

Building well-architected serverless applications: Regulating inbound request rates – part 2

by Julian Wood | on 27 JUL 2021 | in [Amazon API Gateway](#), [Amazon QuickSight](#), [Amazon RDS](#), [Amazon Simple Queue Service \(SQS\)](#), [AWS Lambda](#), [AWS Well-Architected Framework](#), [Kinesis Data Streams](#), [Serverless](#) | [Permalink](#) |

[↪ Share](#)

This series of blog posts uses the [AWS Well-Architected Tool](#) with the [Serverless Lens](#) to help customers build and operate applications using best practices. In each post, I address the serverless-specific questions identified by the Serverless Lens along with the recommended best practices. See the [introduction post](#) for a table of contents and explanation of the example application.

Reliability question REL1: How do you regulate inbound request rates?

This post continues [part 1](#) of this security question. Previously, I cover controlling inbound request rates using throttling. I go through how to use throttling to control steady-rate and burst rate requests. I show some solutions for performance testing to identify the request rates that your workload can sustain before impacting performance.

Good practice: Use, analyze, and enforce API quotas

API quotas limit the maximum number of requests a given API key can submit within a specified time interval. Metering API consumers provides a better understanding of how different consumers use your workload at sustained and burst rates at any point in time. With this information, you can determine fine-grained rate limiting for multiple quota limits. These can be done according to a group of consumer needs, and can adjust their limits on a regular basis.

Segregate API consumers steady-rate requests and their quota into multiple buckets or tiers

[Amazon API Gateway](#) usage plans allow your API consumer to access selected APIs at agreed-upon request rates and quotas. These help your consumers meet their business requirements and budget constraints. Create and attach API keys to usage plans to control access to certain API stages. I show how to create usage plans and how to associate them with API keys in "[Building well-architected serverless applications: Controlling serverless API access – part 2](#)".

The screenshot displays the AWS API Gateway console for a usage plan named 'RestrictUsage'. The 'Details' tab shows the plan's configuration: ID 'jcfq33', Name 'RestrictUsage', Description 'No description.', Rate '1 requests per second', Burst '2 requests', and Quota '10 requests per day'. Below this, the 'Associated API Stages' section shows a table with one entry: 'Airline-Loyalty-develop' in the 'Prod' stage, with 'No Methods Configured'. The 'API Keys' tab is also visible, showing a search bar and a table with one entry: 'julianwood'.

API	Stage	Method Throttling
Airline-Loyalty-develop	Prod	No Methods Configured

Name	Usage	Extension
julianwood		

API key associated with usage plan

You can extract utilization data from usage plans to analyze API usage on a per-API key basis. In the example, I show how to use usage plans to see how many requests are made.

The screenshot shows the 'Usage Plans' list on the left, with 'RestrictUsage' selected. The main panel shows the 'Usage' tab for the 'RestrictUsage' plan. A table displays the usage for the API key 'julianwood', showing '10 requests made on Jul 21, 2020 (UTC)'.

