Speedshop Tune: Babylist

@nateberkopec

Prepared with care

@nateberkopec

2024-09-03

Contents

Hello Babylist!

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 actually just simple configuration changes you can make today that will save you a lot of money. Others are more long-term projects and skills that you can improve on over the next six months.

This document is organized at the top level by our desired Outcomes, which are my goals for your performance improvements over the next six months. Underneath that are specific Recommendations to achieve those outcomes. Each Recommendation has an associated cost and benefit, rated subjectively on a 5 point scale.

At the end of the document, I will present again the Outcomes and Recommendations without commentary. This is intended to be a quick reference to help you turn this document into action and assist during planning your sprints, etc.

I hope you enjoy this document and find it a useful guide for the next 6 months of performance work on your application.

Nate Berkopec The Speedshop

Outcome: Measure performance accurately

When it comes to measurement, I'm a stickler for methodology. Measuring the wrong thing is often worse than measuring nothing at all: it can lead to wasted work being done to fix problems that don't exist, or work being misprioritized based on a red herring of data. In addition, keeping performance data neat, orderly, and high-signal ensures that it will understood and used by as many people as possible at the company.

Measurement - in the form of metrics, traces, and aggregations of user experiences - is the foundation of any performance effort.

Recommendation: Get environments out of service names in Datadog.

Currently, your datadog service names often include environment names. For example: `bl-web-staging10-redis`, `production-rails-app`, etc.

In Datadog, it is intended that every object - a trace, a service, a metric - has an `env` tag, corresponding to the environment. This means that information for the same service can be easily aggregated across environments (for example, for all staging environments).

So, the recommendation is that in Datadog, **no service name should contain an environment name**, and all service data should be served with an `env` tag.

Regarding staging, it looks like you have multiple staging environments. I think it would be best to tag their `env` including the number, e.g. `staging1` or `staging2`.

It looks like how this is configured varies a lot based on what kind of service it is (DB, Rails, etc), so I can't give a specific recommendation here as to how to fix this.

Cost: 2, Benefit: 1 - somewhat tedious, but will really make the Datadog service catalog much more navigable, and would make comparing services across environments much more clear.

Recommendation: Rename transactions terminating in Rack middleware

Currently you have a fair number of transactions in the main Rails app that get named things like `GET 200` or `OPTIONS 200`. This is because these transactions are terminating in Rack middleware, i.e.:

```
class MyRackMiddleware
  def call(env, status, app)
    return 200 if do_some_business_logic
    @app.call
  end
end
```

In this case, Datadog doesn't know what to name the transaction. There's an option to make Datadog use the name of the last executed Rack middleware to name the transaction instead. Sounds perfect. Here's a PR to turn it on

Cost: 0, Benefit: 1 It's already done!

Recommendation: Create a stack of a performance dashboard, monitors and SLOs

You should have a single dashboard which contains the following information for the main monolith - it's not specifically scoped the `production-rails-app` service, however. It does represent all the important performance metrics surrounding the "main monolith".

The dashboard should include:

- Latency/Customer 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.
  - Number 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 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
    * CPU (load and utilization)
    * IOPs
    * Read/write latency
    * Error rates
  - Memcached service
    * Memory utilization %
    * Hitrate
    * CPU Utilization %
    * Error rate
  - Redis services
    * Memory utilization %
    * CPU Utilization %
    * Error rate
  - Error rates
    * Web, worker
  - www.babylist.com uptime

For each number on this dashboard, there can and should be an associated SLO and an associated monitor. This means that each number has associated "good" and "bad" values.

Cost: 3, Benefit: 3 Much of this instrumentation is missing, and many of the "SLOs" have not yet been set. However, a complete set of SLOs, monitors and this dashboard would represent a complete picture of the performance state of the app.

Recommendation: Remove Sentry RUM and use Speedcurve or Datadog

I tried your Sentry real-user-monitoring setup (Sentry calls this their "Performance" product) and, while it was fine, I think you've already got Speedcurve and it's much better.

The shortcomings of Sentry were:

1. No way to change thresholds for Apdex to a number that's actually meaningful. The only options are LCP and window.load.
2. No accurate page load metric for iOS safari. iOS safari doesn't report LCP, so for >50% of your traffic, you have no accurate load time number. Ouch.
3. The filtering UI is kinda weird, has some very strange visualizations that I think are just confusing rather than helpful.

Speedcurve has a few advantages:

1. Lighthouse stuff is integrated quite nicely. While I'm not like a huge Lighthouse fan, it's nice to have that running in a synthetic way constantly so you can quickly see the results.
2. Has a nice "budget" structure, essentially the equivalent of a Datadog SLO.
3. Filmstrips are quite nice.
4. Very easy to use Synthetic testing

So, between the two, I think Speedcurve is the clear winner for me and I don't see an advantage of paying for both.

As far as I can tell, it looks like both Speedcurve and Sentry are costing you about $500 a month.

One other alternative here would be to just use Datadog RUM. You currently get about 750k sessions per month, and Datadog's annual pricing is $1.50 per 1000 sessions, for a cost of $1125.

Advantages of using Datadog RUM:

1. Now RUM data is in the "Datadog warehouse". It can be correlated and compared with backend APM data, infrastructure data, turned into metrics and stored forever, etc. It can appear on the same dashboard as our previous recommendation, etc.
2. RUM data now be set up with monitors, SLOs, alerts in Datadog.
3. The feature set is broadly equivalent to Speedcurve, and definitely better than Sentry.
4. Datadog has their own home-cooked page load metric which I rather like (it's based on no DOM/network activity for a period of 5 seconds) which works with iOS Safari.

