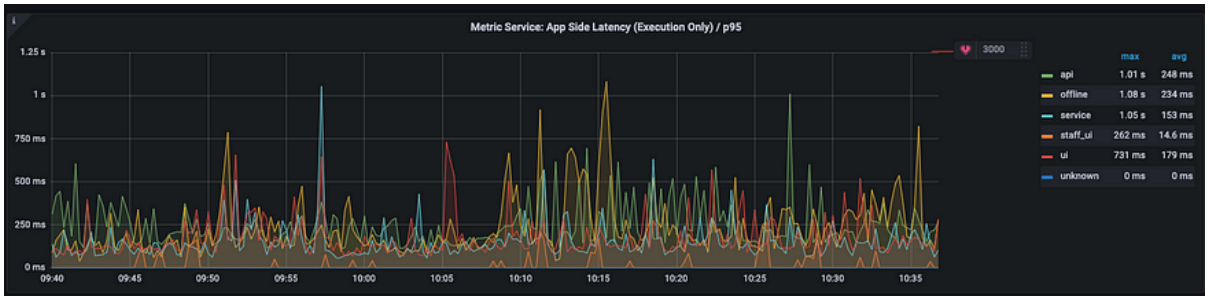
Adaptive Concurrency Control for Mixed Analytical Workloads

Author: Dan Kleiman

Claps: 223

Date: Jan 4

At any given moment, I can pop open a monitoring dashboard for our Metrics Service and see a healthy blend of traffic as we serve analytics to our customers across a variety of channels like public APIs, dashboards, and reports.

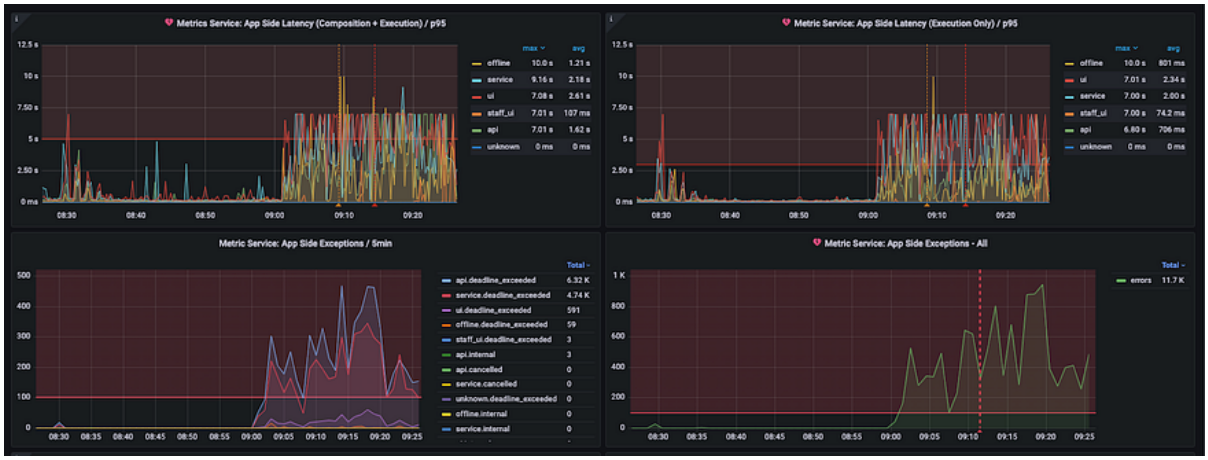


Yes, there is some variation in latency across callers â€” denoted by the prefixes in the legend like â€œapiâ€, â€œofflineâ€, etc., but we are humming along answering questions like:

- How much revenue did my last campaign generate?
- Which email domains have the best open rates?
- Which products have my customers recently purchased?

The Metrics Service is provisioned to answer thousands of requests like this per second, across time ranges that showcase a recent campaign with up-to-the-minute freshness or year-over-year reporting breakdowns across flexible dimensions and filter criteria.

Imagine the sinking feeling in your gut when you hear that klaxon blaring on your phone and open up the same dashboard to see this:



Congestion and timeouts across all callers of the Metrics Service.

In the months after we released the Metrics Service, we would hit spikes of Deadline Exceeded errors that would impact all of our callers. A caller that we could normally answer with a p95 of less than 100 ms would start hitting timeouts after 5 seconds of waiting to have their request processed.

Initially these waves of congestion were a mystery to us, but in this post, I will explain how we diagnosed the issue, reached for an open source solution by adapting a Netflix concurrency limit library, and ultimately implemented our own concurrency control layer for the Metrics Service.

Sharing a Giant Calculator

Previous versions of our analytics engine were fast, but not very flexible. If the answers to a question hadnâ€™t been anticipated and coded into the write-time aggregation engine, we couldnâ€™t answer it.

We accumulated new data pipelines and backing datastores for any new use case that didnâ€™t quite fit the existing storage and query patterns. Over the years, this became an expensive way to extend our analytics system. More datastores and materialization pipelines to fill them meant more operational complexity for our engineering teams. Eventually this approach became too expensive and too complex, so we pivoted.

Last year we introduced a more flexible read-time aggregation engine â€” the Metrics Service â€” that could answer all the same questions, but calculate the answers from raw data as the questions were asked, not as the data was ingested.

Redesigning our analytics engine around newer, faster OLAP databases has paid off. Not only were we able to save significant costs by eliminating redundant datastores, but we unlocked new analytics use cases: other engineering teams can now ask questions in ways we previously hadnâ€™t anticipated, with no additional data infrastructure needed.

