AWS Compute Blog

# Understanding AWS Lambda scaling and throughput

by Julian Wood | on 18 JUL 2022 | in Amazon CloudWatch, Auto Scaling, AWS Lambda, Serverless, Technical How-to | Permalink | Share

*Update: AWS Lambda functions now scale 12 times faster when handling high-volume requests. For more information, see the announcement post.*

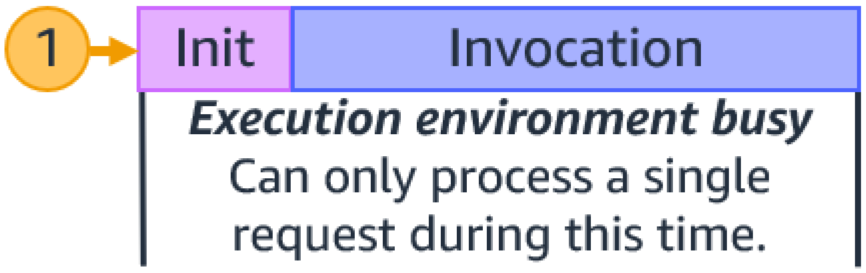
AWS Lambda provides a serverless compute service that can scale from a single request to hundreds of thousands per second. When designing your application, especially for high load, it helps to understand how Lambda handles scaling and throughput. There are two components to consider: concurrency and transactions/requests per second.

Concurrency of a system is the ability to process more than one task simultaneously. You can measure concurrency at a point in time to view how many tasks the system is doing in parallel. The number of transactions or requests a system can process per second is not the same as concurrency, because a transaction can take more or less than a second to process.

This post shows you how concurrency and transactions per second work within the Lambda lifecycle. It also covers ways to measure, control, and optimize them.

## The Lambda execution environment

Lambda invokes your code in a secure and isolated execution environment. The following shows the lifecycle of requests to a single function.

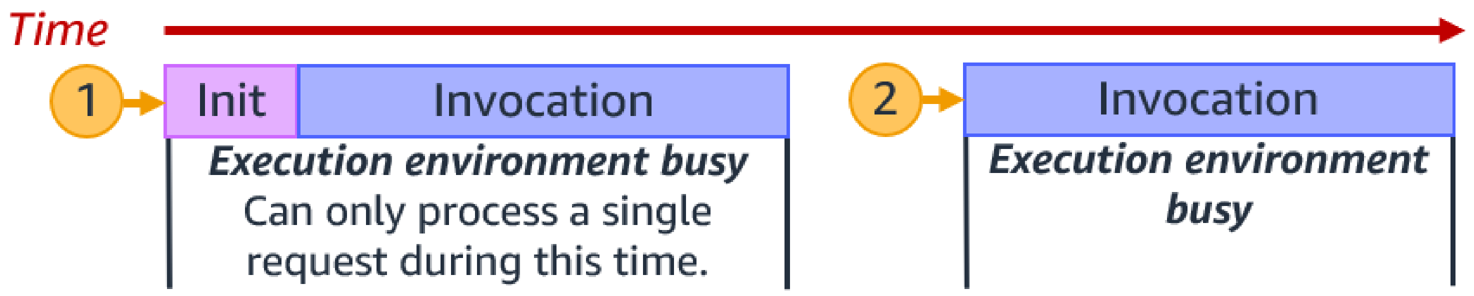

1
Init
Invocation
Execution environment busy
Can only process a single request during this time.

First request to invoke your function

At the first request to invoke your function, Lambda creates a new execution environment. It then runs the function's initialization (init) code, which is the code outside the main handler. Lambda then runs the function handler code as the invocation. This receives the event payload and processes your business logic.

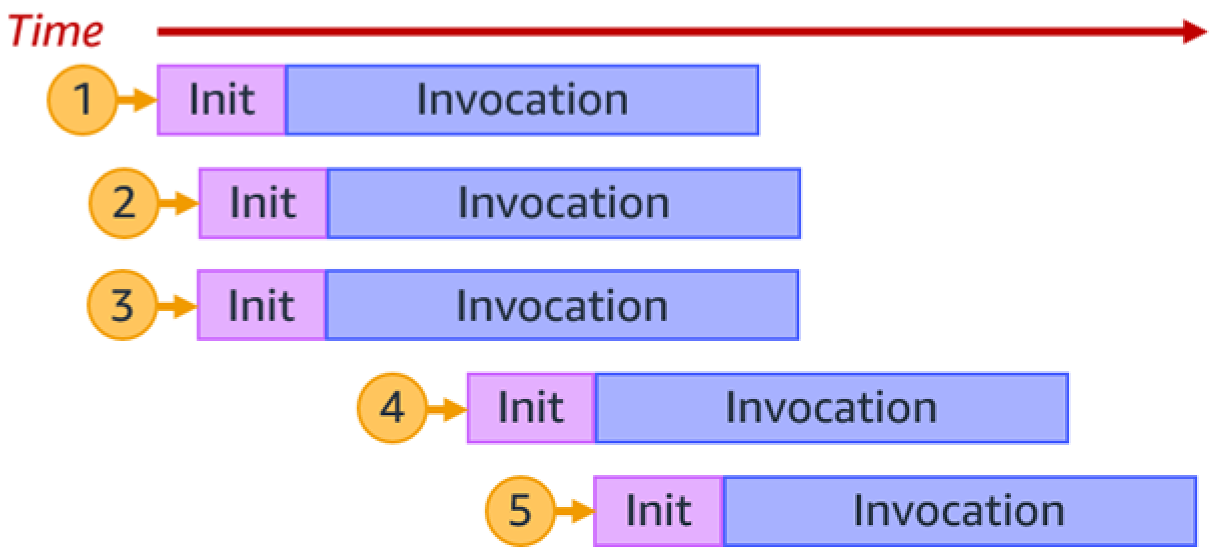
Each execution environment processes a single request at a time. While a single execution environment is processing a request, it cannot process other requests.