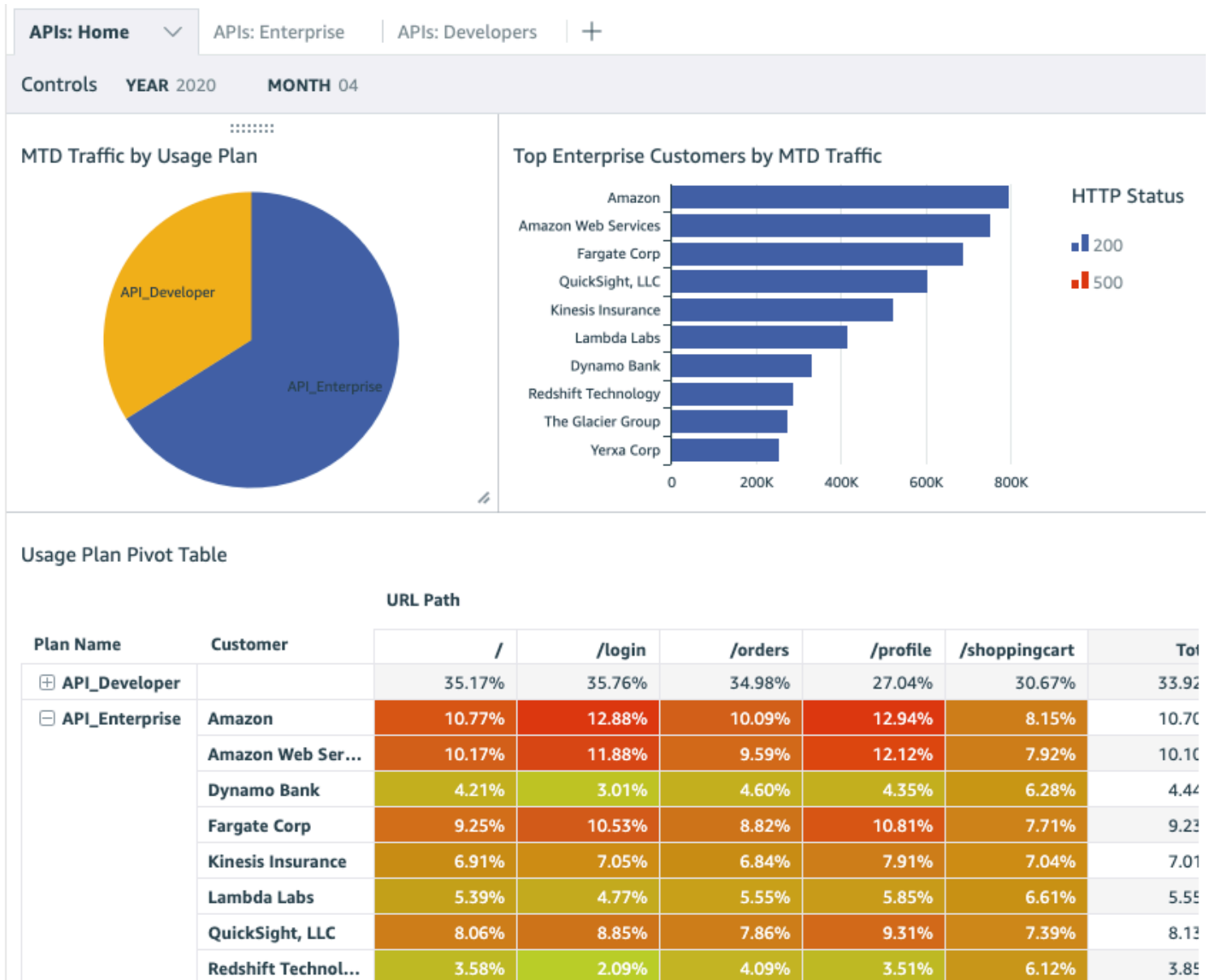
Name	Usage	Extension
julianwood	10 requests made on Jul 21, 2020 (UTC)	

View API key usage

This allows you to generate billing documents and determine whether your customers need higher or lower limits. Have a mechanism to allow customers to request higher limits preemptively. When customers anticipate greater API usage, they can take action proactively.

API Gateway Lambda authorizers can dynamically associate API keys to a given request. This can be used where you do not control API consumers, or want to associate API keys based on your own criteria. For more information, see the [documentation](#).

You can also [visualize usage plans](#) with [Amazon QuickSight](#) using enriched API Gateway access logs.



Visualize usage plans with Amazon QuickSight

Define whether your API consumers are end users or machines

Understanding your API consumers helps you manage how they connect to your API. This helps you define a request access pattern strategy, which can distinguish between end users or machines.

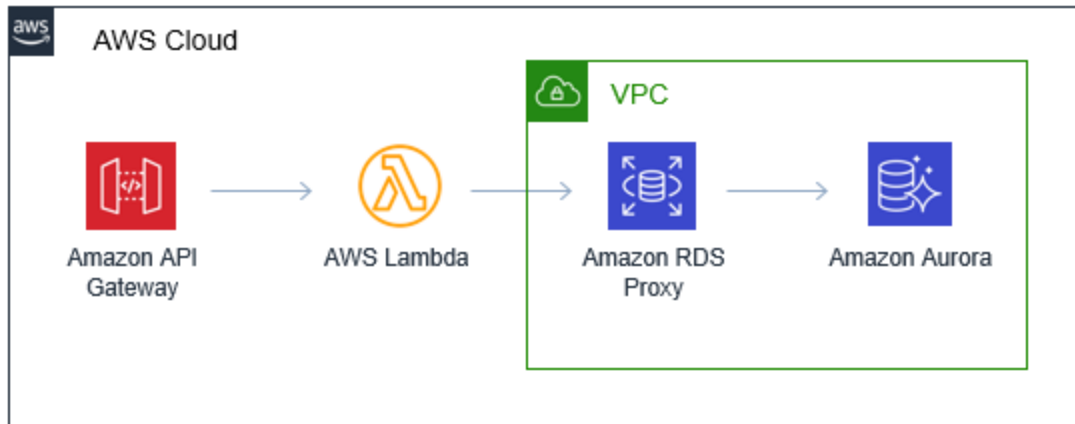
Machine consumers make automated connections to your API, which may require a different access pattern to end users. You may decide to prioritize end user consumers to provide a better experience. Machine consumers may be able to handle request throttling automatically.

