
Fairness in multi-tenant systems

David Yanacek



Introduction

This article looks at a few approaches Amazon has taken to manage API requests to its systems to avoid overload by implementing API rate limiting (also referred to as “throttling” or “admission control”). Without these kinds of protection, a system becomes [overloaded](#) when more traffic flows into the system than it is scaled to handle at that time. API rate limiting lets us shape incoming traffic in different ways, such as prioritizing the client workloads that remain within their planned usage, while applying backpressure to the client workload that spikes unpredictably. In this article, I’ll cover topics ranging from admission control algorithms to operational considerations for safely updating quota values in a production system. I also focus on ways Amazon has designed APIs with fairness in mind to provide predictable performance and availability and to help customers avoid the need for workloads that could lead to rate limiting.

However, before I get into the details of *how* Amazon systems implement rate limiting, I’ll first describe *why* Amazon has found rate-based quotas to be important, and why they lead to higher availability for clients of multi-tenant systems. Although you can read this article on its own, if you want some background on how systems behave when they are overloaded, check out the article [Using load shedding to avoid overload](#).

The case for multitenancy

Service-oriented architecture (SoA) is a core part of the Amazon culture of strong ownership and loose coupling of teams and systems. This architecture comes with another important benefit—improved hardware efficiency through resource sharing. If another application wants to make use of an existing service, the service owner doesn’t need to do much work to take on a new tenant. After reviewing the use case and performing a security review, the service owner grants the new client system access to call particular APIs or access particular data. No additional infrastructure setup or installation needs to happen for the service owner to create and operate a new copy of the service for this new use case—it just re-uses the existing one.

Resource sharing is a central benefit in multi-tenant systems. A *multi-tenant system* handles multiple workloads, such as work from multiple customers at once. This system can also handle low priority, non-urgent workloads along with high-priority, urgent workloads. A *single-tenant system*, on the other hand, handles workloads from a single customer.

Resource sharing is a powerful concept behind SoA and cloud architecture, leading to savings in infrastructure cost and human operational cost. In addition, resource sharing can lead to reduced environmental costs because higher utilization means fewer servers, and therefore less energy is needed to power and cool the infrastructure.

The evolution of multi-tenant databases

When I compare single-tenant and multi-tenant systems, I like to think of the differences between types of database systems. Nearly all of the systems I've worked on as a software engineer have needed some sort of database to store state. Some systems are low-traffic tools to make someone's life easier occasionally, and other systems are mission-critical and serve enormous amounts of traffic.

Early in my career at Amazon, I worked on a team that was responsible for automating the operations of the Amazon.com web server fleet. My coworkers and I built systems that forecasted how many servers we would need to provision over time for each website fleet, monitored the health of the servers, and performed automatic remediation and replacement of broken servers. We also built systems that helped teams deploy software to the website.

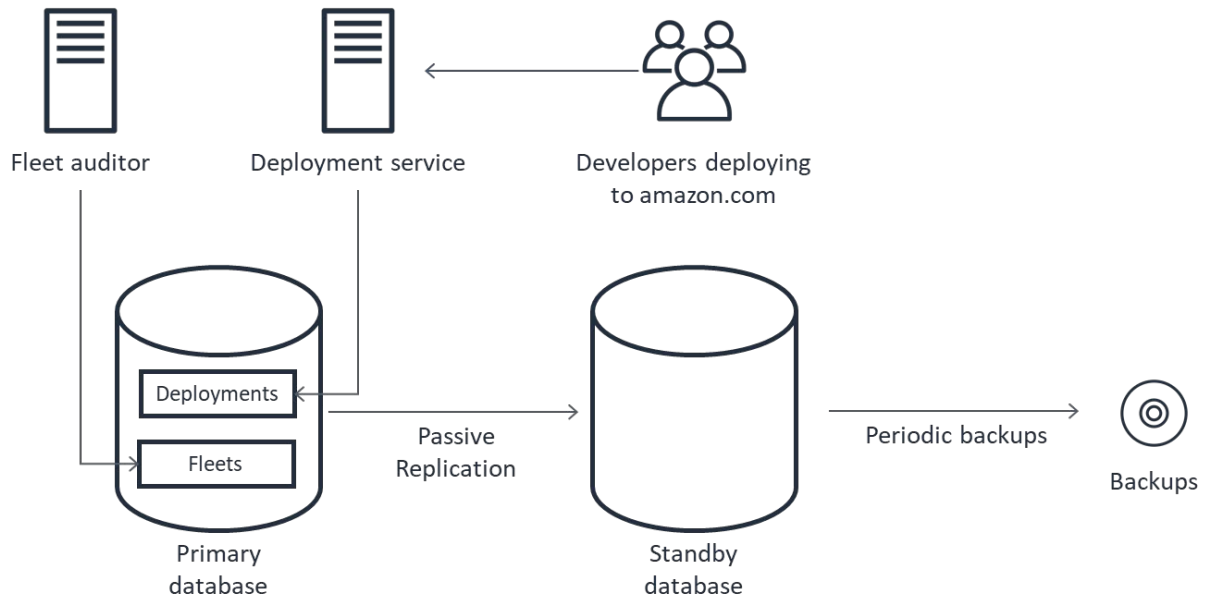
To manage state for many of these tools, we needed a database. At Amazon today, the default choice would be a NoSQL database because these databases are designed for manageability, scalability, availability, and predictable performance. However, this story took place before those technologies were widely available, so we set up some servers running MySQL, with replication pairs for high availability and redundancy. We also implemented backup and restore, and tested for failure to ensure that we could rely on this otherwise single point of failure.

