosopite` is a different style of N+1 tool, with a different philosophy and workflow.

I prefer it over bullet because:

1. It detects a few additional kinds of N+1 (like the one I gave above)
2. It has almost 0 false positives
3. It can be easily integrated into tests

Prosopite makes regression testing for N+1s easy. You configure it to `raise` exceptions in tests if N+1s occur (you can do this easily for only a particular test or example group in RSpec), which means you can require that all N+1 fixes also include a test. That's incredibly powerful as a team workflow tool.

It will require some work from Vanguard to set up properly - mostly around ignoring certain queries/stacks. The "user story" is "a developer can add nplusone_fails: true to any rspec example/spec and it will fail on n+1".

Cost: 1, Benefit: 2

Outcome: Reduce admin p99 response times and timeouts

p99 response times are usually quite distinct from p75 and p50. The reason why any particular response ends up at the 99th percentile is usually quite idiosyncratic (I mean, duh, it didn't happen 99% of the time) so there's usually a lot of "weird stuff" going on at this part of the curve.

Why focus on this in particular? You've got a number of requests that are timing out at the ~30 second mark, which is a particularly bad customer experience.

This really only occurs on your admin pages. This is important because if these pages time out (probably only for particular customers), people can't do their job (literally, because the page explodes).

Recommendation: Don't lock rows for update on Company::Bookings::EditsController#show

This is the most popular timeout on the whole site. I'm confused as to this query, which frequently blocks for 20 seconds or more waiting for the lock:

```
SELECT finance_bookings . id, ... FROM finance_bookings WHERE finance_bookings . id = ? LIMI
```

I can see how the bookings table would be difficult to lock. So why are we trying to do it on a `show` action, which doesn't appear to update these rows at all?

Cost: 1, Benefit: 1

Recommendation: Burn down admin pages that are timing out

I've added two extra charts to the Performance Dashboard, outside our usual set: "% of responses over 20 seconds", by 'url path group' and by domain. Unsurprisingly, your biggest customers pop to the top of the domain list: Swingers, Sherlock, Rise, Afronation. However, admin responses taking >20 seconds for these customers is probably a source of possible churn. Eventually, they're gonna get tired of the site being slow enough to check Twitter while it loads (or just straight out failing most of the time).

Much like the % of responses > 500ms chart, I suggest that these charts be used to create work around the admin panels. Most of the problems on these pages are related to absolutely colossal N+1 queries or the lock issue above.

When viewing their products variants timeline, a recent Swingers response saw over 8000 total SQL queries. Ouch.

Cost: 3, Benefit: 3. It's hard to judge the benefit of these because the people affected are a limited set of high-ARR admins.

Outcome: Make background job processing more robust

We've focused mostly so far on the quality of service of your web requests. What about background jobs?

Recommendation: Throw a "chaos monkey" into your queue scaling using Judoscale's API

Netflix's "chaos monkey" project is pretty famous now.

One thing that can happen with SLO-based queues is that the *actual* performance of them becomes much higher than their *labeled* performance. This is actually *particularly* bad with your current setup where `worker` can pull from any queue.

Over the last week, the *maximum* latency on the 1 hour queue was about 6 minutes. The maximum latency on the 6 hour queue was 20 minutes.

What happens here is that people mis-categorize jobs into the wrong SLO and then end up finding out they can't accept that SLO at the worst possible time: an incident, when SLO queues get paused and everyone is already frazzled and in #fire dealing with something else.

It would be better if SLO queues frequently *did* take about 75-90% of the actual time of the SLO to process. That way, mis-categorized jobs (oops, that should really be in within 5 minutes, not within 6 hours) are noticed early and quickly.

A "chaos monkey" service would:

1. Look at the launch calendar to check if there are any launches today. If there are, shut down and wait until tomorrow (no chaos on launch days!).
2. Schedule random pauses on high-SLO queues (probably 20 minutes, 1 hour, and 6 hours) of appropriate length. For example, it might schedule a 3 hour pause on the 6 hour queue (dyno max: 0). I would probably only schedule this during work hours, so that people don't have to get paged out of work to respond to anything going wrong.