Best practice: Use mechanisms to protect non-scalable resources

Limit component throughput by enforcing how many transactions it can accept

[AWS Lambda](#) functions can scale faster than traditional resources, such as relational databases and cache systems. Protect your non-scalable resources by ensuring that components that scale quickly do not exceed the throughput of downstream systems. This can prevent system performance degrading. There are a number of ways to achieve this, either directly or via buffer mechanisms such as queues and streams.

For relational databases such as [Amazon RDS](#), you can limit the number of connections per user, in addition to the global maximum number of connections. With [Amazon RDS Proxy](#), your applications can pool and share database connections to improve their ability to scale.

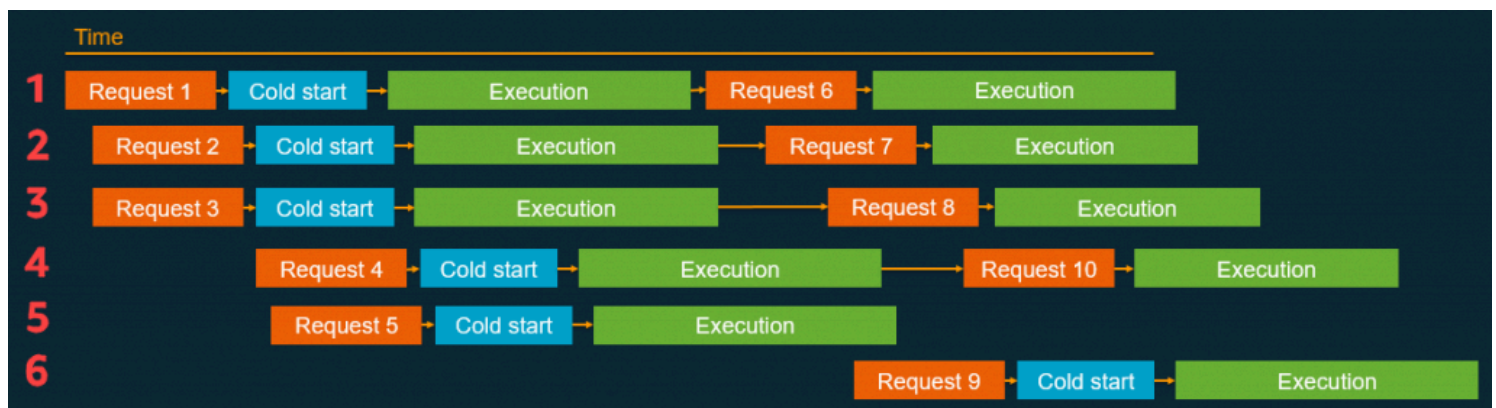


Amazon RDS Proxy

For additional options for using RDS with Lambda, see the [AWS Serverless Hero](#) blog post "[How To: Manage RDS Connections from AWS Lambda Serverless Functions](#)".

Cache results and only connect to, and fetch data from databases when needed. This reduces the load on the downstream database. Adjust the maximum number of connections for caching systems. Include a caching expiration mechanism to prevent serving stale records. For more information on caching implementation patterns and considerations, see "[Caching Best Practices](#)".