When we built new services and tools that needed new databases, we were often tempted to re-use our existing databases by simply adding more tables to them. Why was that temptation so strong? For one, we looked at database server utilization and saw that the servers were not heavily loaded. Remember, these particular databases were at the scale of the number of developers deploying code to the website back in the mid-2000s and the number of web servers that we had at the time. In addition, the operational overhead for configuring new databases, monitoring them, upgrading them, and trying to automate every aspect of their operation was painful.

The following diagram shows an example architecture for the use of multiple website tools by Amazon.com website fleet operations a few years back. The diagram indicates that both the deployment service and the periodic fleet operator shared the same database, which our team operated.

Amazon.com website fleet automation tools circa 2007

(high level approximation)



However, when we gave in to the temptation to use the same set of database servers for multiple applications, we regretted it. I remember being on call and getting paged for a performance degradation in the deployment tool. I scratched my head trying to figure out what was wrong. Then I realized that a completely separate tool that we owned (a fleet auditor of some kind) was running its nightly state synchronization cron job. This put a lot of extra load on the shared database. The fleet auditor tool didn't mind that the database was slow, but the deployment tool (and its users) sure did!

This constant tension between the desire to share databases (for lower costs for infrastructure and some operations) versus the need to separate databases (for better workload isolation and lower operational costs in other ways) felt like a no-win scenario—a [Kobayashi Maru](#) of sorts. The types of databases we used were meant for a single tenant, so, unsurprisingly, we ran into pain when we tried to use them in a multi-tenant way.

When Amazon Relational Database Service (RDS) was launched, it made our lives easier by automating much of that operational work. It was easier for us to run single-tenant systems as separate databases rather than sharing the same database across multiple applications. However, some workloads were quite small and other workloads varied in size, so we still needed to pay attention to the utilization of each database to get the instance sizes just right. Plus, we needed enough headroom to handle periodic swings in load.

Later at Amazon, when I was seeking out a new challenge, I learned about a new type of database we were building in AWS. The goals of the database were to be highly elastic, scalable, available, low latency, and fully managed. These goals were compelling to me because as a software engineer, I really dislike doing the same thing over and over again, especially if it's

hard to do. So, I expend a great deal of effort trying to automate those repeatable tasks (see also <https://xkcd.com/1319/>). This new database seemed like the perfect opportunity to finally fully automate every aspect of database maintenance that I found painful, so I joined the team that launched Amazon DynamoDB in 2012. Just like Captain Kirk, we used programming to pass the Kobayashi Maru test!

DynamoDB leverages multitenancy to provide a database that is highly elastic, durable, and available. Unlike when I use Amazon RDS, when I create resources in DynamoDB, I don't even provision a whole Amazon Elastic Compute Cloud (EC2) instance. I simply interact with my database through an API, and behind the scenes DynamoDB figures out the fraction of a server that's required for the workload. (Actually, it's using a fraction of multiple servers in multiple Availability Zones for high availability and durability.) As my workload grows and shrinks, DynamoDB adjusts that fraction, and enlists more servers or fewer, as needed.

Just as with databases, there are degrees of multitenancy for general-purpose compute servers. With AWS Lambda, compute resource sharing happens at a sub-second interval, using [Firecracker](#) lightweight virtualization for resource isolation. With Amazon API Gateway, resource sharing is at the API request level. Customers of these services benefit from the advantages of multi-tenant systems: elasticity, efficiency, and ease of use. Under the hood, these services work to address challenges that come with multitenancy. Of these challenges, the one I find the most interesting is fairness.

Fairness in multi-tenant systems

Any multitenancy service works in concert with systems to ensure fairness. *Fairness* means that every client in a multi-tenant system is provided with a single-tenant experience. The systems that ensure fairness in multi-tenant systems are similar to systems that perform bin-packing algorithms, which are classic algorithms in computer science. These fairness systems do the following things:

- Perform placement algorithms to find a spot in the fleet for new workload. (Similar to finding a bin with room for the workload.)
- Continuously monitor the utilization of each workload and each server to move workloads around. (Similar to moving workloads between bins to ensure that no bin is too full.)
- Monitor the overall fleet utilization, and add or remove capacity as needed. (Similar to adding more bins when they're all getting full, and removing bins when they're empty.)
- Allow workloads to stretch beyond hard-allocated performance boundaries as long as the underlying system isn't being fully utilized, and hold workloads to their boundaries when the system is fully utilized. (Similar to allowing workloads to stretch within each bin as long as they're not crowding out other workloads.)

Advanced fairness systems combine these techniques in interesting ways. For example, a fairness system can monitor the utilization of each workload, estimate how likely it is that any two workloads will play nicely together, and move them together into the same bin. As long as one workload isn't fully utilizing its provisioned resources, another workload in the same bin can borrow those resources.

For this resource sharing to work, the borrowing needs to go unnoticed by the workloads. If a workload needs to use all of its provisioned resources, the time to return those borrowed resources needs to be nearly instantaneous. In addition, workloads need to be moved quickly between bins. If a busy workload grows accustomed to exceeding its provisioned resources by borrowing from a neighbor, but its neighbor changes its behavior and starts using more of its provisioned resources, the busy workload needs to be moved to another bin.

