CompanyCam Initial Plan

Key Metrics

Here is my suggestion for an actionable and measurable set of goals, based on the priorities you have explained to me.

1. **Quality of Service**: 99.9% SLO uptime on web request queue time of p95 10ms (100ms for core and any other customer-facing services), 99.9% SLO uptime of all queues in Sidekiq.
2. **Overprovisioning**: Reduce vCPU and memory allocation on Fargate by 25% for web, and by 50% for Sidekiq (roughly: by average of 75 cores/month for Web, 150 cores/month for Sidekiq) to save roughly $7,000/month
3. **Uptime**: 99.99% uptime based (ping) SLO for web services.

Quality of Service

Here are the details of how to accomplish this goal.

Track request queue time for web in Datadog

This is our main quality of service measure on web.

In my mind, the way that end operators like yourself think about scaling is "I will pay money to keep my response times between 100-110% of what they are when my app has no load".

Response times get slower for 2 main reasons under load:

1. Time spent queueing increases. You are not measuring this today. This is how long a request spends waiting for a Puma process to start working on it. It primarily consists of time spent waiting in the socket because there are no Puma processes on the machine currently calling `accept` (because they are all processing requests instead).
2. Time spent actually running the request increases, because CPU contention is increasing. This is basically another queue: the queue of CPU tasks waiting for an empty CPU. We don't measure this directly (in time/latency as we do with requests) but indirectly via CPU utilization and load. We try to keep CPU utilization below about 80% and load below 80% of the core count allocated to us. If we can do that, requests generally don't get more than 5-10% slower under load.

So, we don't have any measurement at all for point 1. We need that to make more intelligent decisions about how many tasks we need. Without measuring it, we could be inflicting really bad conditions on our users without knowing!

There's two steps for this:

1. We need to stamp a header on every incoming request with the time since the epoch in milliseconds.
2. Use the Datadog library to read that header and compare it with the current time, reporting it to Datadog.

You're currently running ECS tasks. We cannot add this header on an Amazon ALB. Most people in your situation add NGINX to the webapp ECS task and then have an NGINX config like the following:

AI generated text starts here

```
server {
    # ... existing server config ...

    # Add X-Request-Start header with millisecond precision
    add_header X-Request-Start $msec;

    # ... rest of server config ...
}
```

AI generated text completed

Then, on the Datadog side we change our initializer:

```ruby
# config/initializers/datadog.rb
Datadog.configure do |c|
  c.tracing.instrument :rails, request_queueing: true
end
```

Here's what this will allow us to do in Datadog:

1. We now get request queue time added to each individual companycam trace. This is neat because you can see "not only was this request slow, it also queued for 5 seconds!"
2. We can create monitors and SLOS around this condition.

I would like request queue time to be our primary quality of service measure for web.

Hunt down and quarantine or fix jobs which take longer than 30 seconds

You have a handful of jobs which take longer than 30 seconds a decent amount of the time. There are several problems with this:

1. **It causes unpredictable queue behavior**. Queue times are more consistent and predictable when jobs are frequent or short. You will violate your SLOs less if you have 10x the jobs but each job is 1/10th the duration.
2. **It causes idempotency bugs**. While IN THEORY every job should be idempotent because you're all very good programmers, in practice we

forget. Sidekiq's job shutdown timeout is 25 seconds, which means that jobs which reliably execute >= 25 sec will eventually be hard-stopped halfway through execution, possibly causing bugs at that point or when they are eventually retried.
3. **Long running jobs are a frequent source of memory pressure**. Frequently, long running jobs also end up iterating over large collections and causing lots of memory to be used at once.

I would like to inventory and fix each of these jobs to remove these pressures.

Implement queue sharding

You are having issues around "fairness" in job scheduling.

The best way to solve this is queue sharding.

I'd like to talk with you a bit more to understand *what we should shard on.* At previous engagements we've sharded on job class, but I think in your case it may make more sense to shard by tenant.

The end state here will be that important queues will have multiple copies (within_30_seconds_0, _1 and so on) so that a single tenant or job class can only "take down" X% of total capacity in the worst case.

Change how_long and cooldown, which are probably too slow for scaling up

Your infrastructure currently has some pretty conservative settings for scaling Sidekiq in the up direction. This will eventually impact quality of service by causing short and frequent SLO violations, where the autoscaler is either not re-evaluating or not scaling up enough to respond.

I'd like to work with you to understand why these values are currently set to what they are, and potentially make these values much more aggressive.

New Sidekiq SLO/monitor suite

We discussed the current Sidekiq SLOs. I find the metrics they're working on a little hard to understand. Instead of thinking in terms of:

```
Number of jobs which executed within the queue time SLO

------
Number of jobs executed
```

. . . it's easier (and pretty similar) to track:

```
Amount of time maximum reported queue latency was within the queue time SLO

------
Time
```

That's not *exactly* the same thing, but it's close enough, and much easier to track and understand in Datadog.

I'd like to set up a new suite of monitors and SLOs, provisioned in terraform, based on this concept. At a minimum we'll do this for each latency/SLO queue, but hopefully also for the other handful you have as well (depends on whether or not those queues truly have SLOs. They may not.)

Full Speedshop dashboard including CPU/Mem %s on containers, carried traffics

We have a "standard" dashboard with a set of 25 or so indicators that I think would be very useful to you:

- 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.
  - % of responses which took longer than 500ms, organized by controller action.
- Scalability
  - Web utilization
    - Total Puma process count
    - Concurrent request load (aka carried traffic) (average req/sec * sec/req)
    - Process count / load
    - CPU/mem % for Tasks
  - Task counts (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
    - Hitrate (if cache)
  - Error rates

    - ∗ Web, worker
  - www.*.com uptime

Speedshop will work to put this into place on your Datadog account. We may need action from you to capture the correct data/get it into Datadog.