Disadvantages of Datadog RUM:

1. To keep cost the same, you'd probably have to sample at 50%, which means the total session number would no longer be correct/true as it is in Speedcurve/Sentry.
2. Synthetic testing is more difficult to use.
3. Default dashboards in Datadog are not as nice as the ones in Speedcurve.

If cost were no object, I'd just run Datadog RUM at 100% sample rate and call it a day.

Cost: 0, Benefit: 1: This recommendation is more about cutting a small amount of cost than improving any particular customer experience.

Recommendation: Implement a custom "page is loaded" event in RUM

Determining "when a webpage is loaded" is actually quite hard to do in a way that's agnostic to how that webpage works. Largest Contentful Paint is the best attempt anyone's made, but it's not perfect.

As mentioned before, the biggest drawback of LCP right now for you is that it's not being sent by >50% of your traffic (iOS Safari). That size of an observability blind-spot is not great.

All RUM tools allow you to implement custom timings and events. I think Babylist can ship one of their own that represents "this page is loaded".

At Gusto, that event was "when there are no more Loading spinner elements rendered by the React app, the page is loaded". I'm not sure yet what that could be at Babylist. You need something that's generally aware of how the app is architected and works, yet is agnostic to the particular page being displayed. So far at Babylist, my experience is that the frontend is a bit of a mishmash and the rendering approach might be significantly different from page to page. We might just implement something of our own similar to Datadog where the page is loaded "after a few seconds of inactivity" on network/DOM? I'm not sure.

Without an accurate number representing "this is how long someone waited until the page is usable" we are really stabbing in the dark on customer experience.

Cost: 3, Benefit: 4 This kind of project can end up being difficult to implement yourself. If we use Datadog's load number and decide it's "Good Enough", this becomes Cost 0, but I'm not sure that will be the case. This number will basically become the foundation of all of our customer-experience efforts, so that's why the Benefit number is high.

Outcome: 99.99% Uptime

One thing I took from Jaime after our initial conversations was the value of uptime - at Babylist, uptime is worth at least $1,000/minute, just in sales/revenue terms. Of course saying that "the site is down for 1 minute, that costs us $1,000" is fudge-y math (people retry things after they're fixed, reputation damage, etc) but it's a great starting point.

Three nines of uptime is 43 minutes of downtime a month, and four nines is about 4 minutes. That's a $40,000 a month difference at Babylist, something probably well worth our time to figure out. Going from 4 to 5 nines, though, eh, that's probably not going to get anybody major plaudits. 4 nines is a great target for this level of business.

While Babylist is overall in very good shape, I think there's a few areas that could cause trouble in the future or be the cause of a future incident. These "footguns" are also included in this section.

With this outlook in mind, these recommendations are focused around potential performance-based threats to uptime.

Recommendation: Move to SLA-based queues

Every background job has a nautral deadline of "when it should be done". In a well-set-up background system, each job is also usually quite short, taking 30 seconds or less to execute (when jobs take longer, they may be interrupted mid-execution by a shutdown, which is dangerous).

That means we generally can say that the time a Sidekiq job will take to finish is:

```
time_spent_enqueued + 30.seconds
```

So really, the main component in how long any Sidekiq job will take to execute is the time it spends in the queue. That's the number we have to control.

But how long is acceptable to spend in the queue? This varies widely from job-to-job.

Some jobs are what I call **customer-blocking**. A customer is actively waiting on their completion, right now. The job completing *unblocks* that customer and lets them go do something else. Some examples of customer-blocking jobs:

- Password reset emails
- Any job where completion is announced in the web UI by WebSocket
- Returning an item back to inventory (it was "reserved" temporarily by a customer, now we're making it available again)

For these jobs, the acceptable amount of time_spent_enqueued is basically zero. That makes these jobs very special. They should live in their own queue, which will usually be quite heavily resourced. Customer blocking job queues cant even be effectively autoscaled, because by the time you bring up new capacity via an autoscaler, the SLA is alread long-ago breached.

On the other end of the spectrum, there are jobs that could be executed 24 hours from now and still be fine. Backfills, reporting, and more are not particularly time sensitive at all.

In my book Sidekiq in Practice I call these **SLA queues**. Each queue is *named after it's service level*. This is a very powerful idea which allows a number of things to happen:

1. Product engineers are forced to set a latency SLA for every single job by assigning it ito the appropriate queue.

2. It is extremely clear regarding what number to set your monitors, SLOs and alerts to for these queues (it's in the name!!!).
3. Autoscaling is a breeze. If the SLA is under threat, scale up!
4. Load shedding is easy. Database under heavy load? Pause the 24 hour queue… it can wait a while. More on this later.

Cost: 3, Benefit: 4 It's a significant migration, as every job will need to be recoded to a different queue. This usually involves a lot of manual "thinking" labor. However, the clarity this brings around service levels expected and obtained out of background jobs is incredible.

Recommendation: Autoscale Sidekiq based on queue latency

While SLA-based queues make "when to autoscale" very clear, they're not strictly necessary for this recommendation to work.

As previously mentioned, every job has an inherent SLA expecation. Hopefully, every job in each queue you have has the same SLA (again, may be true today, maybe not).

Since "how long it takes jobs to execute in total" is what we care about, this number should also be the basis of our autoscaling.