After Lambda finishes processing the request, the execution environment is ready to process an additional request for the same function. As the initialization code has already run, for request 2, Lambda runs only the function handler code as the invocation.


Time
1
Init
Invocation
Execution environment busy
Can only process a single request during this time.
2
Invocation
Execution environment busy

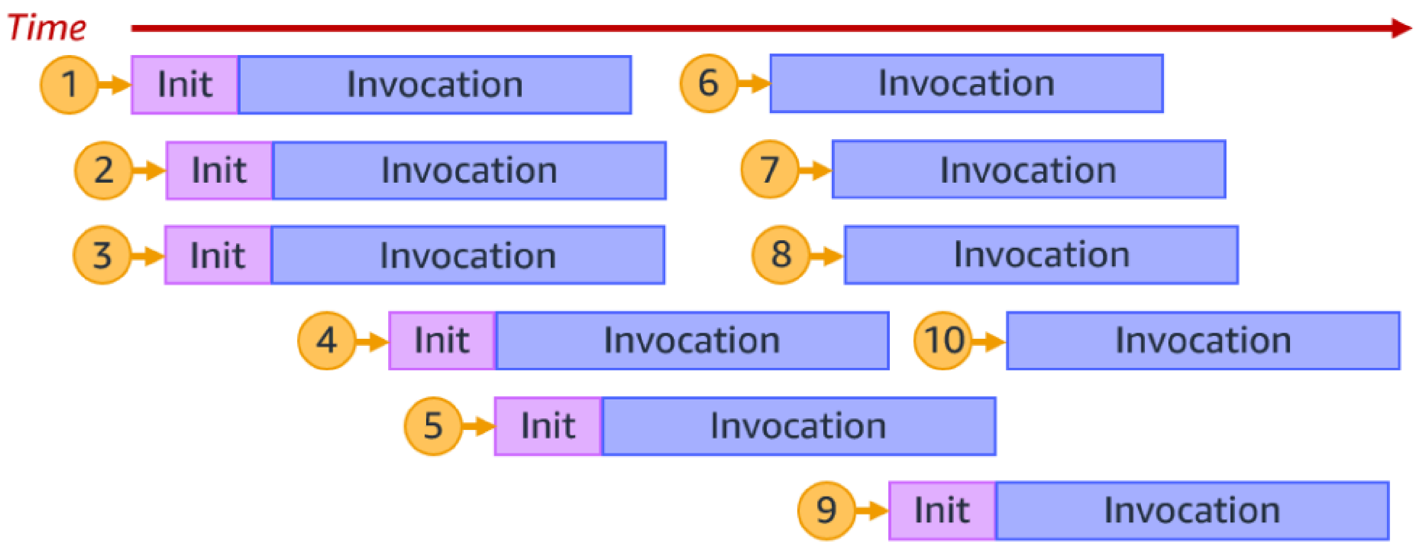
Lambda ready to process an additional request for the same function

If additional requests arrive while processing request 1, Lambda creates new execution environments. In this example, Lambda creates new execution environments for requests 2, 3, 4, and 5. It runs the function init and the invocation.



Lambda creating new execution environments

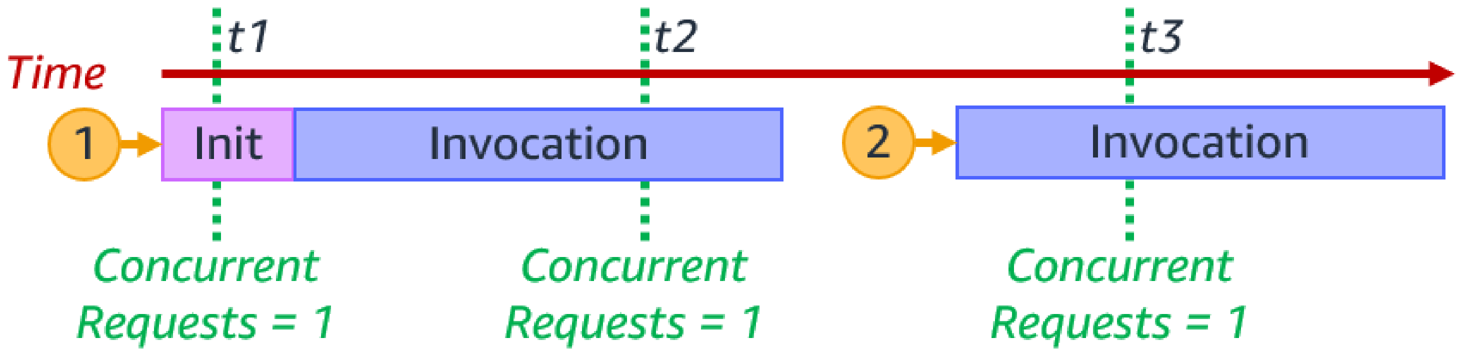
When requests arrive, Lambda reuses available execution environments, and creates new ones if necessary. The following example shows the behavior for additional requests after 1-5.