Load shedding plus fairness

As load increases on a system, it should automatically scale up. The simplest way to do this is to add more capacity and scale horizontally. For services that adopt a serverless architecture, such as those built on AWS Lambda, horizontal scaling happens nearly instantaneously as capacity is spun up on demand to handle work. For non-serverless services, it takes longer to auto scale.

Typically, even scaling within minutes is fine. However, what happens if load increases on a service faster than Auto Scaling can add capacity? The service has two options: It can become overloaded and slow for all requests, or it can [shed excess load](#) and maintain consistent performance for the requests that it does take on. At Amazon, we greatly prefer maintaining consistent, predictable performance (option two)—especially during any overload situation. Adding increased latency to an overload scenario (option one) can cause ripple effects in a distributed system that spread the impact to other systems. The fail-fast strategy of option two helps overloaded systems continue to make progress and do useful work.

We've found that load shedding is one useful tool for dropping excess load in an overload scenario. [Load shedding](#) is the act of cheaply rejecting work instead of spending limited resources on it. For an HTTP service, load shedding means returning an immediate error. An example of this is returning an HTTP 503 error code. This buys time for Auto Scaling to kick in and add the necessary capacity, since the alternative to returning the error is to become slow for all requests. Because returning a load shed response to a request is significantly cheaper than fully processing the request, this approach lets the server continue to offer predictable performance for the requests it decides to fully process.

Typically, we design our services to return load shed responses to the client as quickly as possible to minimize the amount of work performed by the server. However, in some situations, we deliberately slow down these responses slightly. For example, when a load balancer is used with a [least outstanding requests](#) algorithm, we slow down fast error responses to match the latency of successful responses. This way we avoid having the load balancer send extra traffic to a server that might already be overloaded.

In a multi-tenant service however, load shedding is not enough to make the multi-tenant service appear as a single-tenant service to each customer. Typically, load from multiple tenants is uncorrelated (that is, each customer has its own use case and request rate). Therefore, if the overall load increases abruptly on a service, that increase is most likely driven by a single tenant. With fairness in mind, we want to avoid failing some number of requests across all tenants in response to an unplanned load increase from a single tenant.

To add *fairness* to multi-tenant systems, we use rate limiting to shape unplanned increases in traffic, but we enforce quotas (maximum values for resources and actions) at a per-tenant or per-workload granularity. This way, if the multi-tenant service experiences an unplanned increase in load, the unplanned portion of that workload is rejected, and the other workloads continue operating with predictable performance.

However, the use of quotas paradoxically both increases and decreases the availability of a service. When one tenant's workload exceeds its quota, it will start to see its excess requests fail—this can be perceived as a drop in availability. However, in reality, the service might have had plenty of capacity to serve those requests. API rate limiting is a useful technique for protecting our services' availability, but we also spend effort to help our callers avoid exceeding their quota unnecessarily.

Like load shedding, enforcing rate-based quotas involves cheaply sending an error response instead of handling the request. However, this response indicates that the client has exceeded its quota, not that the server is out of capacity. Therefore “API rate limit exceeded” responses generally are returned with a 429 status code. In general, status codes in the 500 range mean that the server failed for some reason, but status codes in the 400 range mean that the client is doing something unexpected, or, in this case, unplanned.

Note You might notice that some AWS services actually return 503 status codes for rate exceeded. The 429 status code was not officially added to the HTTP specification until 2012 in [RFC 6585](#). Many AWS services were created before that, starting with the release of Amazon Simple Queue Service (SQS) in 2004. AWS focuses heavily on backward compatibility, so we haven't changed the behavior of pre-existing services to avoid breaking clients unnecessarily.

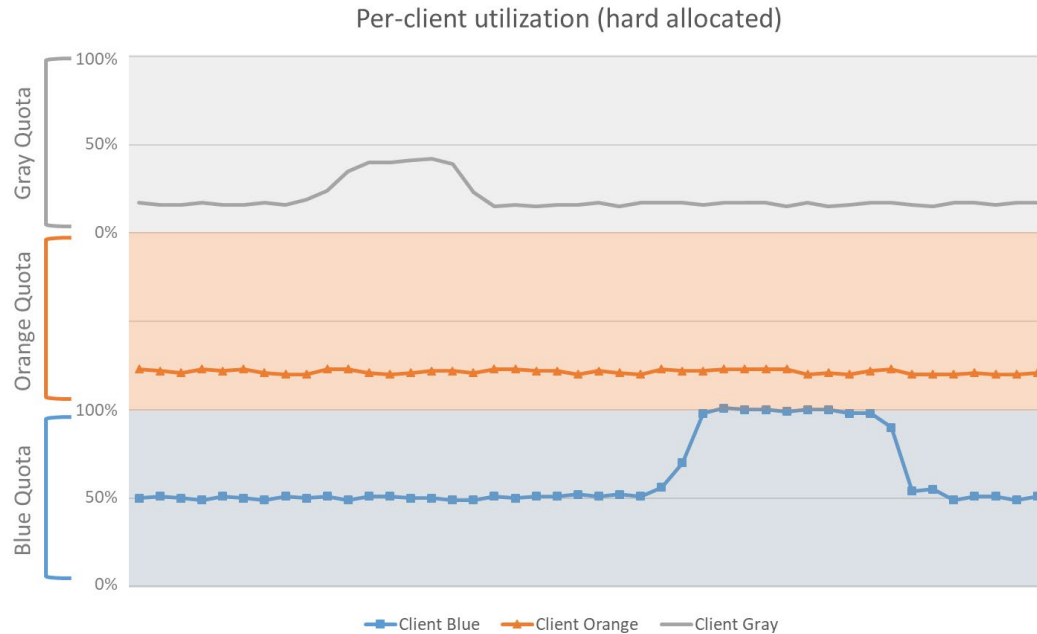
Quota visibility and flexibility

Service owners often configure a quota per client. For example, for AWS services, a client is typically an AWS account. Sometimes quotas are placed on something more fine-grained than client, such as on a particular resource owned by a client, like a DynamoDB table. Service owners define rules that give each caller a default quota. If a client grows its usage in the normal course of business and is approaching its limit, or if the client anticipates an upcoming load increase, they often ask the service to raise its quota.