Unfortunately, thereâ€™s a catch to exposing a fast, flexible read-time aggregation engine.

The Metrics Service runs a huge mix of workloads, so it is essentially a giant, shared calculator.

The Metrics Service as â€œshared calculatorâ€ between heterogeneous workloads.

We accept queries from our public APIs, charts in our app, internal services, and for offline report generation. Each of these use cases has different query complexity:

- Some queries ask for a single value, others want time series grouped by multiple dimensions.

- Some queries want an aggregate over 7 days, others want aggregates over 3 years.
- Some queries want high level roll-ups, others have very specific, targeted filters.

The throughput from each use case can vary wildly too. Our UIs may make a dozen requests per second as people analyze charts in real time, but our public APIs can serve thousands of requests per second.

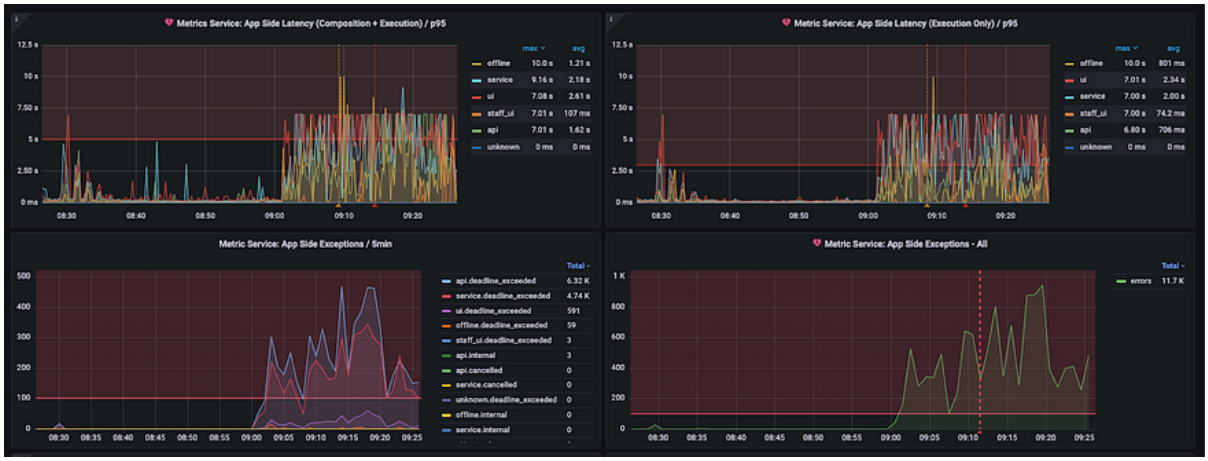
All of this complexity and throughput flows through to the same service. Managing the health and capacity of the service in the face of constantly shifting query complexity, throughput, and mixed workloads became our next big challenge.

How do you balance these two competing considerations?

1. Everyone deserves fair access to the â€œshared calculator.â€
2. Everyone becomes a potential DOS attacker of the â€œshared calculator.â€

In other words, if any one user starts sending too many requests or issuing queries that are too complex, they can steal resources unfairly from other users that need access to the service, causing query processing to back up and requests to timeout.

Imagine thinking â€œmy workload hasnâ€™t changed, why are my metrics suddenly unavailable?â€ That was what we started running into.



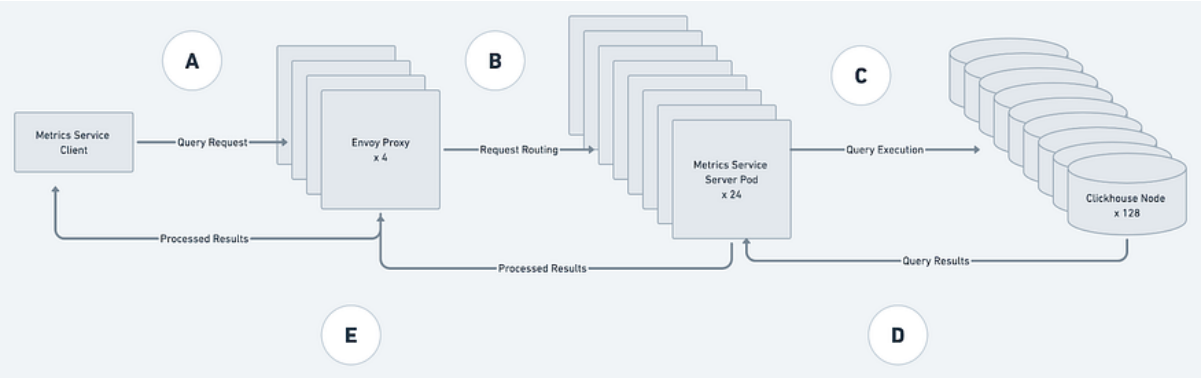
Congestion from heavy â€œapiâ€ traffic impacting other users of the service, even though their call patterns havenâ€™t changed.

As we investigated the bottlenecks in our service, it became clear that our waves of timeouts were caused by attempting to process too many requests at once.

Enter **concurrency control** â€” a strategy for limiting the amount of work in flight at any one time.

Inspired by Jon Mooreâ€™s 2017 Strangeloop talk â€œStop Rate Limiting! Capacity Management Done Rightâ€, Netflixâ€™s blog post Performance Under Load, and their Concurrency Limits library, we set out to apply the principles of concurrency control to the Metrics Service.