Lambda reusing execution environments

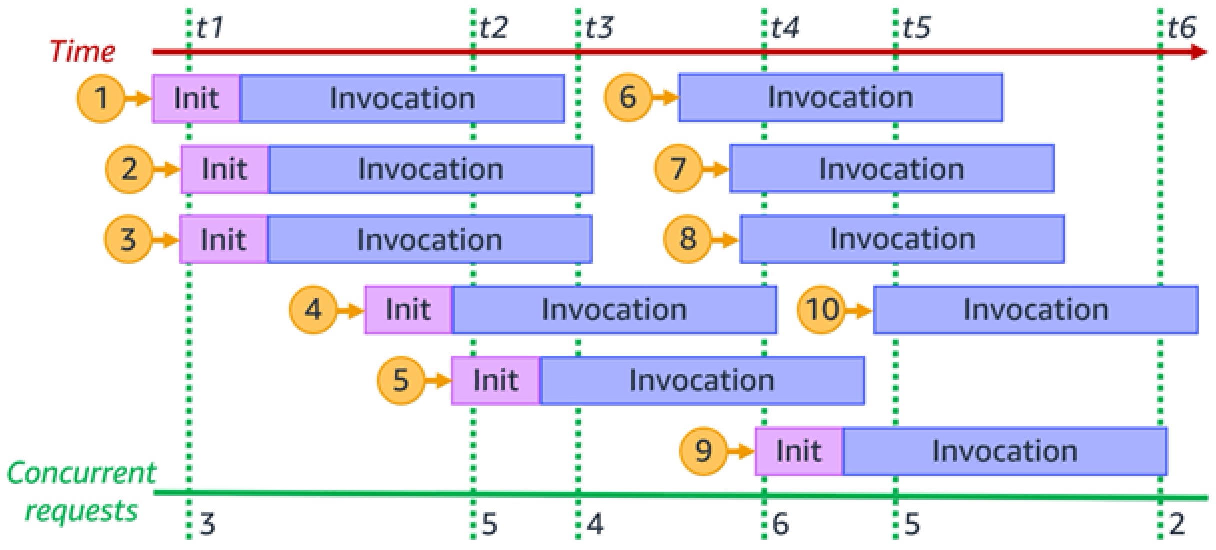
Once request 1 completes, the execution environment is available to process another request. When request 6 arrives, Lambda re-uses request 1s execution environment and runs the invocation. This process continues for requests 7 and 8, which reuse the execution environments from requests 2 and 3. When request 9 arrives, Lambda creates a new execution environment as there isn't an existing one available. When request 10 arrives, it reuses the execution environment freed up after request 4.

The number of execution environments determines the concurrency. This is the sum of all concurrent requests for currently running functions at a particular point in time. For a single execution environment, the number of concurrent requests is 1.



Single execution environment concurrent requests

For the example with requests 1–10, the Lambda function concurrency at particular times is the following:



Lambda concurrency at particular times

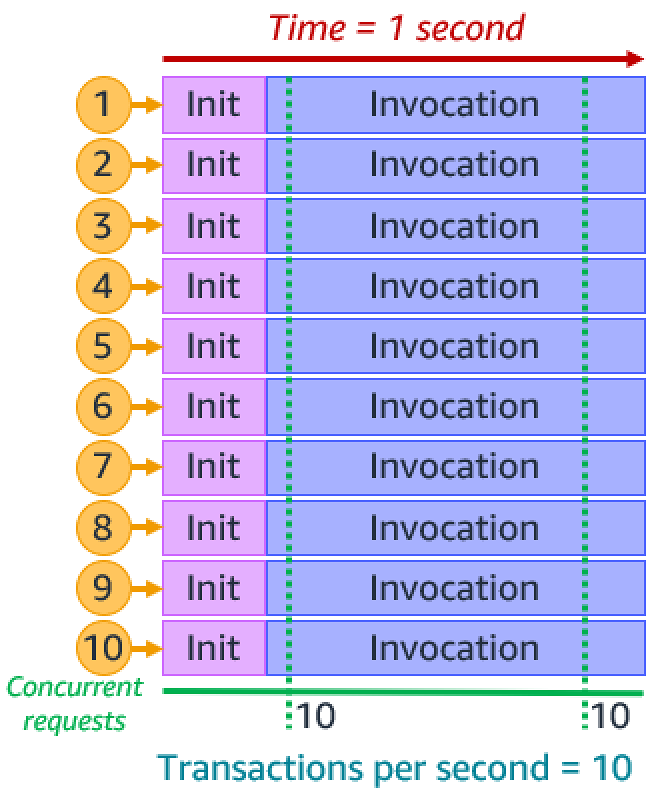
| time | concurrency |
|---|---|
| *t1* | 3 |
| *t2* | 5 |
| *t3* | 4 |
| *t4* | 6 |
| *t5* | 5 |