Autoscaling based on CPU is particularly fraught for background jobs, as the amount of time spent waiting on CPU is highly variable based on the job type. Imagine you have a job that spends 10 seconds waiting on an HTTP call, and then you enqueue 1 million of them. What's the CPU usage going to look like while we run those jobs? Will your autoscaler ever trigger?

Instead, each job should report, via a Sidekiq server middleware, how long it has spent in the queue before being executed (queue time). This number can then be piped into your HPAs using a custom metric.

You can also do some fanciness around prediction here, but it's generally not necessary. For example, in a 5 minute SLA queue, I would scale up when queue latency reached `5 minutes - how long it takes to bring up a new pod` and scale down when queue latency was less than ~20% of the SLA.

Cost: 2, Benefit: 3 The only hard part is really getting the piping set up the first time for the custom metric. After that, just requires some thinking caps on regarding SLAs. This is a Cost 1 task if you are using SLA queues.

Recommendation: Break up or iterable-ize your 10 longest running Sidekiq jobs.

I mentioned previously that it's good if Sidekiq jobs take 30 seconds or less to execute. There's two main reasons:

1. The Sidekiq shutdown timeout is 25 seconds by default. Jobs that always execute in this time or less don't get interrupted by shutdowns.
2. Jobs which take up capacity for long amounts of time have a big impact on the queue time of that queue. For example, imagine enqueueing 10 jobs that all take 60 minutes each on a concurrency 10 queue. What happens?

You have a handful of jobs which reliably execute in over a minute. Simply look at the `rails-app-sidekiq-worker` service and sort descending by "p75 latency". These jobs are dangerous!

Most of them are in BLFulfillment and run on a daily schedule.

You have two options here:

1. Break them into a parent/child job structure, where the parent job "fans out" to X number of child jobs and each child job executes a portion of the job.
2. Use the new Sidekiq 7.3 "iterable" features to at least safely iterate through large amounts of work and deal with interruption, and then put these jobs in a high-SLA queue of 1 hour or higher to minimize impact on queue times.

Cost: 2, Benefit: 2 The work here is straightforward but can be a little bit involved - luckily you really only have to do it for 10 jobs. I don't think currently that these jobs are having a big impact on queues, but they are certainly footguns that will blow up in the future.

Recommendation: Merge the Schema Cache PR, aim to cut p95 request queue time

I think your p95 request queue time when running a deploy is a little high. It's probably not really noticable for anyone using the site, but I would say that this magnitude of request queue disturbance is not typical for my other clients, so there's probably room to improve here.

I don't really have a great idea of where to go here. I think the schema cache PR you already put up is a great start: ship it. I don't agree that the best place to do this is in the entrypoint, but I don't have solutions to the disadvantages of doing it in the image.

At least doing it in the entrypoint we still solve the problem: servers come online with a hot schema cache. Yes, they also take longer to boot now, but I think we can afford to have that problem (and ofc it's an easy revert if that's a big issue).

Once this goes out, I think we need to reassess the degree of request queueing and decide if more investment is necessary.

Cost: 0, Benefit: 1 Code's already done, just do what you have to in order to hit merge.

Recommendation: Shed load using background job queues, either automatically or manually

On Prime Day, you had an incident which involved DB load going too high. This is only going to happen more as you grow.

A very useful tool during these kinds of scenarios is the ability to pause background job load. Background jobs can be a real firehose of database load, and they tend to correlate with web load (lots of people ordering things creates both web and background load). The ability to pause this load until the storm subsides is a valuable tool in the responder's toolbox.

However, SLA queues are *basically* a prerequisite to this. Without it, responders need to consult potentially-out-of-date documentation regarding which queues can be paused, and for how long.

Think through this scenario in your head and ask yourself if an incident responder can do this today:

The DB is going down under heavy load. The responder wants to pause all unnecessary background job load.

1. Which queues can they pause?
2. For how long can each queue be paused?
3. Can you give them any idea what the impact of pausing each queue will be?

At Gusto we also had one or two "unpausable" queues. We simply name these queues something like "5-minute-DO-NOT-PAUSE" to denote that they could not be paused for any reason.

Cost: 0, Benefit: 2 This is basically just a benefit of SLA queues. During periods of high database usage, your background job setup can generate pretty high amounts of DB load (24 hours spent waiting on the database for every hour, in the most recent week), so I think the payoff here could be high.

Recommendation: Set SQL database pool based on thread concurrency, controlled by a single, consistent ENV variable

Currently your database pool is set like this:

```
pool: <%= if ENV['SERVICE_NAME'] == 'worker' then 100 else 25 end %>
```

This *works*, because ActiveRecord's connection pool is lazy. You don't use a connection from the pool unless a thread needs it. Your actual thread concurrency is much lower than the number here in most environments (like less than 5 for web at least!) so most likely these limits are never hit. You might as well set this number to a million or something for all the good it's doing you here.

I'm not a huge fan of this as it means that connection pool leaks will never be detected, because even in a leak scenario you won't hit pool limit. That could mean you end up have a ton of idle connections on the database, slowing it down, and you don't notice.

My preferred db pool set up is:

```
pool: <%= ENV.fetch('RAILS_MAX_THREADS', 5).to_i + 5 %>
```

That way, we've got a small buffer for minor leaks, but it's always "right-sized" for our actual level of concurrency. Then, I use `RAILS_MAX_THREADS` everywhere to set concurrency. Puma and Sidekiq will both read this value to set their thread pool size by default, no configuration required. You can also set your Redis connection pool sizes this way, though that's less important as idle Redis/memcached connections don't use as many resources as an idle Postgres connection.