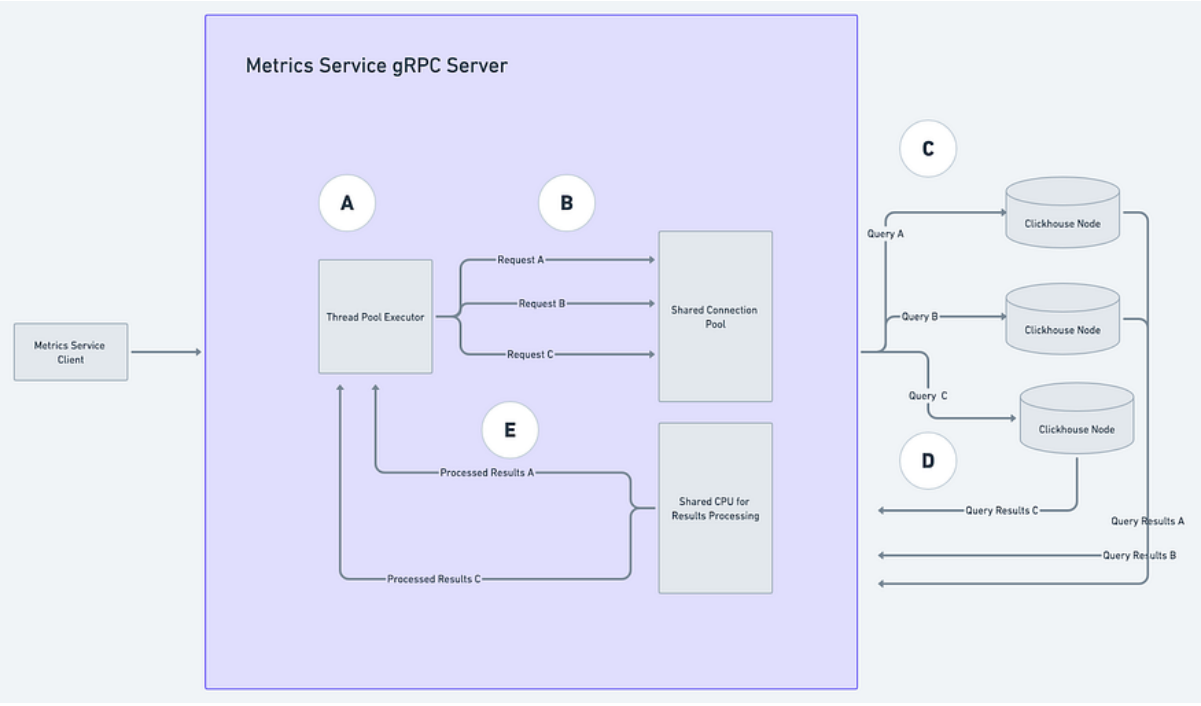
Finding Bottlenecks in Request Flow



Our basic request flow looks like this:

- Callers compose a query and issue a request via our gRPC client with a specific deadline for processing. (A)
- We route all requests through an Envoy proxy layer that then forwards traffic to our gRPC server pods. (B)
- Server pods turn the gRPC request into a SQL query that is executed against our Clickhouse cluster. (C)
- Server pods process the raw results from the database and compose a response. (D)
- Which is then routed back to the original caller. (E)

Most of these components are lightweight, like request forwarding in envoy, but there were places where we saw highly variable processing times.



In the diagram above, weâ€™re focusing on what happens inside the gRPC server pods.

- Once the gRPC server receives a request, it submits the request to a ThreadPoolExecutor that manages query execution across multiple threads. (A)
- Each thread grabs a database connection from a shared connection pool. (B)

- Queries are issued against the database cluster once the thread grabs a connection successfully, but queries can take a variable amount of time to execute at the database level. (C)
- Results come back from the database. There is high variability here in terms of the number of rows returned. (D)
- Finally, processed result sets are shipped back to the callers. (E)

There are several places where we see contention in this request model:

1. We are limited to how much work we can process in parallel by the number of threads in the thread pool.
2. The working threads share a fixed number of connections in the database connection pool.
3. The database itself can only process so many queries in parallel.
4. Large result sets are more expensive to process than smaller ones and this happens across shared CPU resources on the server.

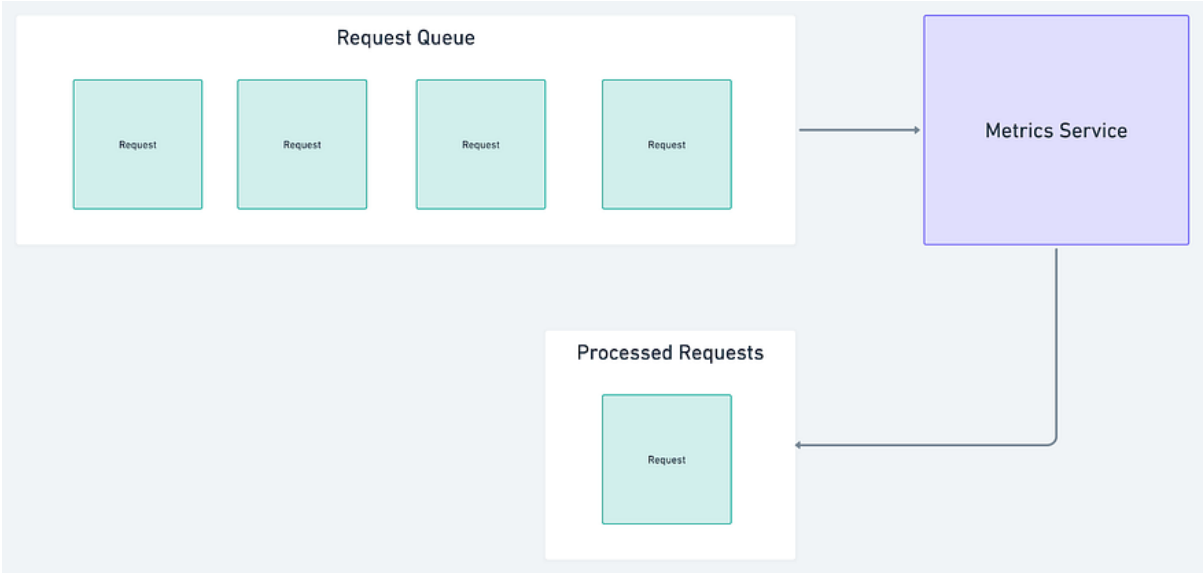
We poked and prodded at each step of this request flow to try to understand congestion and latency and there is nuance at each step.