There are a few types of quotas, each measured with its own units. One type of quota governs as “the number of things the client can have running at the same time.” For example, Amazon EC2 implements quotas for the number of instances that can be launched by a particular AWS account. Another type of quota is a rate-based quota. Rate-based quotas are often measured in units like “requests per second.” Although this article focuses on the nuances of rate-based quotas, many concepts that apply to rate-based quotas also apply to other types, so throughout this article I'll just use the word “quota.”

The following graph demonstrates the use of quotas. It shows a service with finite capacity (total provisioned capacity is represented by the maximum of the y-axis). The service has three clients: Blue, Orange, and Gray. The service has hard-allocated each client one-third of its total capacity. The graph shows that client Blue is attempting to exceed its hard-allocated throughput, but it's not able to do so.

Fairness in multi-tenant systems



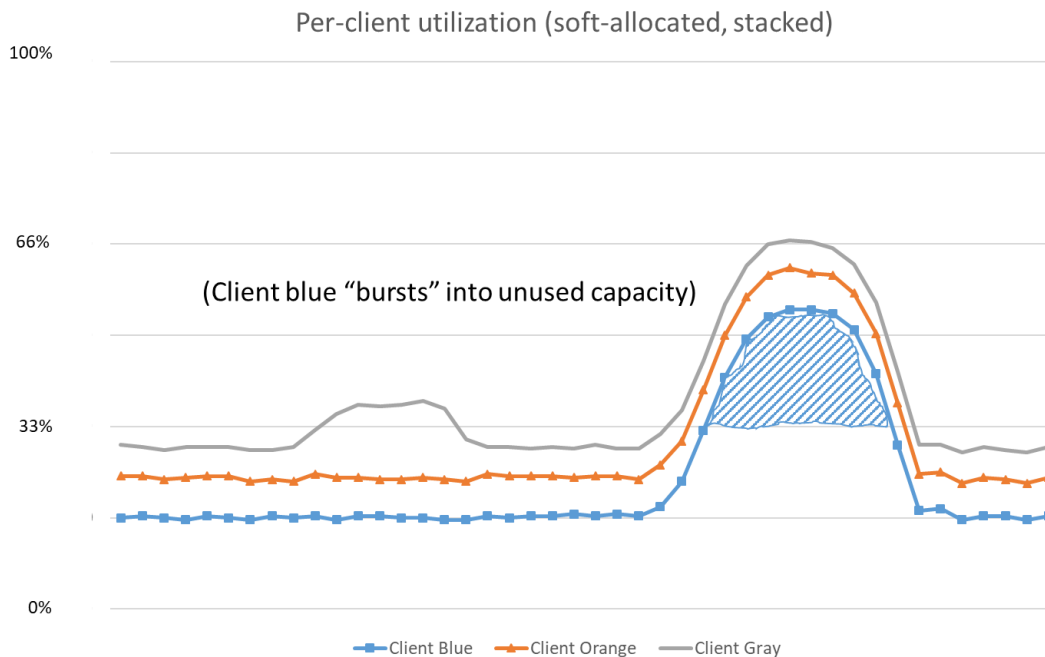
For this quota allocation to scale operationally, services expose information to clients about their quota and how close they are to hitting their quota. After all, when a client exceeds its quota, it's likely that it's returning errors to its clients in response. Therefore, the services provide clients with metrics that they can see and use to alarm on when their utilization is approaching the maximum quota value. For example, DynamoDB publishes Amazon CloudWatch metrics that show the throughput that is provisioned for a table, and how much of that throughput is consumed over time.

Some APIs are far more expensive for a service than others. Because of this, services might give each client a lower quota for expensive APIs. Similarly, the cost of an operation is not always known up front. For example, a query that returns a single 1 KB row is less expensive than one that returns up to 1 MB of rows. Pagination prevents this expense from getting too far out of control, but there still can be enough of a cost difference between the minimum and maximum page size to make setting the right threshold challenging. To handle this issue, some services simply count larger responses as multiple requests. One implementation of this technique is to first treat every request as the cheapest request, and then, after the API call is completed, go back and debit the client's quota based on the true request cost, possibly even pushing their quota negative until enough time passes to account for the actual usage.

There can be some flexibility in implementing quotas. Consider the case where client A has a limit of 1,000 transactions per second (TPS), but the service is scaled to handle 10,000 TPS, and the service is currently serving 5,000 TPS across all of its clients. If client A spikes from 500 TPS to 3,000 TPS, only its 1,000 TPS would be allowed, and the other 2,000 TPS would be rejected. However, instead of rejecting those requests, the service could allow them. If other clients simultaneously use more of their quotas, the service can begin to drop client A's "over quota"

requests. Dipping into this "unplanned capacity" should also act as a signal to operators of the client and/or the service. The client should know that it's exceeding its quota and is at risk of seeing errors in the future. The service should know that it might need to scale its fleet and increase the client's quota automatically.