This wouldn't take long at all to implement. Yet another (probably GitHub Action powered) service I'm recommending :)

Cost: 1, Benefit: 2

Recommendation: (COMPLETED) Add SLOs to Datadog for all Queues

All done now! I like having the aggregated monitor as well, I'd never considered that before. Nice.

Cost: N/A, Benefit: 2

Recommendation: Create a within_0_seconds queue for jobs which a customer is actively waiting upon their completion

You already have the within_1_minute queue. Why do you need another, tighter queue?

For any job, when assigning them to an SLO queue, I think: "would it be OK if this job executed at 90% of the SLO 100% of the time?". So, for a password reset email, would it be OK if that job executed 50 seconds after enqueueing all of the time?

I would say no. A password reset email represents a blocked customer, someone actively waiting to do something, in exactly the same way a web request does.

Every app has at least one job like this. These background jobs are essentially web requests that we process asynchronously rather than sync. Each job is tied to a customer whose eyeballs are waiting for the completion.

We don't want a lot of jobs in this queue, because this queue cannot be autoscaled at low volumes. In order to ensure this level of service, we simply have to provision a lot of firepower in advance. If we have to autoscale, the SLO has usually been violated.

In an exception to an earlier recommendation, I would be ok if you ran a single process type which pulled from `within_0_seconds` and `within_1_minute` in order. Both of these queues can't really be autoscaled, so combining them isn't a big deal with your low background job volume.

Cost: 1, Benefit: 2

Recommendation: All background job classes should have a p99 of less than 30 seconds, fixed by use of Iterable.

Right now, you have about 15 jobs whose p99 execution (not queue time, time to actually run the job) latency is over 30 seconds. For most of these jobs, their p50 is also over 30 seconds.

Jobs which routinely take over 30 seconds to execute are dangerous. Sidekiq's shutdown timeout (and Heroku's kill/restart timeouts) are generally about 30 seconds. Jobs reliably only have about 30 seconds to finish and shutdown. Jobs that don't do that are a great source of idempotency bugs. Usually, no one notices this because these jobs don't execute frequently enough and are only shutdown/killed during execution infrequently… but it will eventually bite you.

High execution times also make it difficult to ensure quality of service. What does it mean to be "on the 5 minute queue" if executing the job *also* takes 5 minutes? Better to keep exec times reliably short, so we can treat them as a rounding error and focus on queue times.

Sidekiq 7.3 has introduced an incredible way to refactor these jobs: Iterable.

Let's take `QueueTicketEmailsWorker` as an example, which does this in a loop:

```ruby
def enqueue_due_ticketed_events
  TicketedEvent.in_ticket_dispatch_window.each do |ticketed_event|
    Rails.logger.info("TicketDispatch - started", log_data: {
      event_id: ticketed_event.id,
      event_name: ticketed_event.name
    })

    new(ticketed_event).enqueue_emails
    log_dispatch_overview(ticketed_event)
  rescue => e
    Bugsnag.notify(e, true)
  end
end
```

This usually takes about 2-5 minutes to execute. What happens when, 60 seconds in, this job is killed by a shutdown? This job is not idempotent, so all of the events which already had their emails sent before the job was killed will have that happen again.

Iterable fixes this - we would iterate over each event, send the tickets for that event, and then bump the iteration. It's extremely unlikely that a single iteration would take more than 30 seconds to complete, so we've effectively eliminated this class of idempotency error.

Cost: 2, Benefit: 2

Outcome: Reduce LCP to 3 seconds or less on most pages

Before we nuked the WebVitals controller, LCP looked okay on most sites but there are certainly a few that could be improved.

Recommendation: Use srcset with multiple image sizes

Consider unplugged.rest. Loading this page has about a 6 second LCP for me. The LCP event is triggered by a particular image on the page popping in (not uncommon for LCP). That image took about ~2 seconds to download, more or less because lots of images were also downloading at the same time (about 30-40). Over 7 MB of images are on this homepage!