It turned out, though, that we could actually take a much more simplified view to understand the essence of the problem.

Request Queueing

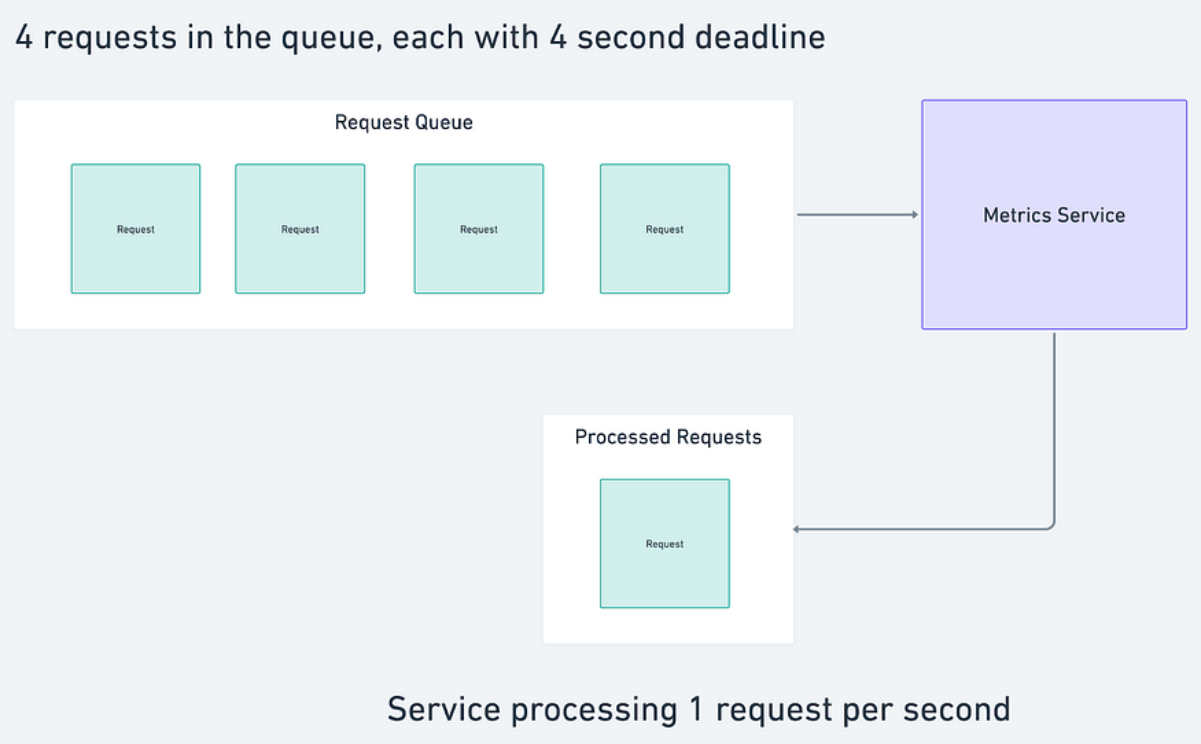
If you treat the downstream steps in the request flow as highly variable and essentially unpredictable, queueing theory can show us the difference between a healthy and unhealthy service.

First, the healthy state where requests are processed in time, as they come in:



As long as the processing rate of the Metrics Service is balanced with the input rate of incoming requests, we are in a healthy state.

If we add some made up timing and numbers to the diagram above, you can see how queue depth and processing rate interact:

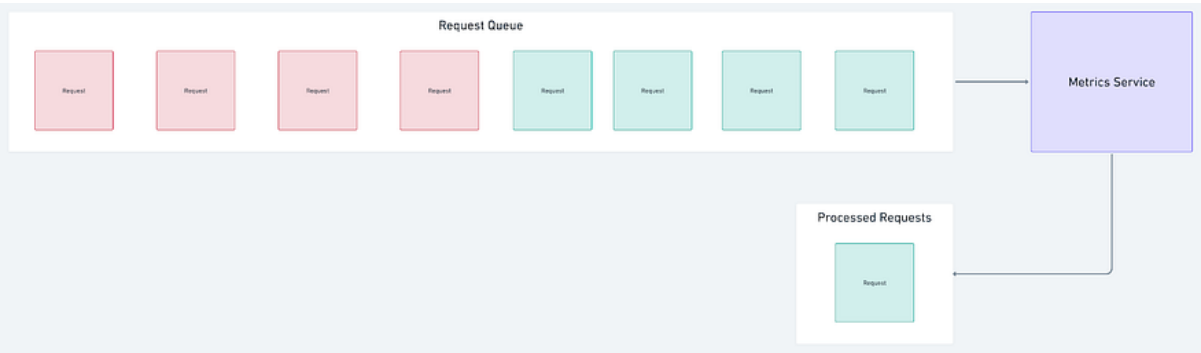


If we have 4 requests in the queue and weâ€™re processing 1 request per second, then all 4 of those requests will be processed within their deadline.