To demonstrate this situation, we created a graph similar to the one used previously to show a service that hard-allocated capacity to its clients. However, in the following graph the service stacked capacity for its clients, instead of hard-allocating it. Stacking allows the clients to use the unutilized service capacity. Since Orange and Gray aren't using their capacity, Blue is allowed to exceed its provisioned thresholds and tap into (or *burst* into) the unused capacity. If Orange or Gray were to decide to use their capacity, their traffic should take priority over Blue's burst traffic.



At Amazon, we also look at flexibility and bursting by considering typical customer use case traffic patterns. For example, we found that EC2 instances (and their attached Amazon Elastic Block Store (EBS) volumes) are frequently busier at the time the instance is launched than they are later on. This is because when an instance launches, its operating system and application code need to be downloaded and started up. When we considered this traffic pattern, we found that we could be more generous with up-front burst quotas. This results in reduced boot times, and still provides the long-term capacity planning tools that we need to provide fairness between workloads.

We also look for ways to allow quotas to be flexible over time and adjust to the increase in a client's traffic that happens as their business grows. For example, some services automatically

increase quotas for clients over time as they grow. However, there are some cases where clients want and depend on fixed quotas, for example, quotas used for cost control. Note that this type of quota is likely to be exposed as a feature of a service rather than a protection mechanism that happens behind the scenes.

Implementing layers of admission control

The systems that shape traffic, shed load, and implement rate-based quotas are known as *admission control systems*.

Services at Amazon are architected with multiple layers of admission control to protect against large volumes of rejected requests. At Amazon, we often use [Amazon API Gateway](#) in front of our services and let it handle some dimensions of quotas and rate limiting. API Gateway can absorb a surge in traffic with its large fleets. This means that our service fleets remain unburdened and free to serve real traffic. We also configure [Application Load Balancer](#), API Gateway, or [Amazon CloudFront](#) to use the web application firewall service [AWS WAF](#) to offload the act of admission control even further. For a layer of protection beyond that, [AWS Shield](#) offers DDoS protection services.

We've used a number of techniques to implement these layers of admission control in systems at Amazon over the years. In this section, we explore some of those techniques, including how we build server-side admission control, how we implement clients that react gracefully to backpressure from services they call, and how we think about accuracy in these systems.

Local admission control

One common method for implementing admission control is to use a token bucket algorithm. A token bucket holds tokens, and whenever a request is admitted, a token is taken out of the bucket. If there aren't any tokens available, the request is rejected, and the bucket remains empty. Tokens are added to the bucket at a configured rate, up to a maximum capacity. This maximum capacity is known as the *burst capacity* because these tokens can be consumed instantly, supporting a burst in traffic.

This instantaneous burst consumption of tokens is a double-edged sword. It allows for some natural non-uniformity in traffic, but if the burst capacity is too large it defeats the protections of rate limiting.

Alternatively, token buckets can be composed together to prevent unbounded burst. One token bucket can have a relatively low rate and a high burst capacity, and a second bucket can have a high rate and a low burst capacity. By checking the first bucket and then the second bucket, we allow for high burst, but with a bounded burst rate.

For a traditional service (one that doesn't have a serverless architecture), we also consider how uniform or non-uniform the requests are across our servers for a given customer. If requests are non-uniform, we use more relaxed bursting values or distributed admission control techniques.

There are many off-the-shelf implementations of local rate limiting available, including Google Guava's [RateLimiter](#) class.

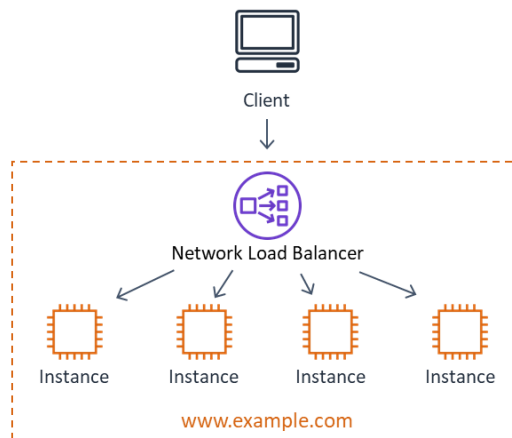
Distributed admission control

Local admission control is useful for protecting a local resource, but quota enforcement or fairness often needs to be enforced across a horizontally scaled fleet. Teams at Amazon have taken many different approaches to solve this problem of distributed admission control, including:

Computing rates locally and dividing the quota by the number of servers. Using this approach, servers perform admission control based on traffic rates they observe locally, but they divide the quota for each key by the number of servers that are serving traffic for that throttle key. This approach assumes that requests are relatively uniformly distributed across servers. When an Elastic Load Balancing load balancer spreads requests across servers in a round-robin fashion, this generally holds true.

The following diagram shows a service architecture that assumes that traffic is relatively uniform across instances and can be handled using a single logical load balancer.