| | |
|---|---|
| *t6* | 2 |

When the number of requests decreases, Lambda stops unused execution environments to free up scaling capacity for other functions.

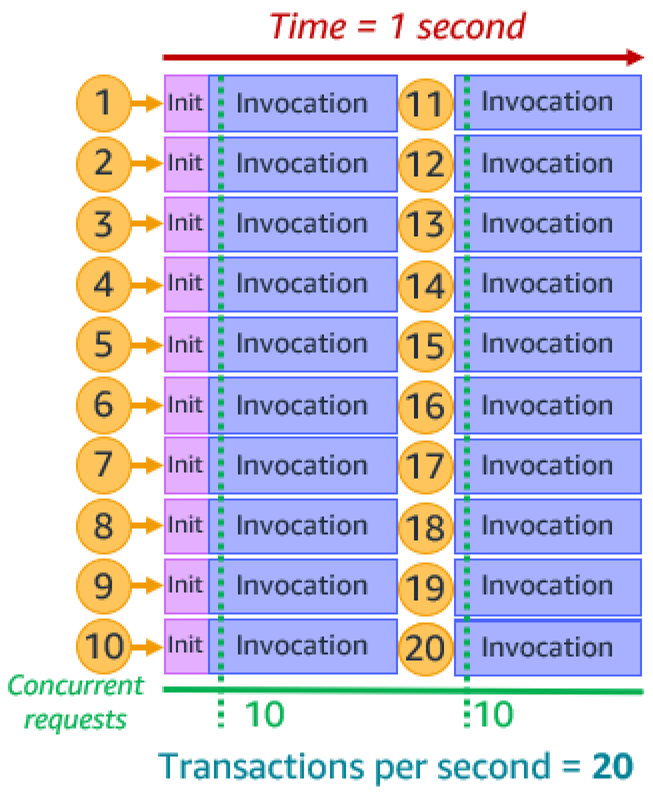
## Invocation duration, concurrency, and transactions per second

The number of transactions Lambda can process per second is the sum of all invokes for that period. If a function takes 1 second to run, and 10 invocations happen concurrently, Lambda creates 10 execution environments. In this case, Lambda processes 10 requests per second.



Lambda transactions per second for 1 second invocation

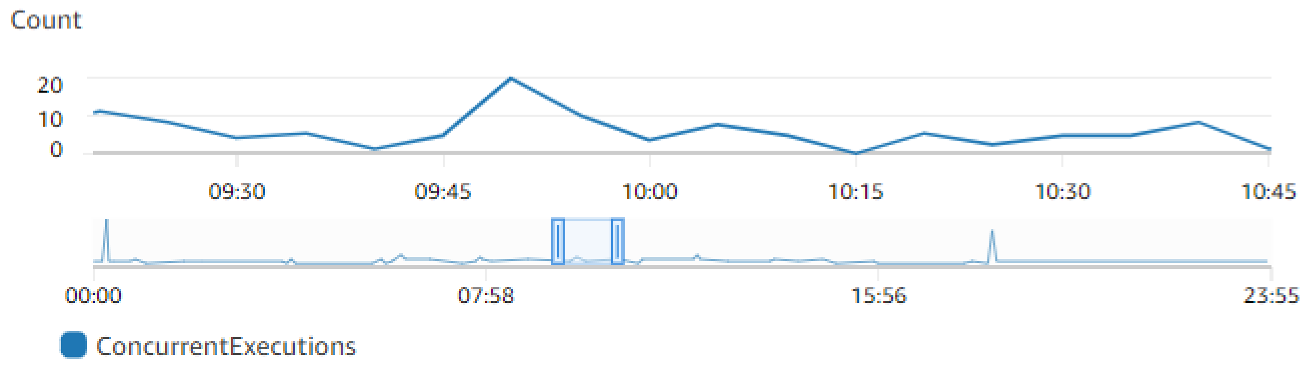
If the function duration is halved to 500 ms, the Lambda concurrency remains 10. However, transactions per second are now 20.



Lambda transactions per second for 500 ms second invocation

If the function takes 2 seconds to run, during the initial second, transactions per second is 0. However, averaged over time, transactions per second is 5.

You can view concurrency using [Amazon CloudWatch metrics](). Use the metric name *ConcurrentExecutions* to view concurrent invocations for all or individual functions.