The first request will be processed after 1 second, with 3 seconds of deadline â€œbudgetâ€ left over. The second request will be processed after 2 seconds, with 1 second of wait time and 1 second of processing time. The third request will wait 2 seconds and again take 1 second to process. Finally, the fourth request will be processed just in time, after a 3 second wait.

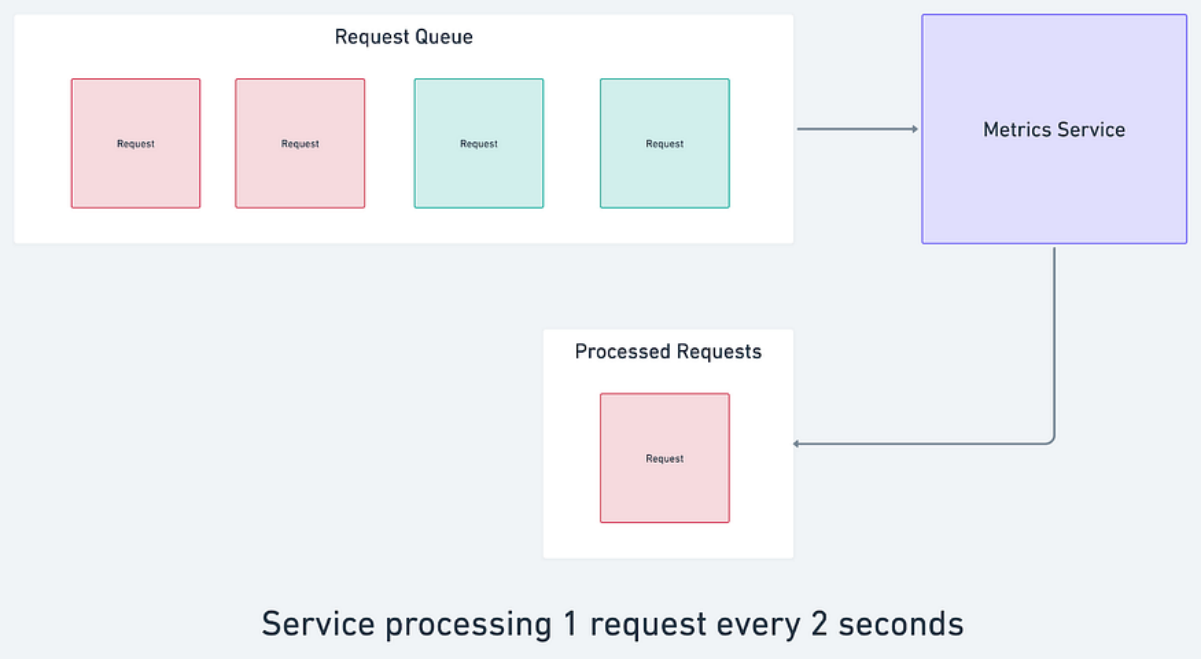
In other words, given a processing rate of 1 request per second and a client timeout of 4 seconds, the maximum healthy queue depth is 4.

Any additional requests that get enqueued â€” beyond the 4 we can process with this processing rate and timeout configuration â€” will time out while they are waiting in the queue.



The fifth request will have 5 seconds of wait time, exceeding its deadline of 4 seconds and so on as the queue grows beyond the service capacity.

Alternatively, If our processing rate slows down, the same thing can happen from the perspective of new requests entering the queue that we would have served successfully under different conditions:



Slower processing time means that we cannot get through the queue as fast as before. Again, the caller experiences this as a timeout.

Recalculation of queue depth with our reduced processing rate: given a processing rate of 0.5 requests per second and a client timeout of 4 seconds, the maximum healthy queue depth is 2.

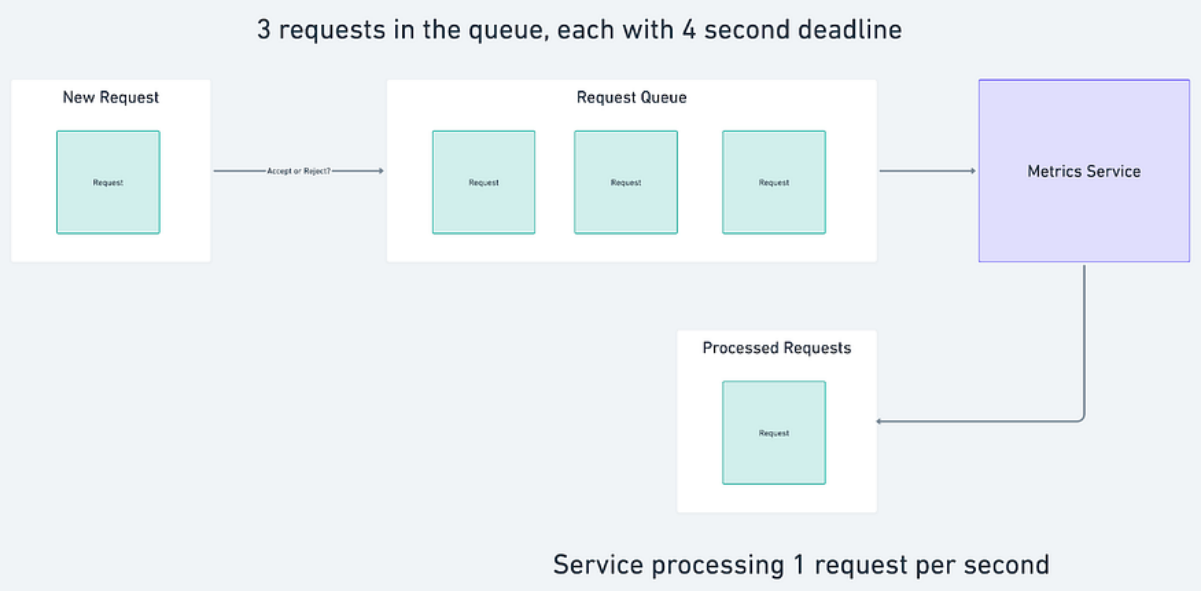
Now, when you look at the â€œwave of timeoutsâ€ charts above, you start to see a queue management problem â€” weâ€™re trying to work on too many requests at once.

