Hey Sergio and Beyond team,

It's been nearly a year since we started working together this time around, and my audit from last year is therefore about nearly as old. I thought, heading into 2026, that it would be good to take a step back and reassess Glue holistically. Priorities, of course, change, and you've implemented a number of things from last year's report which means it's less useful now.

In terms of progress since last year, here's the main things from last year's audit that were implemented:

- Increased observability of Salesforce queries which are slow
- Migration to Datadog (still in progress but certainly moreso than January)
- Some I/O operations now run in parallel (see: dashboard documents and Async use)
- Reporting request queue time
- We have a decent "Speedshop dashboard" in place
- Salesforce queries now have attached team ownership metadata

Overall latency on `glue` web hasn't gotten significantly better or worse since January, overall latencies look basically the same.

It feels to me like a significant amount of breathing room has been achieved on the Salesforce integration, and we're now looking towards reducing overall user-facing latency.

After looking through the app with new eyes, I have two key metrics to track for 2026:

1. **99.9% latency SLO on read/write latency on primary**. A key part of both glue's stability and latency that's lacking is consistent database performance. This is a major, dangerous footgun that must be addressed.
2. **No glue controllers >2 sec >25% of the time**. Currently, there are 3 controllers which take more than 2 seconds more than 25% of the time. That means that, up to 75% of them, we're delivering a pretty painful latency experience.

I'm going to structure my recommendations into two sections: removing footguns, and reducing latency.

Footgun Removal

Recommendation: Create database read/write latency SLOs and remove overnight database pressure

This issue was initially identified in the September report. It's been ongoing since then, so it's clearly not having any customer impact… yet.

A *daily* database performance brownout of about 7 hours is a sign of a system which is teetering on the brink. Add a little more load here or there and you can easily get a major customer incident.

First off, I will install two monitors and SLOs, one for read latency and one for write latency, and I'd like to get those to 99.9%. I'm going to set the latency target at whatever the current "normal" level is during the day, which appears to be about 1 millisecond for reads and 5 or 10ms for writes.

The root cause here is a smearing strategy being applied to within_12_hours jobs, which causes work to be executed at an arbitrarily high concurrency that's outpacing the database.

You have three options as I see it:

1. Upgrade the database to provision more IOPs, because you are being throttled by IOPs here.
2. Reduce concurrency on the within_12_hours queue.
3. Fix the database queries happening in the jobs which execute during this time so they stop doing so many sequential reads and causing so much I/O pressure.

I would like to do a combination of all three.

Recommendation: Dedicate even fewer minimum resources to within_12_hours and within_24_hours queues, and scale to zero for ~4 hours or more.

Queue time performance on of your "very long SLO" queues continues to be far better than required. These 12+ hour queues mostly sit at latencies of less than a minute.

The purpose of high SLO queues is to reduce resource usage by increasing utilization. The processes pulling from these queues should be either "off" (scaled to zero) or utilized at 100% (on). Currently, you're not getting this benefit.

I would also encourage you to regularly (perhaps daily on a schedule?) scale these queues down to zero. That can be automatically (based on queue time autoscaling, below) or scheduled (say for 4 hours in the middle of business hours, or over a weekend at some time). It's useful to regularly "test" your assumptions about the acceptable latency of jobs in these queues by *actually making them wait that long.* Currently your request queue times on these queues are so low that I worry that there are jobs in here which actually should be in 5 minutes or faster queues, but they never notice because latency in practice never exceeds a minute.

Recommendation: Move to queue time autoscaling for background jobs, report to Cloudwatch

Your load is so predictable that Sidekiq SLO performance has not been a meaningful issue. Your SLO performance for the last 30 days looks great.

However, not having autoscaling based on queue times, to me, is an incident waiting to happen. Eventually the queue latency will balloon overnight when no one's awake.

The remediation here of reporting to Cloudwatch and setting up a scaling policy based on that is a pretty simple task.

For web, I'm less bothered, because your web load is even more predictable than background jobs. With jobs, you theoretically always have the potential problem of a change in data or input or a new job type being deployed that 100xs your load, but that threat is far less present on the web side.

Recommendation: Glue's Redis connection pool creation is not thread safe

This is a quick one, I'll just open a PR to this effect:

Storing the connection pool singleton in a class instance variable which is instantiated by `||=` is potentially thread unsafe. You could end up with two threads with different connection pools. This isn't a huge deal but I would like to clean this up by converting this to a constant which is set using `=` to avoid any issues.