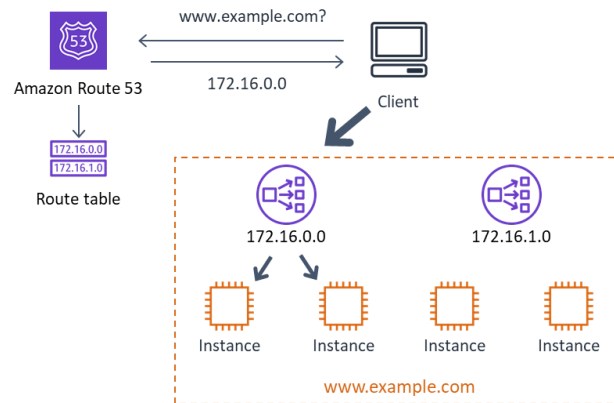
Uniform traffic with a single endpoint



However, assumptions about uniformity across servers might not always be true in some fleet configurations. For example, when a load balancer is used in a connection balancing mode instead of a request balancing mode, clients with few enough connections will send their requests to a subset of servers at a time. This could be fine in practice when the per-key quota is high enough. Assumptions about uniformity across servers can also break down when there is a very large fleet with multiple load balancers. In cases like this, a customer could connect through a subset of the load balancers, resulting in only a subset of the service instances serving the requests. Again, this could be fine in practice if the quotas are high enough, or if clients are unlikely to get close enough to their quota maximum for this case to apply.

The following diagram illustrates a situation where services that are fronted by multiple load balancers find that traffic from a given client is not uniformly spread across all servers due to DNS caching. This tends to be less of an issue at scale, when clients are opening and closing connections over time.

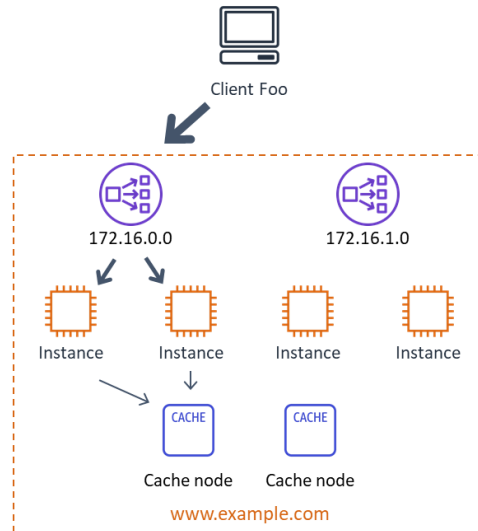
Non-uniform traffic from multiple endpoints



Using consistent hashing for distributed admission control. Some service owners run a separate fleet, such as an Amazon ElastiCache for Redis fleet. They apply consistent hashing on the throttle keys to a particular rate tracker server, and then have the rate tracker servers perform admission control based on local information. This solution even scales well in cases where key cardinality is high because each rate tracker server only needs to know about a subset of the keys. However, a basic implementation would create a “hot spot” in the cache fleet when a particular throttle key is requested at a high enough rate, so intelligence needs to be added to the service to gradually rely more on local admission control for a particular key as its throughput increases.

The following diagram illustrates using consistent hashing to a data store. Even where traffic isn't uniform, using consistent hashing to count traffic across some sort of datastore, such as a cache, can solve the distributed admission control problem. However this architecture introduces scaling challenges.

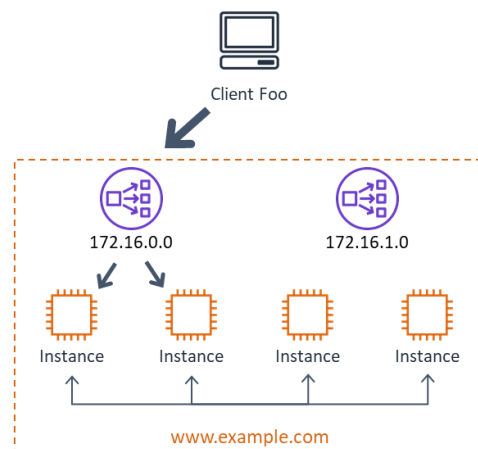
Consistent hashing to a data store



Taking other approaches. At Amazon, we've implemented many distributed admission control algorithms in web services over time, with varying degrees of overhead and accuracy depending on the specific use case. These approaches involve periodically sharing the observed rates for each throttle key across a fleet of servers. There are many tradeoffs between scalability, accuracy, and operational simplicity in these approaches, but they would need their own articles to explain and compare them in depth. For some starting points, check out the research papers on fairness and admission control, many of which are from the networking space, which I link to at the end of this article.

The following diagram illustrates the use of asynchronous information sharing between servers to account for non-uniform traffic. This approach comes with its own scaling and accuracy challenges.

Asynchronous information sharing



Reactive admission control

Quotas are important for handling regular unexpected spikes in traffic, but services should be prepared to encounter all kinds of unexpected workloads. For example, a buggy client could send malformed requests, or a client could send a workload that is more expensive to process than anticipated, or a client could have some runaway application logic and ask for help in filtering out their accidental traffic. Flexibility is important so we can put an admission control system in place that can react to various aspects of the request, such as HTTP headers like the user-agent, or the URI, or the source IP address.

In addition to ensuring the right visibility and hooks, we incorporate mechanisms for changing rate limit rules quickly and carefully. Rule configuration that loads into memory on process startup might seem like a good first cut. However, it can be very awkward to deploy rule changes quickly when the situation demands these changes. It's also important to implement a dynamic configuration solution with safety in mind, and keep an audit trail of what has changed over time. Some systems at Amazon deploy quota value configuration changes first in an evaluation mode, where we verify that the rules would affect the right traffic before we make the rules live.