CloudWatch metrics ConcurrentExecutions

You can also estimate concurrent requests from the number of requests per unit of time, in this case seconds, and their average duration, using the formula:

*RequestsPerSecond* x *AvgDurationInSeconds* = *concurrent requests*

If a Lambda function takes an average 500 ms to run, at 100 requests per second, the number of concurrent requests is 50:

*100 requests/second x 0.5 sec = 50 concurrent requests*

If you half the function duration to 250 ms and the number of requests per second doubles to 200 requests/second, the number of concurrent requests remains the same:

*200 requests/second x 0.250 sec = 50 concurrent requests.*

Reducing a function's duration can increase the transactions per second that a function can process. For more information on reducing function duration, watch this re:Invent video.

## Scaling quotas

There are two scaling quotas to consider with concurrency. Account concurrency quota and burst concurrency quota.

Account concurrency is the maximum concurrency in a particular Region. This is shared across all functions in an account. The default Regional concurrency quota starts at 1,000, which you can increase with a service ticket.

*Update: AWS Lambda functions now scale 12 times faster when handling high-volume requests. For more information, see the announcement post.*

The burst concurrency quota provides an initial burst of traffic, between 500 and 3000 per minute, depending on the Region. This is also shared across all function in an account.

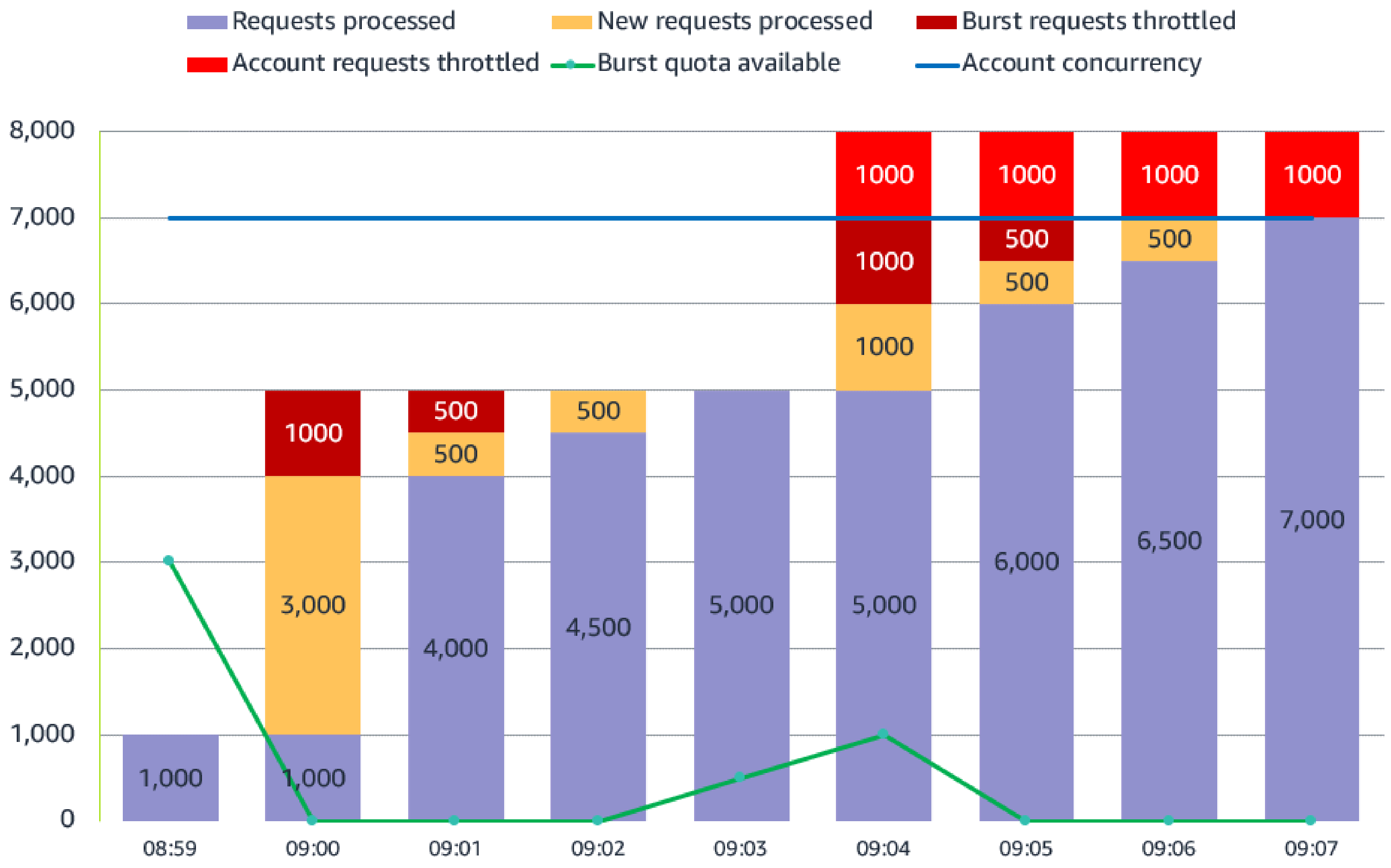
After this initial burst, functions can scale by another 500 concurrent invocations per minute for all Regions. If you reach the maximum number of concurrent requests, further requests are throttled.