Recommendation: Reduce Salesforce Load: Optimize OpportunitiesController#create

You have, effectively just a few major sources of time spent waiting on Salesforce. Since Salesforce is such a huge part of your reliability and uptime, it would be useful to make sure that our biggest sources of time spent waiting on Salesforce are absolutely optimized.

Every day, you spend about ~1.5 days of wall clock time waiting on Salesforce. OpportunitiesController#create accounts for 6% of that. This controller calls Salesforce 70 to 75 times, on average. I have to believe that this can be reduced.

Recommendation: Reduce Salesforce Load: Optimize SettleTradelineJob

Accounting for about 4% of total Salesforce load, this job suffers from a loop which is repeated anywhere from 1 to a couple dozen times:

1. Request to cftpay.com to v1/creditor/id/?/externalAccount
2. Salesforce GET to the Transaction table
3. Salesforce GET to the CreditorPayee table

Since these are GETs, I have to imagine we can de-N+1 the salesforce queries here and do them in a single query instead.

Recommendation: Reduce Salesforce Load: Optimize ContinueProgramEnrollmentJob

Also accounting for about 4% of total Salesforce load, this job runs about 150 to 200 requests to Salesforce on every execution.

There's a significant number of PATCH requests here as well.

As an example, in ContinueProgramEnrollmentService, we actually make two separate update calls in the same method:

```
if @program.attorney_approval_status == Af::Program::AttorneyApprovalStatus::PENDING
@program.update!(
    attorney_approval_status: Af::Program::AttorneyApprovalStatus::APPROVED,
    attorney_approval_status_date: Date.current
)
end

#
@program.update!(
contract_executed_date_time: contract_executed_date_time,
client_signed_date_time: client_signed_date_time,
co_client_signed_date_time: co_client_signed_date_time
)
```

If there's stuff here that's *that* easy to fix, I'm sure we could take a few % points of total Salesforce usage just by combining some calls here.

It seems a frequent pattern in Beyond's Salesforce services that operations are written for a *single* instance of a resource and naively looped over. Instead, you need services which operate over *several* instances and will batch calls where applicable.

Improving Web Latency

Overall latency on the web side hasn't really changed in either direction this year. As you already know, latency here is driven by the Salesforce integration.

We fundamentally can't make Salesforce any faster than it already is. We have two options:

1. Call it less.
2. Call it in parallel.

I would like to do more of both.

Recommendation: Establish a global maximum Salesforce query count in tests, for controllers and jobs

I've been seeing a lot of success with other companies employing prosopite as a kind of semi-automated workflow for reducing N+1s in Rails apps.

The workflow is:

1. Add Prosopite to the test-suite, default "on" so that tests fail if Prosopite detects an N+1.

2. Add skips/whitelists for failures for everything in the test suite today.
3. Slowly burn down the catalog of the whitelist, until the entire test suite is passing with no skips and exceptions.
4. Turn it on in development mode to also raise as the dev is working.

This approach of "whitelist all current violations but forbid future violations, then burndown" is a great strategy that I've found works well for mitigating this technical debt.

However, we don't have the ability to parse Salesforce queries off-the-shelf in the same way Prosopite does. That could be pretty hard. Also, even if we did, it isn't really a guarantee that it's going to be very helpful, as particularly on the web side I think you've got a lot of Salesforce API usage that *isn't* reducible to N+1. And if you go to Salesforce 100 times and none of them are N+1s, well, it's still gonna be really slow!

I think what *might* work is a global call count limit. Wrap every controller and every job in a middleware that asserts that the number of times Salesforce is reached out to during this job/controller cannot be more than X. Ideally, I think to meet a latency goal of 1 second or so, that number is probably about 30 to 40.

I'm sure you have many, many specs that today would violate this. You also might have to set the threshold even lower because your test data is likely not as complex as your production data and the number of Salesforce calls is likely lower overall in tests. But an atmosphere where:

1. Everyone understands that the Salesforce API is to be used as little as possible (in terms of call count)
2. There has to be a discussion to add a *new* codepath to the violation "whitelist"

… is one that would definitely get these call counts under control.

Your current approach of asserting Salesforce counts around specific operations isn't enough because services can be combined arbitrarily or called in loops. Only a global limit around the entire transaction can guarantee that we're actually not doing anything crazy under the hood.