We were really close to a solution, we just needed to figure out a way to manage queue depth. And if you think about â€œmanaging queue depthâ€ as â€œshould I add this request to the queue or not?â€ you come to the next core concept: load shedding.

Load Shedding for Managing Queues

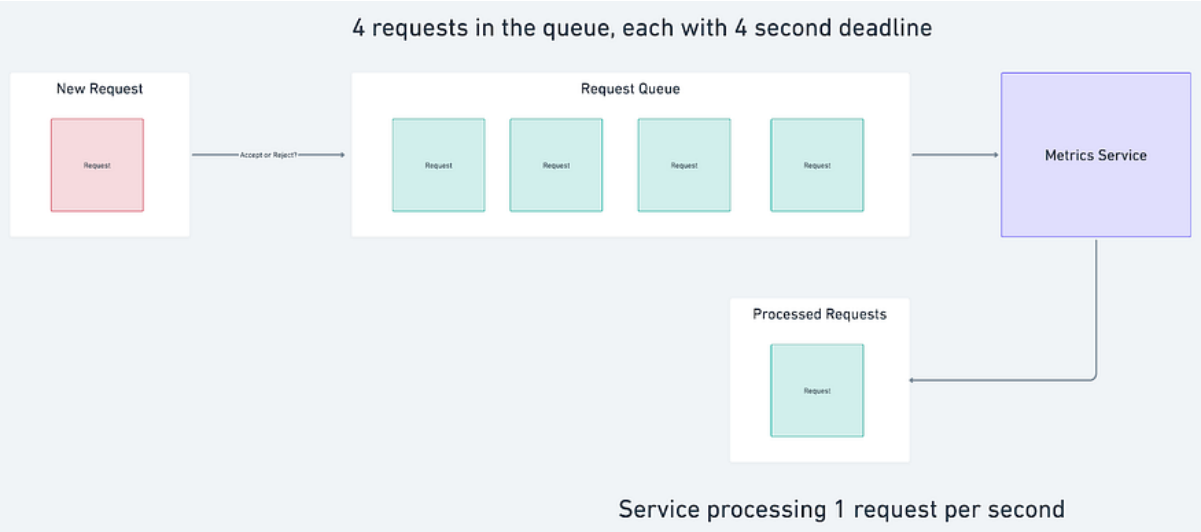
The core question when we set up a system to shed load is: at this moment, do I have the capacity to process this request in the time allotted?

If the answer is yes, then you should accept the request:



The new request will be processed within its deadline, so you should accept it.

However, you should reject a new request in this scenario:



Given the current queue depth and processing rate, we know the new request will time out waiting in the queue, before we can process it, so we should reject it.