Cost: 1, Benefit: 1 Housekeeping, really.

Recommendation: Move to BigRails connection management for Redis, cleanup what's stored in which Redis database.

You have four Redis databases in use by the main app:

- Sidekiq (aka $redis_worker)
- Redis web (aka $redis)
- Rack attack
- Sidekiq limiter

The first part of this recommendation is that **using globals without a connection pool is not thread-safe**. You should migrate both of these globals to a connection pool. Yes, you're not using these today in a multithreaded environment, but someday someone probably will, and they will get footgunned by this.

The second part, which I think is more serious, is that you should **stop mixing cache and critical app data in $redis**. You sometimes use $redis as a cache, and sometimes to store values which cannot be expired without breaking correctness.

The problem is that these two use cases require fundamentally different eviction strategies. Caches are designed such that any key can expire and later be a cache miss and everything is fine. Putting application critical data in that same database which *absolutely cannot be expired* is a recipe for eventual disaster.

For example, just taking the locking code I previously linked. I think the eviction policy for $redis is the redis default, which is `volatile-lru`. Let's say for some reason the caching use case suddenly puts a lot of pressure on the database and

brings DB memory usage to 100%. Suddenly, your job locks are on the chopping block and could be evicted far earlier than you expected!

You should:

1. Move all cache usecases out of $redis into Rails.cache.
2. Set the expiration policy on all Redis databases to noeviction.
3. Use `bigrails-redis` to manage Redis connections for all four databases.
4. Remove all non-connection-pooled use.

Cost: 3, Benefit: 2 It's a fair bit of work, but the footgun potential here is not to be sniffed at.

Outcome: Improve page load time

When I write Outcomes in these reports, I usually like to give concrete estimations and targets for how much we're going to improve the number in question.

However, in this case, I can't do that. We don't have a number that I trust for frontend page load time. That number would:

1. Cover iOS Safari page loads (not covered today by LCP)
2. Cover all cold and warm (React-powered-route-changes) loads.
3. Be hooked in to how your app actually works (based on loading spinners or some other threshold we decide for "this page is ready").

Without that number, I can't give a target for improvement. However, I do know improvement is possible, because there's some low-hanging fruit that we should pick in any case.

Recommendation: Optimize 4 main web transactions.

On any app, page load times are going to be in the ~3+ second range. That means that for the most part, if 75% of the time (p75) our web actions respond in ~500ms or less, they're not making up a noticeable portion of pageload time and we should focus on frontend.

However, you have a handful of web transactions that are reliably slow that would make a 5-20% difference in page load time if we optimized them down:

1. **BlRegistry::API::V3::RegItemsController#index**. This is on the critical path for registry viewing. Fixing it is pretty straightforward - it simply has N+1 issues across about 2 dozen tables. p75 is 700ms, we should shoot for halving that at least.
2. **BLRegistry::API::V2::RegistryController#show**. Not for the webapp, but the mobile. Bad N+1 issues. Luckily, its across fewer tables (like 5 or 6), so may be an easier fix. p75 is 1.5 seconds or so. Again, we should shoot for 300ms or less.

3. **BLRegistry::API::V3::ReservationsController#index**. Unsure of the impact here, because I think a substantial number of these are pre-renders. It gets a lot of volume, though. a p75 of 700ms with this many N+1s (5-6 tables) should easily be optimizable down to 300ms.
4. **BLRegistry::API::V2::ReservationsController#index**. Unsure how much overlap there is here in the code with V3, but the underlying cause is the same, so I'm gonna guess we can kill 2 birds with one stone here.

Simply fixing the n+1 issues across these 4 controllers would go a very long way to improving the full backend performance picture on important pages.

Cost: 3, Benefit: 2 It's a limited number of endpoints, and they're not THAT big of a component of total page load time. I find that fixing the first "batch" of N+1 pages is tough, then you start to set up processes and tools that make future fixes easier.

Recommendation: Profile allocations in V3::RegItemsController#index

I **love** that you're tracking this by the way.

Each week, you have about 4k requests allocate more than 10 million objects. That is *a lot* of objects and will almost certainly result in a very slow response, an increase in GC activity and probably a lot of unnecessary memory usage.

Of the requests allocating more than 10 million objects, 50% of them are the RegItemsController#index action.

Weirdly, most of these high-allocation requests are currently for a single registry: `5192132`, and some of the requests involve high offsets, which suggests bot activity.

Even if this is not "real" traffic, it has the potential to blow out your memory use.

Ideally, you clone this registry down locally and repro the issue with `memory-profiler` and see where all the allocations are coming from.

Cost: 3, Benefit: 1 Might be just a curiousity.

Recommendation: Profile allocations in V2::RegistryController

Something different is happening on this controller. Each week, you get 4 million requests that allocate more than 1 million objects. That's bad for the reasons enumerated in the previous reccomendation. Of these requests, 2.6 million (over 50%!) are from V2::RegistryController.

I *suspect* just fixing the N+1s on that controller will probably just fix this, however sucking down a few registries locally and repro-ing with `memory-profiler` will probably also help a lot to debug.

Cost: 2, Benefit: 1 I'm more convinced that fixing this one leads to a bigger latency gain on the controller action than overall memory improvement.