Admission control of high cardinality dimensions

For most of the types of quotas we have explored so far, admission control systems need to keep track of the current observed rate and quota values for a fairly small number of things. For example, if a service is called by ten different applications, the admission control system might only need to track ten different rates and quota values. However, admission control becomes more complex when dealing with a high cardinality dimension. For example, a system might set a rate-based quota for each IPv6 address in the world, or for each row in their DynamoDB table, or each object in their Amazon Simple Storage Service (S3) bucket. The numbers of these things are practically unbounded, so there is no reasonable amount of memory that could keep track of rates and quotas for each of them.

To put an upper bound on the amount of memory that an admission control system needs to use for these dimensions, we use algorithms like Heavy Hitters, Top Talkers, and Counting Bloom filters, which each provide interesting guarantees around accuracy and error bounds, while limiting the used memory.

When we operate a system with high cardinality dimension like these, we also need operational visibility into the change in traffic over time. Within Amazon we use Amazon [CloudWatch Contributor Insights](#) to analyze high-cardinality traffic patterns for our own services.

Reacting to rate-exceeded responses

When a client of a service receives a rate-exceeded error, it can either [retry](#) or return an error. Systems at Amazon can behave one of two ways in response to rate exceeded errors, depending on whether they are synchronous systems or asynchronous systems.

Synchronous systems need to be impatient because they have someone waiting for them to respond. Retrying the request would have some chance of succeeding on the next attempt. However, if the dependent service is returning rate-exceeded responses frequently, retrying would just slow down every response, and would tie up more resources on an already heavily loaded system. This is why the AWS SDK automatically [stops retrying](#) when a service returns errors frequently. (At the time of writing, this behavior requires the client application to set the [STANDARD](#) retry mode in the SDK.)

Many asynchronous systems have it easier. In response to receiving rate-exceeded responses, they can simply apply back pressure and slow down their processing for a time until all of their requests are succeeding again. Some asynchronous systems run periodically, and expect their work to take a long time to be completed. For these systems, they can attempt to execute as quickly as they can, and slow down when some dependency becomes a bottleneck.

Other asynchronous systems are not expected to encounter significant processing delays, and they can build up large backlogs of work if they can't complete work quickly enough. These types of asynchronous systems have requirements that are more similar to synchronous systems. Techniques for error handling in these systems are covered in more depth in the article [Avoiding insurmountable queue backlogs](#).

Evaluating admission control accuracy

No matter which admission control algorithm we use to protect a service, we find it's important to evaluate that algorithm's accuracy. One method we use is to include the throttle key and rate limit in our service's request log on every request, and perform log analysis to measure the actual fleet-wide requests per second for each throttle key. Then we compare that to the configured limit. From this, for each key, we analyze the "true positive rate" (the rate of requests that were correctly rejected), the "true negative rate" (the rate of requests that were correctly allowed), the "false positive rate" (the rate of requests that were incorrectly rejected), and the "false negative rate" (the rate of requests that were incorrectly admitted).

We use many tools to perform log analysis like this, including CloudWatch Logs Insights and Amazon Athena.

Architectural approaches to avoid quotas

It might seem easy to add admission control to a service to improve its server-side availability, protect customers from each other, and declare victory. However, we also view quotas as an inconvenience for customers. Quotas slow customers down when they are trying to get something done. As we build fairness mechanisms into our services, we also look for ways to help customers get their work done quickly, without having their throughput limited by their quotas.

Our approach for helping clients avoid exceeding their rate-based quota varies depending on whether the API is a [control plane API or a data plane API](#). A *data plane* API action is intended to be called at a higher and higher rate over time. Examples of data plane API actions include Amazon Simple Storage Service (S3) GetObject, Amazon DynamoDB GetItem, and Amazon SQS

ReceiveMessage. On the other hand, a *control plane* API action can be intended for occasional, low-volume use cases that don't grow with the customer's data plane usage. Examples of control plane API actions include Amazon S3 CreateBucket, Amazon DynamoDB DescribeTable, and Amazon EC2 DescribeInstances.

Capacity management approaches to avoid exceeding quotas

Data plane workloads are elastic, so we design our data plane services to be elastic. To make a service elastic, we design the underlying infrastructure to automatically scale to changes in customer workloads. We also need to help customers maintain this elasticity when managing quotas. Service teams across Amazon use a variety of techniques to help their customers manage quotas and achieve their elasticity needs:

- If the fleet is provisioned with some “slack” capacity that is underutilized, we let the caller burst into it.
- We implement Auto Scaling and increase each caller's limit as they grow in the normal course of business.
- We make it easy for customers to see how close they are to their limits, and let them configure alarms to let them know when they're reaching those limits.
- We pay attention to when callers are approaching and reaching their limits—they might not have noticed. At the very least, we alarm when the service is rate-limiting traffic at a high overall rate or rate limiting too many customers at the same time.