For synchronous invocations, Lambda returns a throttling error (429) to the caller, which must retry the request. With asynchronous and event source mapping invokes, Lambda automatically retries the requests. See Error handling and automatic retries in AWS Lambda for more detail.

### A scaling quota example

The following walks through how account and burst concurrency work for an example application.

Anticipating handling additional load, the application builders have raised account concurrency to 7,000. There are no other Lambda functions running in this account, so this function can use all available account concurrency.

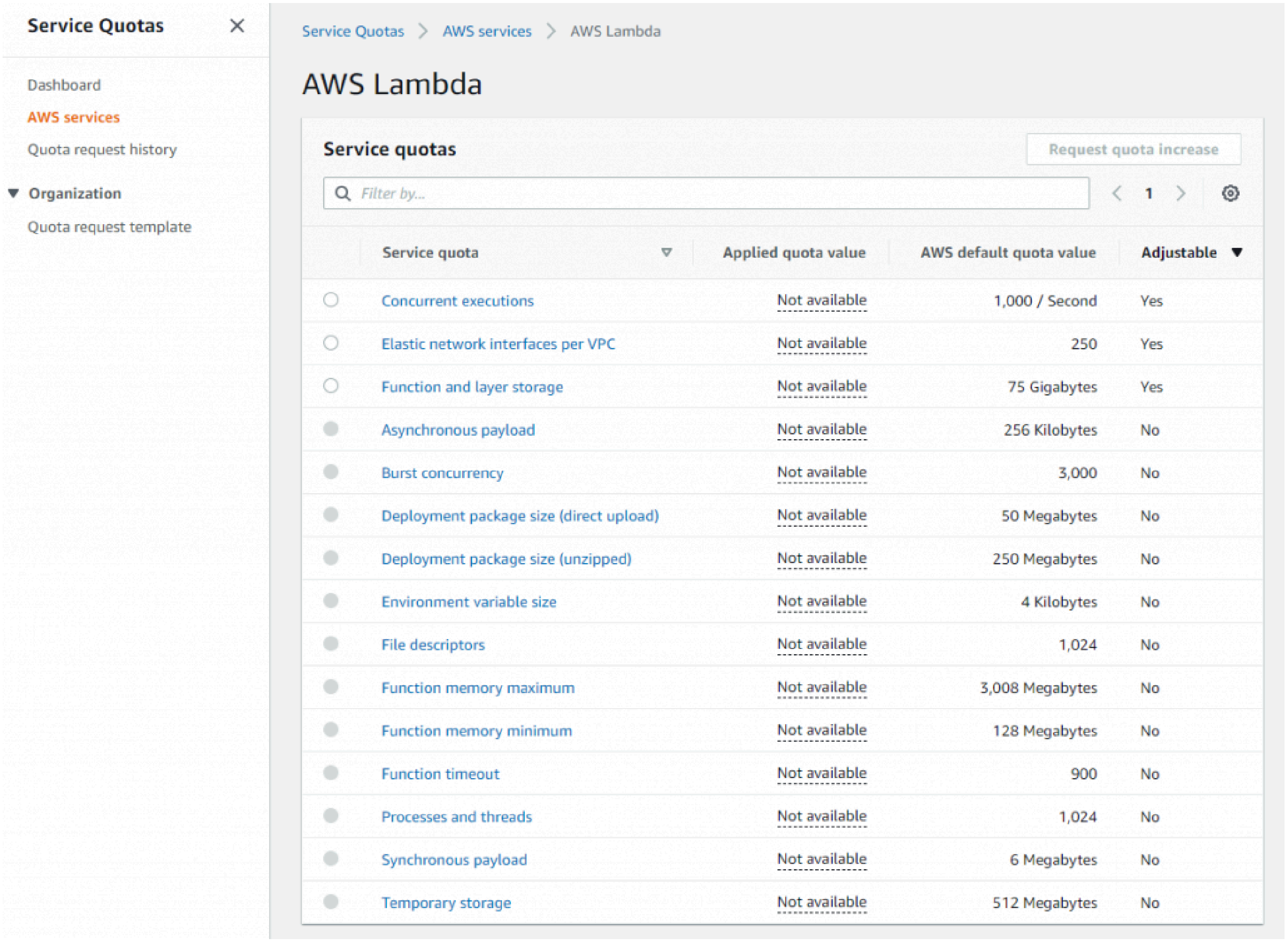


Lambda account and burst concurrency

1. **08:59:** The application already has a steady stream of requests, using 1,000 concurrent execution environments. Each Lambda invocation takes 250 ms, so transactions per second are 4,000.
2. **09:00:** There is a large spike in traffic at 5,000 sustained requests. 1000 requests use the existing execution environments. Lambda uses the 3,000 available burst concurrency to create new environments to handle the additional load. 1,000 requests are throttled as there is not enough burst concurrency to handle all 5,000 requests. Transactions per second are 16,000.
3. **09:01:** Lambda scales by another 500 concurrent invocations per minute. 500 requests are still throttled. The application can now handle 4,500 concurrent requests.
4. **09:02:** Lambda scales by another 500 concurrent invocations per minute. No requests are throttled. The application can now handle all 5,000 requests.
5. **09:03:** The application continues to handle the sustained 5000 requests. The burst concurrency quota rises to 500.
6. **09:04:** The application sees another spike in traffic, this time unexpected. 3,000 new sustained requests arrive, a combination of 8,000 requests. 5,000 requests use the existing execution environments. Lambda uses the now available burst concurrency of 1,000 to create new environments to handle the additional load. 1,000 requests are throttled as there is not enough burst concurrency. Another 1,000 requests are throttled as the account concurrency quota has been reached.

7. **09:05:** Lambda scales by another 500 concurrent requests. The application can now handle 6,500 requests. 500 requests are throttled as there is not enough burst concurrency. 1,000 requests are still throttled as the account concurrency quota has been reached.