Recommendation: Create a seed or "registry pull" process

Your performance issues on the web side are almost entire N+1. In my experience, this is fundamentally caused by developers having unrealistically small and inaccurate (i.e. not completely "filled out") data in development. You can't fix what you can't reproduce.

Luckily, N+1s are not a matter of pulling down entire databases. You only need all the data actually related to a single registry (i.e., everything you see in the trace view on Datadog for a single request). The actual size of data required is not large at all, maybe a couple thousand rows.

There are a number of challenges you have to figure out though:

1. Anonymizing PII
2. Figuring out *which* data to segment out exactly.
3. Making it easy to use for every dev

The solution I saw at Gusto was completely custom-coded, probably only possible if you have 10x the engineering resources that Babylist has. There are a number of vendors and OSS projects (i.e. Snaplet) which purport to do this for you, that may be a possible avenue.

Cost: 3, Benefit: 3. Working with realistic data really does provide a big benefit. Devs love fixing perf problems, but they need to be able to repro them locally to do so.

Recommendation: When viewing registry as a guest, fetchReservedRegItems should only fire once and it should not wait on page readiness.

I noticed this one while looking at the Network graph in Chrome as a guest.

`fetchReservedRegItems` causes a request to the backend.

This is installed as an effect hook on the `RegistryPageGuest` component.

The problem is that this function is not idempotent, and something in the dependency array is firing twice (showGiftTracker changing twice I guess?).

This causes the network request to fire twice unnecessarily.

It's also unclear to me why the network request needs to block on page readiness. It should fire as soon as possible. We generally don't care about the number of parallel requests we're making to the backend when it comes to frontend perf. Blocking on page ready means the modal (which I think is where this is used?) now has an extra loading state that isn't necessary.

You can fix this either by making the fetchReserveRegItems function idempotent or by moving it out of the hook.

Cost: 1, Benefit: 1 Housekeeping issue.

Recommendation: Pick a way forward with frontend: SPA or Turbo.

Currently, almost every click on babylist.com is a "cold" navigation. You change the URL, toss away the old page and start on the new one. Just like grandpappy did!

This is currently taking at least 3 seconds per page load, probably more on mobile (again, hard to tell re: the Safari iOS data).

Using *either* Turbo or a SPA approach will get this down to 1 second on ~66% of your page navigations.

There is simply **no other recommendation I can make which will have as big an impact on page load time as this**. Moving as many page loads as possible into warm/hot navigations will have a massive impact on the feeling of using the site, decreasing overall page load time by 50% or more.

It really doesn't matter from a perf perspective which approach you pick: traditional Rails with Turbo or some React-powered behemoth. There are obviously other, non-perf considerations here. But you need to pick a path and start that migration.

Cost: 5, Benefit: 5. Moving to a world where the Javascript VM is not restarted every time you click is painful. Lots of things can and do break. But there is no other recommendation in this report with as big of an impact on customer experience.

Recommendation: Reduce JS bundle size

This one becomes less important if you follow my previous recommendation.

Your JS bundle is not small. Vendor and common, which block page load, are 1.5 MB on the wire and almost 10MB uncompressed.

Uncompressed size correlates to time spent compiling and running JS, and Speedcurve indicates that pageloads are spending around 2 seconds on JS long tasks on each pageload. That's quite a lot.

There are a number of bundle-cleaning approaches. I'm sure you've probably tried a few. It may be worth trying again, because that is a big, big bundle.

Cost: 2, Benefit: 2 Bundle-pruning is usually not that painful. JS download is on the critical pageload path, so any major reduction here could have a very measurable benefit.

Outcome: Reduce infrastructure spend by $299,000/year

In a previous recommendation, we talked about how uptime is worth $1000/minute, and going from 3 nines to 4 nines (that is, from 40 to 4 minutes downtime per month) is worth $40k a month to the business.

In my estimation, you're currently spending in a few ways which doesn't materially impact that uptime number, but does cost you a lot of money. I'd like to trim some of that fat.

For all infrastructure spend, I'd like to:

1. **Squeeze** the most out of what we've got. Configure things correctly to maximize throughput.
2. **Remove** "HA"-motivated spending where it's not delivering sufficient guarantees that it will materially impact our uptime.
3. **Right-size** all components to the actual load they experience.

Recommendation: Set Sidekiq concurrency to 10 everywhere, CPU limit to 1, and memory limit to 4GB. ($10k/year)

File this under "squeeze".

I cover this in more detail in Sidekiq in Practice, but I basically believe that 10 is the optimal concurrency number for almost all Sidekiq workloads. Setting it higher has minimal benefit in terms of throughput but can lead to increased service time (time spent running the job), and setting it lower can leave too much throughput on the table.

CPU limits on Sidekiq should just be set to how much CPU it can ever actually use - 1 full core. Memory limits should be set to what the m-series allows in terms of memory-to-CPU ratio, aka 4GB per core.

I would probably set the *requests* for about half that, based on previous load.

Currently bl-web-sidekiq-default is set to a 2 core request with 4G of memory. The 2CPU request is not necessary and causes 2x overprovisioning, as a single Ruby process can create more than 1 CPU core worth of load.

Cost: 0, Benefit: 1 Twiddle some numbers, reduce the infra allocation of your Sidekiq fleet by 30 cores, or about $10,000 a year, assuming this boils down to the nodes provisioning 30 fewer cores. My cost assumption is based on on-demand pricing.