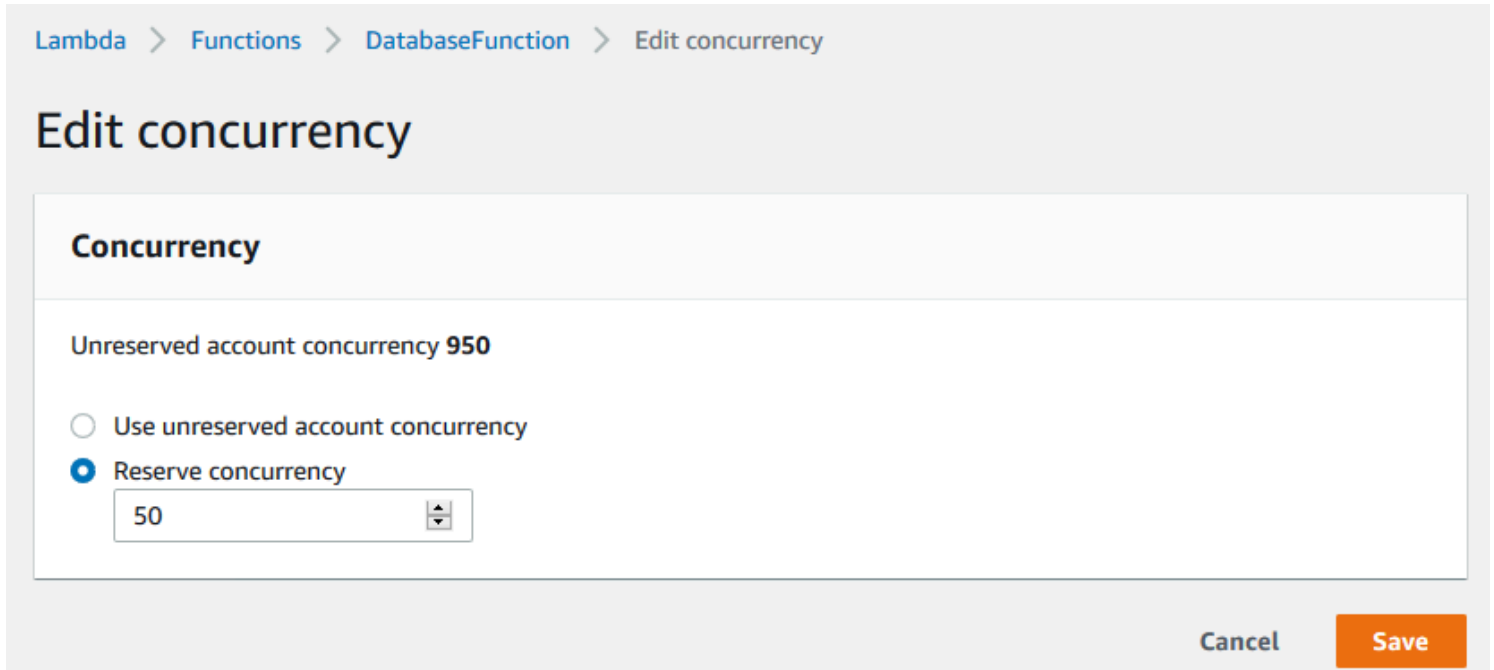
Lambda provides [managed scaling](#). When a function is first invoked, the Lambda service creates an instance of the function to process the event. This is called a cold start. After completion, the function remains available for a period of time to process subsequent events. These are called warm starts. If other events arrive while the function is busy, Lambda creates more instances of the function to handle these requests concurrently as cold starts. The following example shows 10 events processed in six concurrent requests.



Lambda concurrency

You can control the number of concurrent function invocations to both reserve and limit the maximum concurrency your function can achieve. You can configure *reserved concurrency* to set the maximum number of concurrent instances for the function. This can protect downstream resources such as a database by ensuring Lambda can only scale up to the number of connections the database can support.

For example, you may have a traditional database or external API that can only support a maximum of 50 concurrent connections. You can set the maximum number of concurrent Lambda functions using the function concurrency settings. Setting the value to 50 ensures that the traditional database or external API is not overwhelmed.



The screenshot shows the AWS Lambda console interface for editing the concurrency settings of a function named 'DatabaseFunction'. The breadcrumb navigation at the top reads 'Lambda > Functions > DatabaseFunction > Edit concurrency'. The main heading is 'Edit concurrency'. Below this, there is a section titled 'Concurrency'. Under this section, it displays 'Unreserved account concurrency 950'. There are two radio button options: 'Use unreserved account concurrency' (which is unselected) and 'Reserve concurrency' (which is selected). Below the 'Reserve concurrency' option is a text input field containing the number '50' and a small up/down arrow icon. At the bottom right of the form, there are two buttons: 'Cancel' and 'Save'.