Overprovisioning

Here are the details of how to accomplish this goal.

Fix issue causing sawtooth pattern in scaling on Sidekiq queues around deploys.

This is probably the biggest cause behind your overprovisioning problems at the moment, and why I think you can remove significant CPU/memory allocations from Sidekiq.

I do not understand why this is happening, but on every deploy your Sidekiq task counts get scaled to max and then slowly get autoscaled back down as time goes on. This is not necessary, so there's some bug in what's going on here.

I took a quick look at the autoscaling policy and I don't see an obvious reason for this behavior to occur.

We may need to get on a call together to debug.

Scale the 300 second queue less aggressively so that it better matches the target latency of that queue

The 300 second queue's task count is too high for what the queue's average job concurrency is. Higher latency queues, like this one, should have generally low task counts (lower than the lower latency queues), because their main purpose is to run at higher utilization and higher latency! If we're not getting better utilization out of them, they don't really serve a useful purpose.

This needs to be addressed *after* the sawtooth-on-deploy issue is solved, as it's hard to reccommend new autoscaling policy settings until that "noise" is removed from the graph.

Scale web based on a combination of request queue time and utilization

Currently, the web service is scaled based on a Utilization metric.

Utilization is a good thing to control and track. It helps us make sure we're "getting what we pay for" by making it very clear what percent of our capacity is idle and what is used.

However, it is actually a second order metric to what we care about: quality of service. We care about how long users spend waiting to be serviced, just as you would if you were running a grocery store and deciding how many checkout clerks to hire.

I haven't done this on ECS before (I have in KEDA/k8s), but I believe it's possible to run 2 auto scaling policies in ECS. I'd like to *keep* the Utilization policy (we may tune the value its set to) and *add* a request queue scaling policy.

This is an add on to the request queue time measurement discussed previously. Using that same header, you will have a custom middleware to report request queue time to Cloudwatch, and then add an additional auto scaling policy to scale based on that. We'll set the value to scale on based on what we decide re: a request queue time SLO.

Resize web pods

This was discussed in Slack. You can get "more throughput per dollar" with bigger tasks.

Resize Sidekiq pods

Sidekiq, when it's not running in Swarm mode, cannot meaningfully use more than 1 vCPU at a time. You have several ECS tasks for Sidekiq which have > 1024 CPU shares allocated. This isn't a very effective allocation of resources. I would keep all Sidekiq CPU allocations at or below 1 core.

Install GVL metrics and change threadcounts based on it

We have a GVL metrics gem that captures how long threads spend waiting to acquire the GVL during a request. We'll pass you the gem installation instructions and you can get this reported to Datadog.

Once we have that information, Speedshop will be able to make more intelligent recommendations for threadpool sizes for Puma, and for each individual Sidekiq queue.

Eventually, we will be shipping an auto-tuner here that you'll have access to which makes these decisions without human intervention and on-the-fly in response to changes in workload.

Latency buckets aren't the ones I would have chosen

It's possible there's something I'm missing here in terms of your domain/your requirements, but your latency buckets on your SLO queues are not the ones I would have chosen.

You have 10, 30 and 300 seconds queues.

For me, 10 and 30 seconds are too similar. Are there really that many jobs for which that's a meaningful difference? "I can execute within 10 seconds, but 30 seconds is too long?"

It's also missing higher-latency queues. Businesses usually have jobs which can be executed with 1 hour or 24 hours.

And I think it's also missing an "ASAP" or "within 0 seconds" queue. Most businesses seem to have workloads which need to be run on Sidekiq, but which have latency requirements similiar to web, because basically "a real live human being is waiting for this job to complete". A password reset email is a good example here.

I think we need a 15-30 minute discussion on these buckets, why they are what they are today, and what they might be in the future.

Uptime

Here are the details of how to accomplish this goal.

Implement Datadog Database monitoring

I'm a massive fan of Datadog's database monitoring product. It brings all the power of AWS performance insights into the "walled garden" of Datadog. I highly recommend installing it.

Move to r8g

I noticed you're on the second-to-largest r6g RDS plan. You're already (rightly) thinking about buying more database headroom by adding a read replica. However, I think you can also increase your vertical scaling runway here.

There's a meaningful different in read times in the r6g generation versus r8g/m7i. Here's a good benchmark result. Keep in mind that the Graviton2 (the processor r6g runs on) is about 5 years old now, and ARM architcture has progressed a lot in the time since then.

In addition, there are larger plans available on r8g:

AI generated summary below, but I checked the figures

Instance Type	vCPUs	Memory (GiB)	Network Bandwidth (Gbps)	
db.r6g.16xlarge	64	512	Up to 25	
db.r6g.12xlarge	48	384	Up to 20	
db.r6g.8xlarge	32	256	Up to 12	(Amazon Web Services, Inc., Amazon Web Services, Inc.)

Instance Type	vCPUs	Memory (GiB)	Network Bandwidth (Gbps)	
db.r8g.48xlarge	192	1,536	Up to 50	
db.r8g.24xlarge	96	768	Up to 40	
db.r8g.16xlarge	64	512	Up to 30	(Amazon Web Services, Inc., Amazon Web Services, Inc.)

End AI

You should vertically scale until you absolutely cannot anymore. You're reaching that point, but r8g represents a meaningful increase in breathing room that I think would be worth the pain of a maintenance downtime. Costs are not significantly different versus the r6g generation.