These images have already been optimized at least somehow - I see lots of webp everywhere, so I know you're not doing nothing.

The problem is each of these images is enormous - at least 1080p. You simply can't download 40 different 1080p wallpaper-size images quickly, no matter what image format you use. And of course, depending on screen size, these images may actually be displayed at 100x50px!

What you need is responsive image helpers, which is going to be quite complicated for you because there's a lot of layers that have to happen:

1. You need a helper method with the correct `srcset` attributes
2. You need a pipeline for processing images and storing them at various sizes
3. You need at least double, maybe quadruple the size of storage you have today to make that happen.

I've audited ~half a dozen other popular Easol landing pages, and really the only thing holding back LCP is image download times. If you're already serving the correct format, serving the correct size is the best possible next optimization.

Cost: 3, Benefit: 3

Recommendation: Consider (more?) Turbo, and dividing panes with Turbo frames

Really, there's no way to improve frontend experience more than Turbo *or* an SPA. You turn ~3-4 second page navigations into ~0.5-1 second navigations. It's a completely different experience.

You already make some limited use of turbo Frames and Streams, but what I'm talking about is more like Drive: all Turbo all the time. Maybe that's a pipe dream, given that your clients expect to be able to throw arbitrary Javascript onto their pages and have it Just Work. But the juice is worth the squeeze, in my experience. There's no other way to chop your page navigation times by 50% or more.

You could also use Turbo to decompose your admin pages. We did this with great success at Gusto. For admin pages which take way too long to render, simply chop them up into a series of Turbo Frames, and let the backend "parallelize" the processing and rendering of an otherwise massive HTML response. I was able to chop up an ActiveAdmin powered backend with Turbo Frames in just a couple of weeks, and LCP and page load across the entire admin panel dropped by 50%.

Since you're already doing some turbo, this recommendation is really just asking you to keep looking for opportunities to do more.

Cost: 3, Benefit: 5

Outcome: Reduce deployment cost

You asked me to look at whether or not you have a lot of opportunity cost in remaining on Heroku. I've done the math and I think **switching off Heroku represents a limited cost savings at this time**.

Here's why:

You currently spend about $3000/month in compute (dynos). You spend about $4000/month in addons, roughly $3250 of that is on Postgres.

The DB cost is hard to get rid of. Easol definitely should not be managing it's own database. It's too important, the DB is under enough stress already, and as far as I know no one at company is/was a DBA. It makes sense for you to spend an extra couple thousand per month to get the "guaranteed" uptime and backups and easy upgrade/downgrade and everything else “managed” services provide.

Unfortunately, Heroku’s Postgres pricing isn’t that bad, so switching off doesn’t save you that much, maybe 10-20%. Your other databases (Redis, Memcached) have basically the same math involved.

So can we get rid of the compute costs?

Consider a move to Hetzner, for a 64 core machine for 500 euro per month. That removes $2-4k in compute costs per month, or a max of $48k USD per year. You would still have to bring up probably 1 or 2 additional hosts during ultra-huge launches like Vibee, but you could easily serve your current compute load off a single box.

You would definitely have more administrative time on these servers related to security hardening, deployment, and other problems. It’s easy to imagine this would take about 1/3 of a full headcount’s time to deal with every year. What’s that cost to Easol?

So it feels like a wash on compute costs, perhaps a net loss when you consider that there’s no more “dyno slider” to bail you out if things go utterly wrong without prior notice.

Summary

- Outcome: Improve your observability
  - Recommendation: Install RUM, ditch WebVitalsController *Cost: 1 Benefit: 3*
  - Recommendation: Implement the Speedshop Standard Dashboard *Cost: 2 Benefit: 4*
  - Recommendation: Ditch Datadog Profiling *Cost: 0 Benefit: 1*
  - Recommendation: Post Launch Events to Datadog as an Event *Cost: 1 Benefit: 1*
  - Recommendation: Terraform your Datadog monitors and SLOs *Cost: 2 Benefit: 2*
  - Recommendation: Use rack-mini-profiler in all environments *Cost: N/A Benefit: N/A*
  - Recommendation: (COMPLETED) Add Datadog database monitoring *Cost: N/A Benefit: N/A*
  - Recommendation: (COMPLETED) Send Heroku Postgres, Redis logs to Datadog *Cost: N/A Benefit: N/A*
- Outcome: Serve higher req/sec during launch events
  - Recommendation: Prior to Vibee launch, thug it out and crank that DB plan as high as it will go *Cost: 1 Benefit: 3*
  - Recommendation: Delete RUBY_ GC tuning variables, MALLOC_ARENA_MAX=2. *Cost: 0 Benefit: 1*