API design approaches to avoid exceeding quotas

Particularly for control planes, some of the techniques I discussed earlier might not apply. Control planes are designed to be called relatively infrequently, whereas data planes are designed to be called at high volumes. However, when clients of a control plane end up creating many resources, they still need to be able to manage, audit, and perform other operations on those resources. Customers might use up their quotas and encounter API rate limits when managing many resources at scale, so we look for alternative ways to address their needs with different kinds of API actions. The following are some approaches AWS takes for designing APIs that help clients avoid the call patterns that could lead to using up their rate-based quota:

- **Supporting a change stream.** We found, for example, that some customers poll the Amazon EC2 DescribeInstances API action periodically to list all of their EC2 instances. This way they find instances that have been recently launched or terminated. As customers' EC2 fleets grow, these calls become more and more expensive, resulting in an increased chance of exceeding their rate-based quotas. For some use cases, we were able to help customers avoid calling the API altogether by providing the same information through [AWS CloudTrail](#). CloudTrail exposes a change log of operations, so instead of periodically polling the EC2 API, customers can react to changes from the stream.
- **Exporting data to another place that supports higher call volumes.** The [S3 Inventory API](#) is a real-world example. We heard from customers who had an enormous number of objects in their Amazon S3 buckets that they needed to sift through so they could find particular objects. They were using the ListObjects API action. To help customers achieve high

throughput, Amazon S3 provided an Inventory API action that asynchronously exports a list of the objects in the bucket into a different S3 object called an Inventory Manifest file, which contains a serialized list in JSON of all objects in the bucket. Now customers can access a manifest of their bucket at data plane throughput.

- **Adding a bulk API to support high volumes of writes.** We heard from customers that they want to call some write API actions to create or update a large number of entities in a control plane. Some customers were willing to tolerate the rate limits imposed by the API. However, they didn't want to deal with the complexity of writing a long-running import or update program, and they didn't want to deal with the complexity of partial failures and rate limiting to avoid crowding out their other write use cases. One service, AWS IoT, addressed this issue through API design. It added asynchronous [Bulk Provisioning APIs](#). Using these API actions, customers upload a file containing all of the changes they want to make, and when the service has finished those changes, it provides the caller with a file containing the results. Examples of these results include which operations succeeded and which failed. This makes it convenient for customers to deal with large batches of operations, but they don't need to deal with the details of retrying, partial failures, and spreading out a workload over time.
- **Projecting control plane data into places where it needs to be commonly referenced.** The Amazon EC2 DescribeInstances control plane API action returns all of the metadata about instances from network interfaces for each instance to block device mapping. However, some of this metadata is very relevant to code that runs on the instances themselves. When there are a lot of instances, the traffic from every instance calling DescribeInstances would be large. If the call failed (due to a rate-exceeded error or for some other reason), it would be a problem for customers' applications running on the instance. To avoid these calls altogether, Amazon EC2 exposes a local service on each instance that vends instance metadata for that particular instance. By projecting the control plane data to the instances themselves, customers' applications avoid approaching API rate limits altogether by avoiding the remote calls altogether.

Admission control as a feature

In some cases, customers find admission control preferable to unbound elasticity, because it helps them control costs. Typically, services don't charge customers for rejected requests, since they tend to happen rarely and are relatively cheap to process. Customers of AWS Lambda, for example, asked for the ability to control costs by [limiting the number of concurrent invocations](#) of a potentially expensive function. When customers want this kind of control, it's important to make it easy for them to adjust their limits on their own through API calls. They also need to have plenty of visibility and alarming capabilities. This way they can see problems in their system and respond by raising their limit, if they decide it's necessary.

Conclusion

Multitenant services have resource-sharing properties that enable them to operate with lower infrastructure cost and higher operational efficiency. At Amazon, we build fairness into our multitenant systems to provide our customers with predictable performance and availability.

Service quotas are an important tool for implementing fairness. Rate-based quotas make web services more reliable for multi-tenant customers by preventing an unpredictable increase in one workload from affecting others. However, implementing rate-based quotas is not always enough to provide a good customer experience. Customer visibility, controls, burst sharing, and different flavors of APIs all help customers avoid exceeding their quotas.

The implementation of admission control in a distributed system is complex. Fortunately, over time we have found ways to generalize this functionality and expose it in various AWS services. For API rate limiting, API Gateway offers multiple varieties of throttling as a feature. AWS WAF offers another layer of service protection, and it integrates into Application Load Balancer and API Gateway. DynamoDB offers provisioned throughput controls at the individual index level, letting customers isolate the throughput requirements of different workloads. Similarly, AWS Lambda exposes per-function concurrency isolation to isolate workloads from each other.

At Amazon, we find that admission control using quotas is an important approach for building highly resilient services with predictable performance. However, admission control is not enough. We make sure to solve the easy things too, like using Auto Scaling so that if there is unintended load shedding, our systems automatically respond to increased demand by auto scaling.

On the surface, it seems like there is an inherent tradeoff between cost and workload isolation between exposing a service as a single-tenant service versus as a multi-tenant service. However, we have found that by implementing fairness in multi-tenant systems, our customers get the best of both the multi-tenant and single-tenant worlds. As a result, they can [have their cake and eat it too](#).

Links

- [Amazon API Gateway](#)
- [AWS WAF - Web Application Firewall](#)
- [Lambda per-function concurrency](#)
- [Google Guava Rate Limiter](#)
- [DynamoDB use of token bucket algorithms](#)
- Zhang, et al [Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Applications](#), AT&T Labs, 2004
- Floyd, et al [Random Early Detection Gateways for Congestion Avoidance](#), IEEE/ACM, 1993
- Charny [An Algorithm for Rate Allocation in a Packet-Switching Network with Feedback](#), MIT, 1994
- Feng, et al [BLUE: A New Class of Active Queue Management Algorithms](#), University of Michigan, 2013
- McKenny, [Stochastic Fairness Queuing](#), 2002
- Kemp, et al [Gossip-Based Computation of Aggregate Information](#), IEEE, 2003