Edit Lambda concurrency

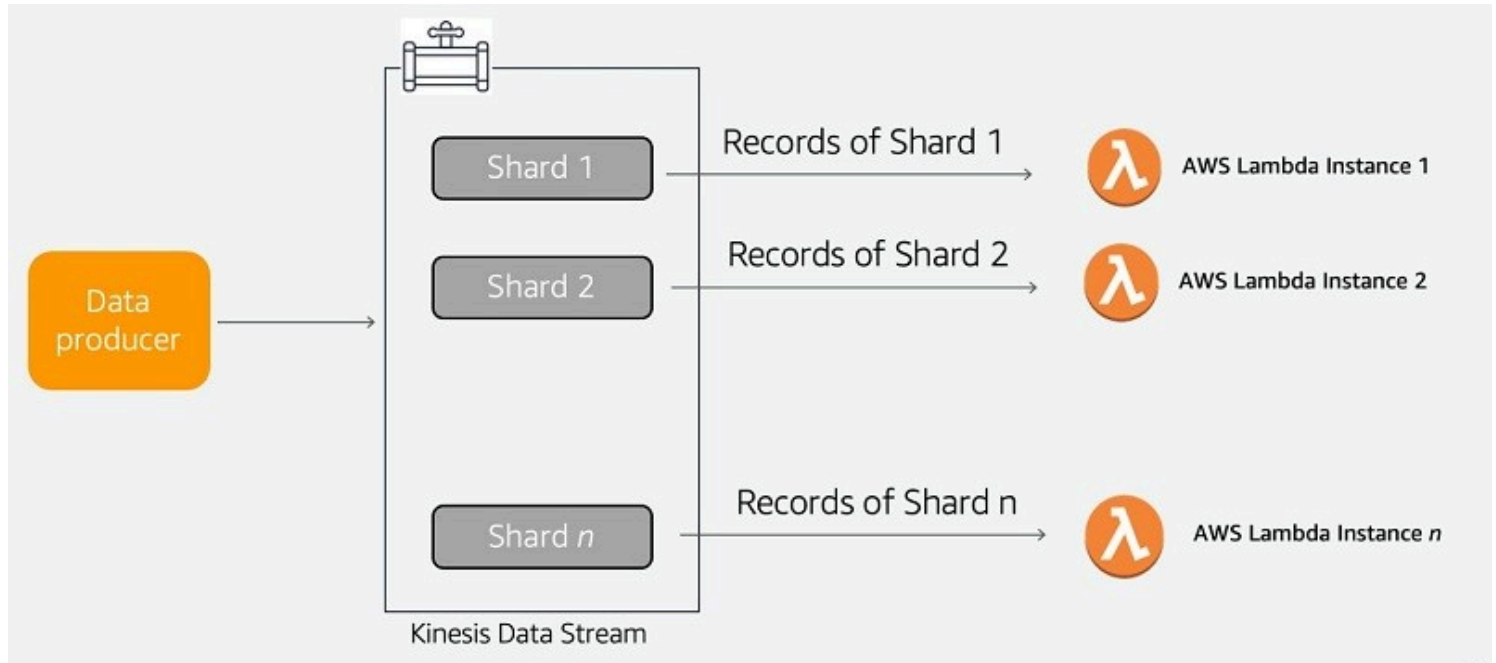
You can also set the Lambda function concurrency to 0, which disables the Lambda function in the event of anomalies.

Another solution to protect downstream resources is to use an intermediate buffer. A buffer can persistently store messages in a stream or queue until a receiver processes them. This helps you control how fast messages are processed, which can protect the load on downstream resources.

[Amazon Kinesis Data Streams](#) allows you to collect and process large streams of data records in real time, and can act as a buffer. Streams consist of a set of [shards](#) that contain a sequence of data records. When using Lambda to process records, it processes one batch of records at a time from each shard.

Kinesis Data Streams control concurrency at the shard level, meaning that a single shard has a single concurrent invocation. This can reduce downstream calls to non-scalable resources such as a traditional database. Kinesis Data Streams also support batch windows up to 5 minutes and batch record sizes. These can also be used to control how frequent invocations can occur.

To learn how to manage scaling with Kinesis, see the [documentation](#). To learn more how Lambda works with Kinesis, read the blog series “[Building serverless applications with streaming data](#)”.