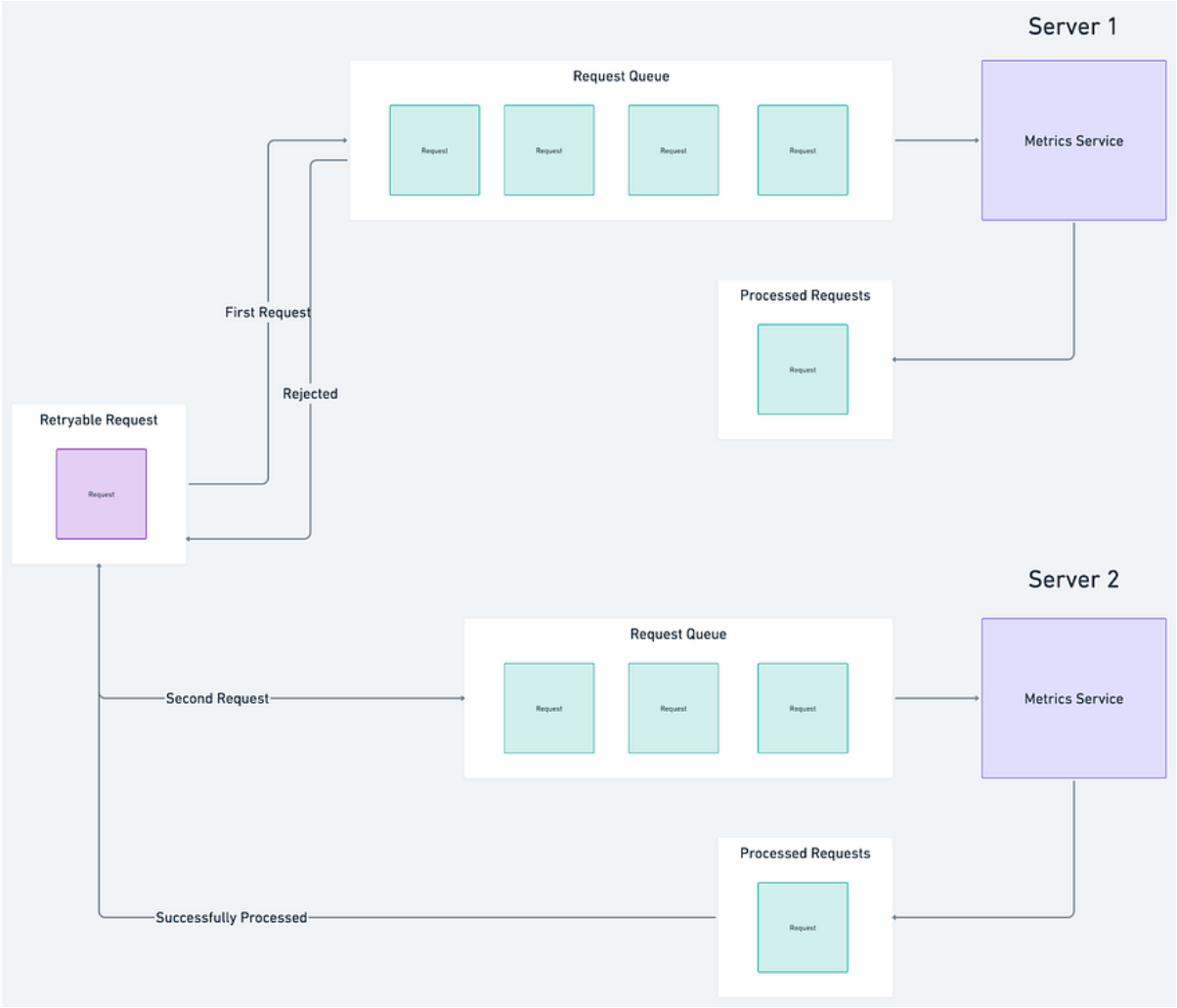
And rejection is actually better for the system overall!

If we donâ€™t reject this request, we now know it will time out waiting in our queue. Weâ€™ve wasted the callerâ€™s entire deadline budget waiting, instead of working. Imagine they are fetching this data for a chart on a dashboard. The UI is just spinning, waiting for data.

Alternatively, if we reject right away:

1. The caller still has time within their SLO â€” we didnâ€™t waste their whole deadline budget.
2. The rejection is retryable, since weâ€™re saying â€œwe know we canâ€™t process it right here at this moment, please try again.â€

In our experience, queue backups mostly happened on individual servers due to the unique mix of query complexity landing on that server at that time. The issue is local congestion, not a problem with the whole cluster of servers.



By rejecting traffic quickly with retryable exceptions, a second request would likely succeed on a different server.

So how do we decide to accept or reject an incoming request?

To Accept or Reject?

Here, Iâ€™ll lean on Netflix to explain the accept-or-reject decision, based on Littleâ€™s Law:

> Concurrency is nothing more than the number of requests a system can service at any given time and is normally driven by a fixed resource such as CPU.
>
> A systemâ€™s concurrency is normally calculated using Littleâ€™s law, which states: For a system at steady state, concurrency is the product of the average service time and the average service rate (L = ð›ŒW). Any requests in excess of this concurrency cannot immediately be serviced and must be queued or rejected. With that said some queueing is necessary as it enables full system utilization in spite of non-uniform request arrival and service time.

> Systems fail when no limit is enforced on this queue, such as during prolonged periods of time where the arrival rate exceeds the exit rate. As the queue grows so will latency until all requests start timing out and the system will ultimately run out of memory and crash. If left unchecked latency increases start adversely affecting its callers leading to cascading failures through the system.

For the Metrics Service, we just need to decide whether to accept â€” and enqueue â€” each request, or reject it as soon as we see it.

The accept or reject decision is going to track with how quickly the system is currently processing requests:

- If our response times slow down too much, we shouldnâ€™t take on new work until we start speeding up again â€” we should reduce our concurrency limit.
- If we are processing requests quickly, we have the capacity to take on more work â€” we can increase our concurrency limit.

Weâ€™ll look at the algorithm that adjusts the concurrency limit below, but first these are the details of how we accept or reject each request with the queuing model inside the Python gRPC server.

We keep track of â€œinflightâ€ requests â€” the requests in the queue + requests being worked on. Then we compare the current inflight count to the current concurrency limit. If the limit is greater than whatâ€™s inflight, we accept this request.

From the gRPC server point of view, this check happens when we are deciding to submit the request to our ThreadPoolExecutor work queue:

```
# if we have capacity to process this request, accept it
# and call "submit" to add it to the work queue
if self._limiter.has_capacity(key):
    future = self._delegate.submit(
        fn,
        rpc_event,
        state,
        behavior,
        lambda: request,
        request_deserializer,
        response_serializer,
    )
    self._limiter.incr(key)
    future.add_done_callback(on_done)
# if we don't have capacity, reject the request
else:
    with state.condition:
        grpc._server._abort(
            state,
            rpc_event.call,
            grpc._cython.cygrpc.StatusCode.resource_exhausted,  # 8
            self._RESOURCE_EXAUSTED_DETAILS,
        )
        raise ResourceExhaustedException(
            self._RESOURCE_EXAUSTED_DETAILS
```