8. **09:06:** Lambda scales by another 500 concurrent requests. The application can now handle 7,000 requests. 1,000 requests are still throttled as the account concurrency quota has been reached.
9. **09:07:** The application continues to handle the sustained 7,000 requests. 1,000 requests are still throttled as the account concurrency quota has been reached. Transactions per second are 28,000.

Service Quotas is an AWS service that helps you manage your quotas for many AWS services. Along with looking up the values, you can also request a limit increase from the Service Quotas console.



AWS Lambda

Service quotas

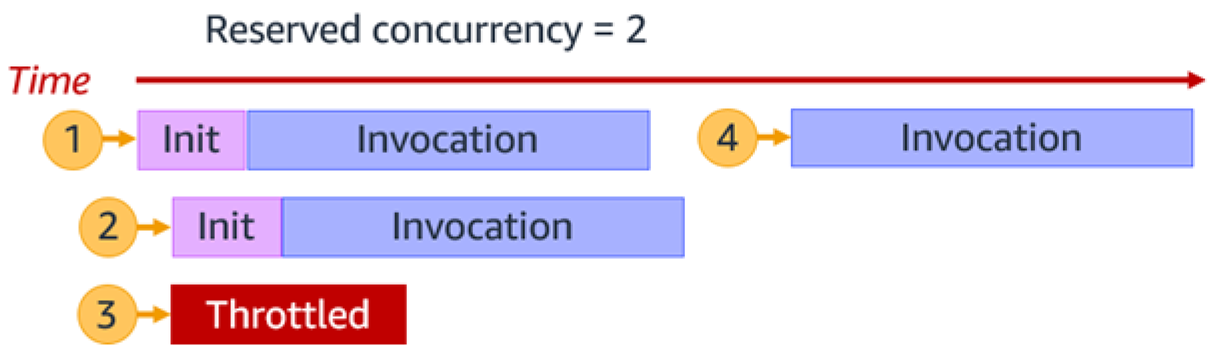
| Service quota | Applied quota value | AWS default quota value | Adjustable |
|---|---|---|---|
| Concurrent executions | Not available | 1,000 / Second | Yes |
| Elastic network interfaces per VPC | Not available | 250 | Yes |
| Function and layer storage | Not available | 75 Gigabytes | Yes |
| Asynchronous payload | Not available | 256 Kilobytes | No |
| Burst concurrency | Not available | 3,000 | No |
| Deployment package size (direct upload) | Not available | 50 Megabytes | No |
| Deployment package size (unzipped) | Not available | 250 Megabytes | No |
| Environment variable size | Not available | 4 Kilobytes | No |
| File descriptors | Not available | 1,024 | No |
| Function memory maximum | Not available | 3,008 Megabytes | No |
| Function memory minimum | Not available | 128 Megabytes | No |
| Function timeout | Not available | 900 | No |
| Processes and threads | Not available | 1,024 | No |
| Synchronous payload | Not available | 6 Megabytes | No |
| Temporary storage | Not available | 512 Megabytes | No |

Service Quotas console

## Reserved concurrency

You can configure a *Reserved concurrency* setting for your Lambda functions to allocate a maximum concurrency limit for a function. This assigns part of the account concurrency quota to a specific function. This protects and ensures that a function is not throttled and can always scale up to the reserved concurrency value.

It can also help protect downstream resources. If a database or external API can only handle 2 concurrent connections, you can ensure Lambda can't scale beyond 2 concurrent invokes. This ensures Lambda doesn't overwhelm the downstream service. You can also use reserved concurrency to avoid race conditions to ensure that your function can only run one concurrent invocation at a time.