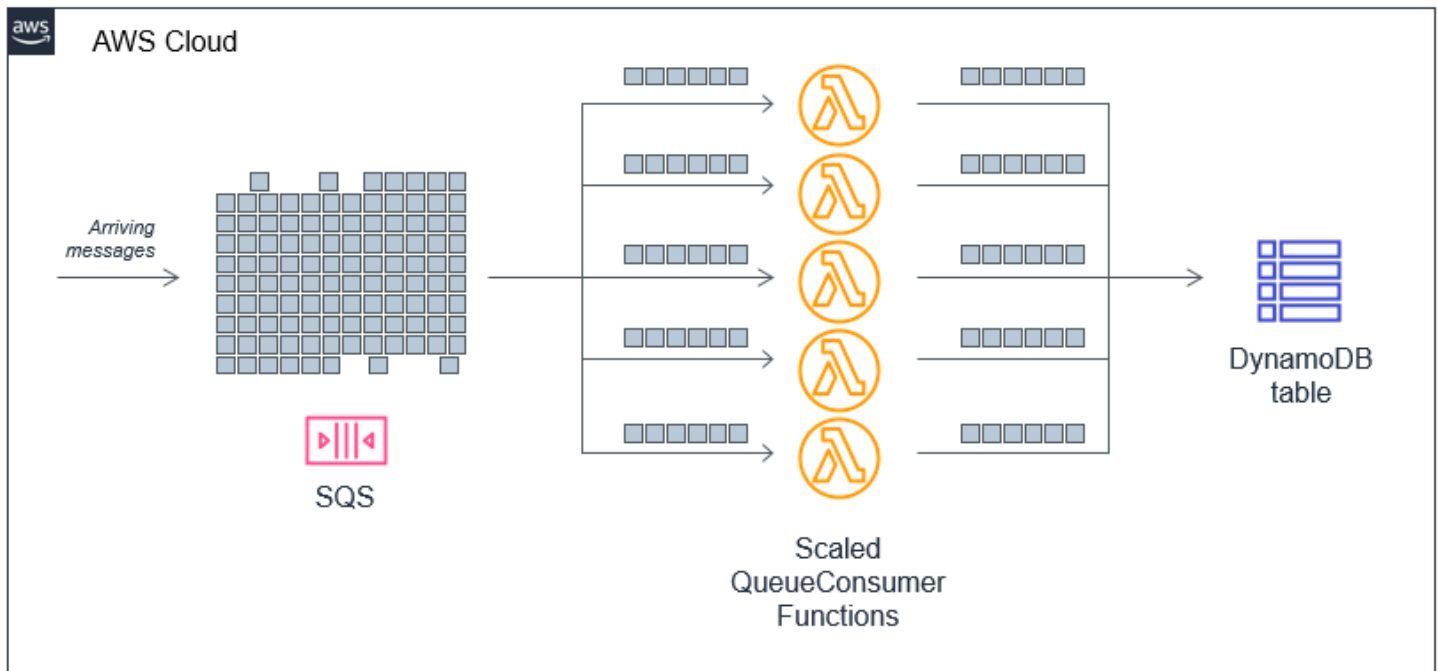


Lambda and Kinesis shards

[Amazon Simple Queue Service](#) (SQS) is a fully managed serverless message queuing service that enables you to decouple and scale microservices. You can offload tasks from one component of your application by sending them to a queue and processing them asynchronously.

SQS can act as a buffer, using a Lambda function to process the messages. Lambda polls the queue and invokes your Lambda function synchronously with an event that contains queue messages. Lambda reads messages in batches and invokes your function once for each batch. When your function successfully processes a batch, Lambda deletes its messages from the queue.

You can protect downstream resources using the Lambda concurrency controls. This limits the number of concurrent Lambda functions that pull messages off the queue. The messages persist in the queue until Lambda can process them. For more information see, “[Using AWS Lambda with Amazon SQS](#)”



Lambda and SQS

Conclusion

Regulating inbound requests helps you adapt different scaling mechanisms based on customer demand. You can achieve better throughput for your workloads and make them more reliable by controlling requests to a rate that your workload can support.

In this post, I cover using, analyzing, and enforcing API quotas using usage plans and API keys. I show mechanisms to protect non-scalable resources such as using RDS Proxy to protect downstream databases. I show how to control the number of Lambda invocations using concurrency controls to protect downstream resources. I explain how you can use streams and queues as an intermediate buffer to store messages persistently until a receiver processes them.

In the next post in the series, I cover the second reliability question from the Well-Architected Serverless Lens, building resiliency into serverless applications.

For more serverless learning resources, visit [Serverless Land](#).

TAGS: [serverless](#), [well-architected](#)