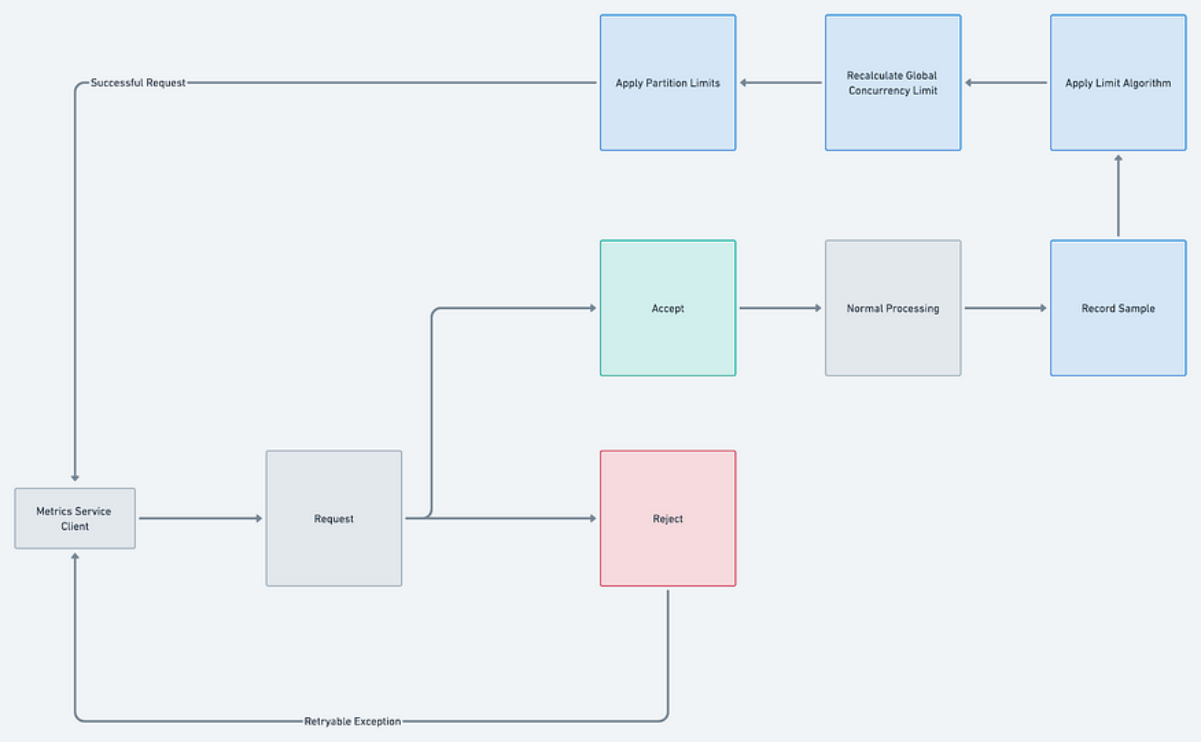
Now, if all we wanted to do was define a static limit, we would be done. This check would work as a configurable upper bound on how many inflight requests we would ever process. It would definitely solve one flavor of timeout conditions for us â€” when request volume spikes and overloads the queue.

In fact, gRPC already has configuration for this at the server level â€” maximum_concurrency_rpcs â€” so setting up a safe upper bound is as easy as setting this in your server configuration.

But remember, we can suffer timeout/overload conditions when *downstream* components slow down as well, like our database response times or how long it takes us to process a large result set. We need a way to adjust our concurrency up and down in response to downstream issues as well as spikes in incoming request volume.

Building an Adaptive Feedback Loop

The blue boxes in this diagram show the adaptive feedback loop.



Here are the steps:

1. After weâ€™ve finished processing the request, we record how long it took to process.
2. Latency observations are accumulated, so that we can derive, for example, the trailing p95 latency of the last 1,000 requests.
3. At a specified time interval, we run an AIMD Limit Algorithm (explained below) using the calculated latency from all our observations in that period.
4. Based on how we are processing the latest batch of request, the Limit Algorithm potentially recalculates the limit, essentially evaluating the current health of the service, so that we can decide whether or not to accept or reject the next request.

For our use case, we found that re-calculating the limit about every 3 seconds was sensitive enough to adjust capacity up and down, using the p95 latency of a few hundred requests. The

calculations happen independently on each server in the cluster â€” hence the hundreds of requests we evaluate, while the overall cluster servers thousands per second â€” with no need for coordination or centralization of the observations.

We are using an AIMD algorithm â€” Additive Increase, Multiplicative Decrease. When it runs, it gradually increases the limit (Additive Increase) if we are within our latency SLO or reduces the limit to a percentage of the current limit (Multiplicative Decrease) if we are exceeding the latency SLO.

This is our python implementation of Netflixâ€™s java version:

```
def _update(
    self,
    start_time: float,
    rtt: float,
    inflight: int,
    timeout_observed: bool
):
    # if we cross the latency threshold,
    # we backoff by our pre-configured backoff ratio
    if timeout_observed or rtt > self._latency_threshold_ms:
        self._current_limit = math.floor(
            self._current_limit * self._backoff_ratio
        )
    # otherwise, we can increase the limit if the current inflight
    # request count is approaching the limit
    # if they are far apart, we don't do anything
    elif inflight * 2 >= self._current_limit:
        self._current_limit += 1

    # finally, we make sure the limit is within the min/max bounds
    self._current_limit = min(
        self._max_limit, max(self._min_limit, self._current_limit)
    )
```