- Recommendation: Reduce query counts and p50 on launch drop pages *Cost: 4 Benefit: 5*
- Recommendation: Do not run high-SLO queues at maximum concurrency all the time, use 2x dynos rather than L. *Cost: 1 Benefit: 2*
- Recommendation: Reduce resources used for each launch *Cost: 2 Benefit: 3*
- Recommendation: Move the rack-attack memcache server to a separate machine *Cost: 1 Benefit: 2*
- Recommendation: Consider permanent over-scaling on web by setting the next day's minimum to be the same as the previous week's maximum. *Cost: 1 Benefit: 2*
- Recommendation: Increase minimum dyno counts in advance of launch events using Judoscale's API *Cost: 2 Benefit: 3*
- Recommendation: Prior to Vibee launch, drop a load-test nuke on your payment flow and try to smoke out any other resource contention *Cost: 3 Benefit: 3*
- Recommendation: Fix /cart/stripe_card_payment?order_id double Stripe hit *Cost: 2 Benefit: 1*
- Recommendation: Chart out your napkin math *Cost: N/A Benefit: N/A*
- Recommendation: (COMPLETED) Change DB vendors *Cost: N/A Benefit: N/A*
- Recommendation: (COMPLETED) Create a launch-focused dashboard *Cost: N/A Benefit: N/A*
- Recommendation: Create automated warnings on timeouts, SLO violations for web requests, and build a postmortem-driven incident culture. *Cost: N/A Benefit: N/A*
- Recommendation: (COMPLETED) Change WEB_CONCURRENCY to 8 *Cost: N/A Benefit: N/A*

- Outcome: Reduce p75 response times
  - Recommendation: If any controller has >5% of it's responses take longer than 500ms, create a ticket. *Cost: 0 Benefit: 2*
  - Recommendation: Ditch bullet, move to a fail-hard Prosopite-driven workflow *Cost: 1 Benefit: 2*
  - Recommendation: Have a seed/db pull process based on production data, particularly for pages *Cost: 3 Benefit: 4*
  - Recommendation: Lay the groundwork for code and experience ownership *Cost: 4 Benefit: 4*
  - Recommendation: (COMPLETED) Index all _id columns, enforce this via Github Action *Cost: N/A Benefit: N/A*
- Outcome: Reduce admin p99 response times and timeouts
  - Recommendation: Burn down admin pages that are timing out *Cost: 3 Benefit: 3*
  - Recommendation: Don't lock rows for update on Company::Bookings::EditsController#show *Cost: 1 Benefit: 1*

- Outcome: Make background job processing more robust
  - Recommendation: Create a within_0_seconds queue for jobs which a customer is actively waiting upon their completion *Cost: 1 Benefit: 2*
  - Recommendation: Throw a "chaos monkey" into your queue scaling using Judoscale's API *Cost: 1 Benefit: 2*
  - Recommendation: All background job classes should have a p99 of less than 30 seconds, fixed by use of Iterable. *Cost: 2 Benefit: 2*
  - Recommendation: (COMPLETED) Add SLOs to Datadog for all Queues *Cost: N/A Benefit: N/A*
- Outcome: Reduce LCP to 3 seconds or less on most pages
  - Recommendation: Consider (more?) Turbo, and dividing panes with Turbo frames *Cost: 3 Benefit: 5*
  - Recommendation: Use srcset with multiple image sizes *Cost: 3 Benefit: 3*
- Outcome: Reduce deployment cost