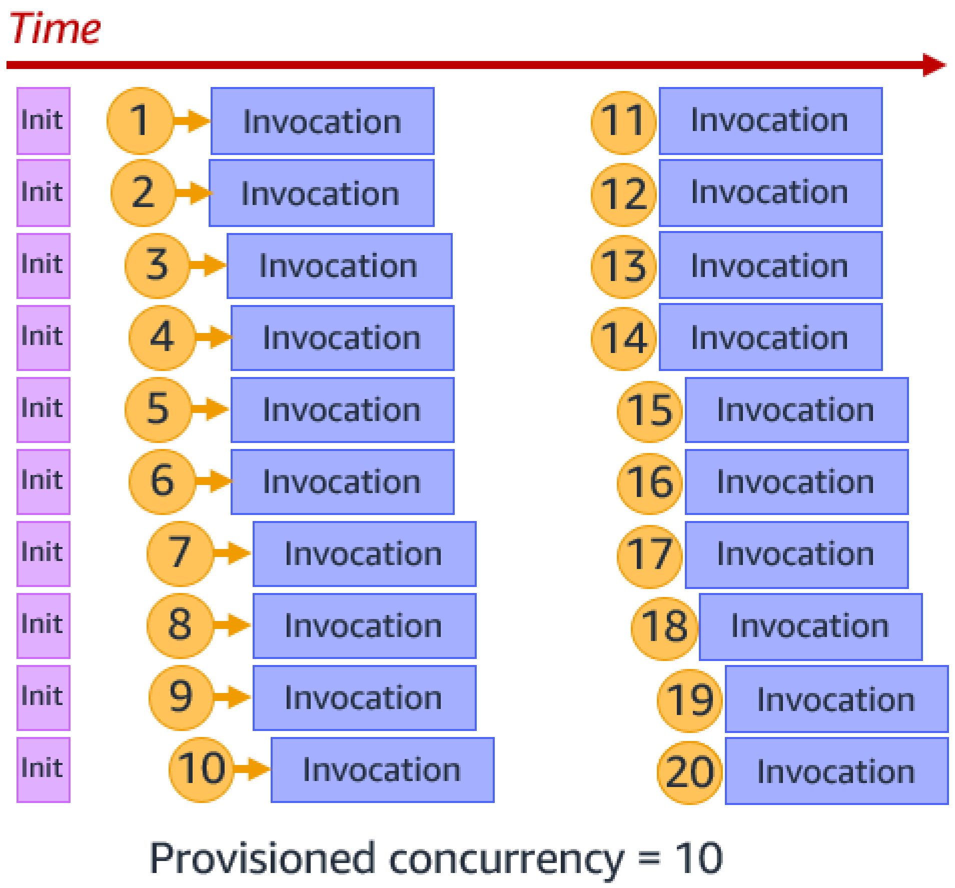
Reserved concurrency

You can also set the function concurrency to zero, which stops any function invocations and acts as an off-switch. This can be useful to stop Lambda invocations when you have an issue with a downstream resource. It can give you time to fix the issue before removing or increasing the concurrency to allow invocations to continue.

## Provisioned Concurrency

The function initialization process can introduce latency for your applications. You can reduce this latency by configuring Provisioned Concurrency for a function version or alias. This prepares execution environments in advance, running the function initialization process, so the function is ready to invoke when needed.

This is primarily useful for synchronous requests to ensure you have enough concurrency before an expected traffic spike. You can still burst above this using standard concurrency. The following example shows *Provisioned Concurrency* configured as 10. Lambda runs the init process for 10 functions, and then when requests arrive, immediately runs the invocation.



Provisioned Concurrency

You can use Application Auto Scaling to adjust Provisioned Concurrency automatically based on Lambda's utilization metric.

## Operational metrics

There are CloudWatch metrics available to monitor your account and function concurrency to ensure that your applications can scale as expected. Monitor function *Invocations* and *Duration* to understand throughput. *Throttles* show throttled invocations.

*ConcurrentExecutions* tracks the total number of execution environments that are processing events. Ensure this doesn't reach your account concurrency to avoid account throttling. Use the metric for individual functions to see which are using account concurrency, and also ensure reserved concurrency is not too high. For example, a function may have a reserved concurrency of 2000, but is only using 10.

*UnreservedConcurrentExecutions* show the number of function invocations without reserved concurrency. This is your available account concurrency buffer.

Use *ProvisionedConcurrencyUtilization* to ensure you are not paying for Provisioned Concurrency that you are not using. The metric shows the percentage of allocated Provisioned Concurrency in use.

*ProvisionedConcurrencySpilloverInvocations* show function invocations using standard concurrency, above the configured Provisioned Concurrency value. This may show that you need to increase Provisioned Concurrency.

Use *ClaimedAccountConcurrency* to track overall account concurrency utilization and monitor when your account is reaching your account limit. This metric is the sum of *UnreservedConcurrentExecution* and allocated concurrency (reserved, provisioned). To learn more about how to use this metric, refer to the documentation.

## Conclusion

Lambda provides a highly scalable compute service. Understanding how Lambda scaling and throughput works can help you design your application, especially for high load.

This post explains concurrency and transactions per second. It shows how account and burst concurrency quotas work. You can configure reserved concurrency to ensure that your functions can always scale, and also use it to protect downstream resources. Use Provisioned Concurrency to scale up Lambda in advance of invokes.

For more serverless learning resources, visit Serverless Land.
TAGS: serverless