Recommendation: Create a general-purpose parallel-capable data repository and template

The next step is providing people with a mechanism to parallelize their Salesforce API usage.

1. We can't make a single Salesforce API call faster than ~50-100ms
2. There will, inevitably, be some endpoints where you cannot reduce the call count to Salesforce easily.

I felt that in 2025 we started down this path a bit with our work to parallelize document downloads, but it wasn't taken much further. I also said something similar to what I'm about to propose in the 2025 report. But, the idea has crystallized a bit better for me, and now that we've got a bit of experience using `Async` and it's APIs in production I feel a bit more clear about the path forward here.

Any transaction (job or web) makes N number of requests to Salesforce. Today, those requests happen strictly sequentially, and strictly in the same order.

However, think about a controller as a black box:

```
In: The rack environment, controller parameters

~~ MYSTERIOUS BLACK BOX OF SALESFORCE INTERACTIONS ~~

Out: A rack response [200,{},some_json_object]
```

In order to generate the JSON object, we have a list of ~100 Salesforce queries we have to make.

Those queries all have a relationship with each other. Currently, since they execute linearly with no parallelism, that relationship looks like:

```
Query 1 → Query 2 → Query 3 → ... → Query N → JSON Object
```

However, if you actually thought about the relationships of these queries to each other, they form a directed acyclic graph which is not at all linear:

```
           Query 1 (Program ID)
          /        |         \
    Query 2    Query 3     Query 4
   (Client)   (Tradelines) (Attorney)
       |          |           |
    Query 5    Query 6     Query 7
   (Address)  (Creditors)  (Fees)
       |          |           |
    Query 8    Query 9     Query 10
  (History)   (Payments)   (Status)
        \         |          /
         \     Query 11     /
          \   (Balance)    /
           \      |       /
            \     |      /
             \    |     /
              JSON Object
```

What you're currently missing is twofold:

1. A clear division between the data-gathering and the serialization of that data into a template. Currently, you've got a mishmash of service objects

with several responsibilities, combining templating and data gathering in one.
2. An easy way to convert a DAG (directed acyclic graph) of Salesforce requests into a repository of data which can be passed to the templating step.

If we could make the DAG something you could actually enumerate in code, we could build a Salesforce data loading abstraction that parallelized automatically. In the example above, we could determine that Query 1 must be done, then query 2-5-8, query 3-6-9-11, and queries 4-7-10 could be done in parallel.

There would need to be several types of objects involved:

1. A Query. This is a function which takes inputs and has outputs. It can be combined with other queries, or it may have no inputs at all and therefore have no relationships in the graph. These might return Promises (like GraphQL's dataloader) or something else entirely (I'd have to look at `conurrent-ruby` and `Async` in more detail).
2. A Repository. Calling `repository.load` would cause the entire graph to "run", and it would output a single hash of data which can then be passed to a templating or serialization layer (so, after the repository is done loading, you can ban all further requests to salesforce).

This would be a pretty major migration. I think I'm also not able to clearly articulate this, and I will need to ship a first version of the code for the abstraction to be clear.

My plan would look like this:

1. Nate ships a demonstration of the approach.
2. We refine and talk about the code, finalize the abstraction.
3. An internal champion is chosen to head up the migration.
4. We decide how far to take this: enforce on 100% of all transactions? Or just use as a tool when call counts get too high?
5. We migrate endpoints one at a time, starting with the three worst endpoints in Glue by latency (listed below).

It would probably require a significant amount of investment in 2026. It depends on how much latency improvement is important to you. Parallelizing queries of course does not reduce total usage of Salesforce, it only makes the transaction itself have lower latency.

Recommendation: Improve latency of Opportunities::pullcreditcontroller#create

This controller responds in >2 seconds about 75% of the time, usually with around 75 to 100 Salesforce calls. It frequently takes more than 5 seconds to respond.

Recommendation: Improve latency of Onlinecreditpullscontroller#create

70% of requests to this controller take more than 2 seconds to complete. With about ~75 calls to Salesforce to complete, this controller generally takes around 2 seconds.

Recommendation: Improve latency of Offerscontroller#approve_by_client

This controller responds in 2 seconds or more about 60% of the time.

In many ways, this controller looks very similar to OnlineCreditPullsController#create. It generally takes about 2 seconds to complete. However, it only has about 10-20 Salesforce calls. Because of this, it might be the easiest of the three controllers here to unwind and fix.