Recommendation: Autoscale Puma based on a combination of request queue latency and utilization/busyness. ($240k/yr)

Web autoscaling works like this:

1. We want a web request to spend 20ms or less waiting on an empty Puma process, 75 percent of the time.
2. We should scale up if that threshold is breached.
3. In general, around 50% of our Puma workers should be busy and processing a request at any given time. If the number is lower than that, we are probably overscaled.

You currently scale based on CPU rather than request queue time. The disadvantage of this is that it's a second-order metric from the one we actually care about (time customers spend waiting). A number of things (I/O changes in the workload) can cause this metric to be inadequate. It's also hard to set. Is 60% CPU utilization bad? Is it good? I'm not sure!

Web pods currently run at a 2 CPU limit, with 9GB of RSS, and 6 workers each. On average during a week you're running almost 600 containers, and during peak load as many as 2000.

Double that number and we get an idea of how many CPU cores you have available to process work/ability to do work in parallel.

You average 331 req/s over a week, with peaks at 1.3k req/sec. Average response time 150 milliseconds.

Using Little's Law, we can get an idea of the long-run concurrency of the whole system. 331 req/sec multiplied by 0.15 seconds per requests is 50. That means that, on average, you are serving 50 requests concurrently at any given moment.

So, you're serving 50 requests but provisioned for 1200. That's a utilization of less than 5%. Not great. During peak periods, it's the same. Ideally, I think that number would be around 50%.

Autoscaling Puma based on request queue latency instead would get us much closer to this number. You simply have to report it somewhere (Datadog? Cloudwatch?) and then pump it back to the HPA. You will need to capture the request queue time metric yourself, which has had some methodological problems in the past so we'll have to do that carefully this time.

Cost: 2, Benefit: 3. Removing about 450 containers on average (probably on average also 450 cores of resource reservation) should save about $20,000/month or $240,000 per year. Getting the HPA right here can be tricky.

Recommendation: Set Puma to 10 workers on 8-core pods with 32 GB requests.

Currently you're running *very* overprovisioned on Puma workers to CPU cores. You've got 6 workers on a 2 CPU setup. This makes me nervous.

What happens here is 6 workers can ingest 6 requests at the same time, and then fight it out over 2 CPU cores. This is probably:

1. Prematurely triggering autoscaling.
2. Causing request queueing during restarts.

Yes, you will need to have a *somewhat* higher number of Puma processes than CPU cores for optimal throughput, because you're running single threaded and so therefore need extra workers to use all available CPU.

However, the amount of time that the app spends on average waiting on I/O is less than 25%.. We can use Amdahl's law to get an estimate of how much we should overprovision (I'm not gonna do the math here) but it's going to be at maximum about 25%. 10 workers with 8 CPU seems good to me.

I also want a slightly higher number of workers per pod than you currently run in order to reduce request queueing further. More workers per pod == less chance 100% of them are busy == less request queueing.

Cost: 0, Benefit: 2 Unsure on benefit here, but it's just number twiddling.

Recommendation: Downsize the main database to r7g.8xlarge. Move read replication further down the backlog. (18k/year)

I have a feeling this is gonna be a spicy one.

You had a database-fueled incident during prime day which caused the organization to decide the solution was a bigger database.

The reasoning for the DB upgrade looks flimsy to me. You never exceeded your IOPS thresholds, and CPU load on the main database was only temporarily extremely high (this isnt a daily occurence) and could easily be scaled up/down in anticipation of a prime day next year.

Post-upgrade, a 20ms-per-request reduction in DB wait time was noted. That's not really doing a lot for anyone. 20ms is not noticeable for the customer, and in terms of throughput/load, you're gonna do a lot more by rightsizing your deployment than by reducing time spent on each request.

In addition, I can't be sure but I think the reason for that 20ms per request improvement was the increase in total available memory, which RDS then automatically just uses to increase buffers/cache. You could have made a lateral move into an r-series instance instead and got that memory without the extra cost.

In the past week, CPU utilization maxes out at 15% and about 4 sessions.. This is on an m5.16xlarge, which is $4700/mo ondemand for 64 cores, 256MB and 60k baseline/max IOPS. An `r7g.8xlarge` is $3127/mo for the same memory, half the core count (which you're doing fine on!) and 40k IOPS (which you're also fine on). If you agree with me that 20 milliseconds isn't worth $18k a year, you can go to the 4xlarge and still be fine.

If your true DB load is around a 4 to 8xlarge, that means you can continue vertically scaling by at least a factor of around 2 to 3. r6i.32xlarges are available with 128 cores and a terabyte (!!!) of RAM. Replication and sharding are *not* simple tasks, and frequently don't "pay off" in terms of reduction the primary as much as you'd like. In the cost/benefit curve, I think you should be thinking in terms of reducing load on the primary and scaling vertically at this point.

ProxySQL, read replication are not simple engineering tasks. I'm more of the mind that could easily vertically scale this to a 2x higher load (at *least*) and, if you're still feeling pain, then you can consider these offramps.

Cost: 1, Benefit: 2 $18k/yr in savings at least for another db swapover.

Recommendation: Reduce memcached to 1 r7g.large node. ($16k/year)

`web-memcached` is currently set to 5 nodes on `r6g.xlarge`. That's $1500/month on-demand.

I'm not sure of the reasoning behind 5 nodes. Each node has 26GB of capacity. It means your cache graph looks very funny - it starts at 128GB of memory free and then slowly reduces until the cache is fully turned over for some reason.. This is not the way a cache is supposed to work. A cache is supposed to be constantly churning and evicting things and more or less operating at 100% utilization with a >90% hitrate.

You'll get by fine with a single r7g large (13GB).

Cost: 0, Benefit: 2 Number twiddling leads to $1300/month in savings.

Recommendation: Slash Redis database sizes ($5k/yr)

You've got a few different Redis databases, but none of them are caches (or they won't be after we implement all this) so all of them memory usage should never approach 100%.

In the past 3 months, only `web` has come close to utilizing this much. The others all use less than 5%, all the time.

`rack-attack` (this one also on `dw`), `sidekiq-limiter`, and `redis-sidekiq-worker` are all on m6g.xlarge (13GB), at $216/mo. I think these could all be bumped down at *least* one size to a m7g.large, for $115/month.

Consider also the cost of the HA replicas here (doubling your costs). In total (including web), they're costing $750/month. Do they prevent at least 1 minute of downtime a month? Maybe.

Cost: 1, Benefit: 1 These db switchovers are more complex than memcached (cause you can just blow that away). This is worth $5000/year.

Recommendation: Remove the Sidekiq high-memory deployment by auditing Sidekiq memory use

You already report allocated object count for web.

I think you could use something like getprocessmem to create a Sidekiq middleware that reported RSS before/after each job. It's a pretty low-overhead system call. Technically it won't be 100% accurate because of concurrency/multithreading, but I think it would be close enough to clue you in as to what jobs exactly are causing high memory usage with what job arguments.

From there, you could use `memory-profiler` to figure out how to reduce this memory usage.

The benefit would be one less sidekiq deployment to manage, and getting more jobs into the "blessed" SLA-based system.

Cost: 2, Benefit: 1

Recommendation: Multithread Puma to remove 15% of your web fleet. ($2,000/yr)

So, I'm actually going to tell you *not* to do this. I know. Hilarious. I'm the maintainer of Puma, telling you not to multithread your app.

Why multi-thread? It doesn't make your app any faster. The only reason to multi-thread is to increase concurrency per unit of infrastructure (CPU cores). And when your app is spending less than 25% of its time on I/O, that gain is quite limited.

If you implement my autoscaling recommendations, switching to Puma with 5 threads probably only improves your throughput by 10-15%. It's just not going to register.

We've already found 1 major reason you can't multithread on web ($redis), what other major multithreading bugs might be lurking?

Cost: 0, Benefit: 0. Forget it.

Recommendation: Reconfigure the fleet to use m7i.2xlarge.

Your current "main fleet" is on the `r6a.2xlarge`. Given the memory usage of the app, it looks like 1 CPU core worth of load uses less than 4GB of RAM. To

me, that shows you could fit almost all your workload onto an m-series.

r6a is an EPYC 7R13. The m7i is a Sapphire Rapids Intel Xeon, available for basically the same price. It should be a roughly 10% gain in straight-line single thread speed for no change in price.

Cost: 0, Benefit: 1 Marginal gains, but just slot it in for your next blue/green swap.

Outcome: Reduce build times to less than 10 minutes.

No one likes waiting on builds. I mean, I know that you spent 2 weeks waiting on a pull request review, but the extra 10 minutes waiting for the build to go just adds to the sting, right?

Recommendation: Consider self-hosting

I think the future of CI is self-hosted runners. There's so much juice you can squeeze here. You can at least get your CI spend on to your own AWS bill, which gives you more power for Savings Plans or reserved instances or spot instances or other shenanigans. Or, you can just use a completely different provider entirely. I've seen a lot of success with CI on Hetzner, which has instances with very high single thread speed.

Cost: 3, Benefit: 2. Too hard to be certain as to the "CircleCI tax" on latency.

Recommendation: Get Journey tests off the critical path

I could be missing something here, but I'm not sure why the Jest tests don't need to wait on `build`, but the Journey tests do. Making Journey not depend on `build` would move it off the critical path (and put the pressure on Jest/Rspec, which can be parallelized further).

Cost: ??, Benefit: 1

Recommendation: Keep Cranking That Parallelism, Baby

Your test examples aren't really that bad. You run ~1500 examples per minute on 8 cores. That's 187 per minute, 24 per core per minute. I think 1 example/core/second is more reasonable, so maybe there's a 50% gain lurking around here if you change hosting provider, but that's really guessing.

Especially if you move to cheaper self-hosted runners, running 26/27k examples is just a matter of how much parallelism you want to crank. Gusto was doing 300+, you're at 32. Take all that money we're going to save on infra and use it to make your devs a little happier.

Keep in mind there's no point in parallelizing further once Jest/Rspec are faster than the build step.

Cost: 3, Benefit: ?? It won't be cheap (I think), but who can put a price on 5 minute builds?

Outcome: Successful iterable integration

Had a quick think about this one, this is just napkin math/off the dome thoughts I had last week.

Recommendation: Treat this as a microservice, run on its own fleet of 40 Puma workers, push work to Sidekiq and batch/flush writes.

The main requirement here is to take 8 million web requests over a very short period of time and turn each of them into a row in the database. This 8 million request dump can come at basically any time, and for the rest of the time, service usage will be minimal.

We want to:

1. Respond to Iterable with a 200 OK within a few seconds, 99.99% of the time or more.
2. Turn each 200 into a row in a SQL DB *eventually* (not particularly latency sensitive here).

This is a system that wants to be tuned for high availability, with a tradeoff of high latency (both on the write side and on the 200 OK side). Iterable's webhook bots will accept MUCH higher latency than any human customer will, and we don't particularly need these database rows to appear microseconds later. A few minutes will do.

This requirements are quite a bit different than your main application, particularly on the web side. Ideally, this web service responds extremely quickly (a couple of milliseconds) and it must be kept at high capacity at all times (to deal with a sudden million-request influx). However, it can also deal with high request queue latency. This is very, very different from the main bl app.

The app also basically just needs to turn a simple webhook into a simple Sidekiq job. It doesn't need to interact with the rest of the BL monolith, code-wise.

40 Puma workers, with response times of ~5 milliseconds, could easily handle 4000 requests per second or more. Slap that on some `m7i`s, it will cost about $1200/month. You probably won't autoscale this (depends on the details of Iterable's retry/backoff behavior).

As for the worker side, I'm a bit more neutral. You could decide to put this in the main app, with the rest of the SLA queues. It could also have its own

queue/deployment combo.

You should also consider whether or not this should go in its own SQL DB. In Rails these days its quite easy to put models into their own database. Of course, you can't join them then - but how often will you do that? I don't know.

As for the workers itself, I would have a rolling cron job start something up every minute to gather the batch of writes to be done, package them into 1,000-row insert statements, and then flush. It will be *critical* that this job can be easily paused by ops if necessary.

(Side note: I did the math on Lambda. It doesn't work out. Don't bother.)

Cost: N/A, Benefit: N/A.

Summary

- Outcome: Measure performance accurately
  - Recommendation: Implement a custom "page is loaded" event in RUM *Cost: 3 Benefit: 4*
  - Recommendation: Remove Sentry RUM and use Speedcurve or Datadog *Cost: 0 Benefit: 1*
  - Recommendation: Rename transactions terminating in Rack middleware *Cost: 0 Benefit: 1*
  - Recommendation: Create a stack of a performance dashboard, monitors and SLOs *Cost: 3 Benefit: 3*
  - Recommendation: Get environments out of service names in Datadog. *Cost: 2 Benefit: 1*
- Outcome: 99.99% Uptime
  - Recommendation: Shed load using background job queues, either automatically or manually *Cost: 0 Benefit: 2*
  - Recommendation: Merge the Schema Cache PR, aim to cut p95 request queue time *Cost: 0 Benefit: 1*
  - Recommendation: Autoscale Sidekiq based on queue latency *Cost: 2 Benefit: 3*
  - Recommendation: Move to SLA-based queues *Cost: 3 Benefit: 4*
  - Recommendation: Set SQL database pool based on thread concurrency, controlled by a single, consistent ENV variable *Cost: 1 Benefit: 1*
  - Recommendation: Break up or iterable-ize your 10 longest running Sidekiq jobs. *Cost: 2 Benefit: 2*
  - Recommendation: Move to BigRails connection management for Redis, cleanup what's stored in which Redis database. *Cost: 3 Benefit: 2*
- Outcome: Improve page load time

  - Recommendation: Reduce JS bundle size *Cost: 2 Benefit: 2*
  - Recommendation: Pick a way forward with frontend: SPA or Turbo. *Cost: 5 Benefit: 5*
  - Recommendation: When viewing registry as a guest, fetchReservedRegItems should only fire once and it should not wait on page readiness. *Cost: 1 Benefit: 1*
  - Recommendation: Create a seed or "registry pull" process *Cost: 3 Benefit: 3*
  - Recommendation: Profile allocations in V2::RegistryController *Cost: 2 Benefit: 1*
  - Recommendation: Optimize 4 main web transactions. *Cost: 3 Benefit: 2*
  - Recommendation: Profile allocations in V3::RegItemsController#index *Cost: 3 Benefit: 1*

- Outcome: Reduce infrastructure spend by $299,000/year
  - Recommendation: Reduce memcached to 1 r7g.large node. ($16k/year) *Cost: 0 Benefit: 2*
  - Recommendation: Set Puma to 10 workers on 8-core pods with 32 GB requests. *Cost: 0 Benefit: 2*
  - Recommendation: Reconfigure the fleet to use m7i.2xlarge. *Cost: 0 Benefit: 1*
  - Recommendation: Downsize the main database to r7g.8xlarge. Move read replication further down the backlog. (18k/year) *Cost: 1 Benefit: 2*
  - Recommendation: Autoscale Puma based on a combination of request queue latency and utilization/busyness. ($240k/yr) *Cost: 2 Benefit: 3*
  - Recommendation: Set Sidekiq concurrency to 10 everywhere, CPU limit to 1, and memory limit to 4GB. ($10k/year) *Cost: 0 Benefit: 1*
  - Recommendation: Multithread Puma to remove 15% of your web fleet. ($2,000/yr) *Cost: 0 Benefit: 0*
  - Recommendation: Slash Redis database sizes ($5k/yr) *Cost: 1 Benefit: 1*
  - Recommendation: Remove the Sidekiq high-memory deployment by auditing Sidekiq memory use *Cost: 2 Benefit: 1*

- Outcome: Reduce build times to less than 10 minutes.
  - Recommendation: Keep Cranking That Parallelism, Baby *Cost: N/A Benefit: N/A*
  - Recommendation: Get Journey tests off the critical path *Cost: N/A Benefit: N/A*
  - Recommendation: Consider self-hosting *Cost: 3 Benefit: 2*

- Outcome: Successful iterable integration
  - Recommendation: Treat this as a microservice, run on its own fleet

of 40 Puma workers, push work to Sidekiq and batch/flush writes. *Cost: N/A Benefit: N/A*