ISSN: 2634 - 8853

Journal of Engineering and Applied Sciences Technology

SCIENTIFIC Research and Community

Review Article

Open Access

Achieving Concurrency Control: Practical Throttling Techniques for AWS Lambda

Balasubrahmanya Balakrishna

Senior Lead Software Engineer, Richmond, VA, USA

ABSTRACT

This technical paper explores nuanced approaches for managing concurrency in Lambda functions, shedding light on the intricacies of AWS services like SQS and Kinesis. It underscores the significance of reserved concurrency to regulate function execution, acknowledging its effectiveness while cautioning about its limitations, particularly in scenarios requiring scalable solutions. The focus extends to the advantages of SQS as a pull-based service, emphasizing its built-in concurrency control and dynamic scalability capabilities.

The paper then delves further into Kinesis data streams, showcasing the distinctive architecture of shard-based scaling and introducing parallelization factors for precise control over Lambda concurrency. Notably, these concurrency control strategies are presented as a means to scale Lambda and as a mechanism to manage traffic downstream, considering potential technical constraints.

The discussion highlights multiple factors in selecting an appropriate messaging service: cost, message replay capability, error-handling mechanisms, message processing order, and concurrency management. Collectively, these insights serve as a comprehensive reference for architects and developers striving to create efficient and scalable serverless applications within the AWS environment, addressing Lambda scalability and downstream traffic control challenges.

***Corresponding author**

Balasubrahmanya Balakrishna, Senior Lead Software Engineer, Richmond, VA, USA.

Received: April 04, 2023; **Accepted:** April 15, 2023; **Published:** April 22, 2023

Keywords: Reserved Concurrency, Lambda Concurrency Control, SQS, SNS, Kinesis, AWS Lambda function, Event Source Mapping

Introduction

Designed for scalability, AWS Lambda incorporates built-in auto-scaling capabilities. Because of its intrinsic scalability, it is ideal for real-time message-processing scenarios. For example, it is useful when the goal is to distribute messages to many workers, hence increasing our system's throughput.

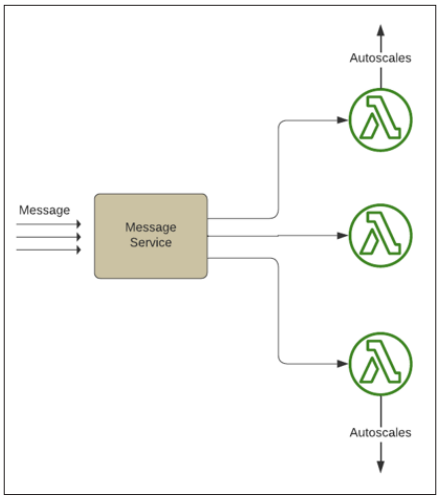

Figure 1: Lambda and AutoScaling

If we need to communicate with a downstream, as shown in Figure 2, a system that has scalability limitations is susceptible to increased loads and has the potential to cause system outages.

Also, we must restrict the concurrent request to downstream service for any reason. In that case, the question becomes: How can we systematically govern the concurrency of our Lambda function to avoid such issues?

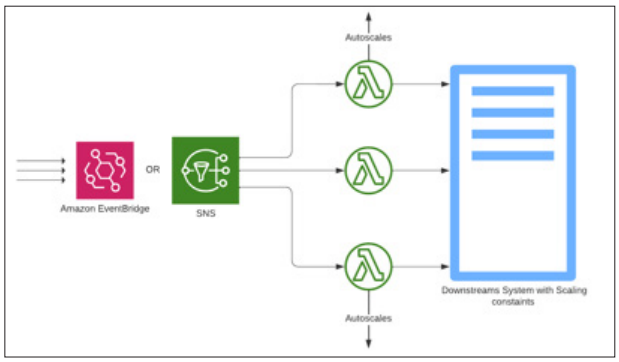

Figure 2: Lambda Interacting with Constraint Downstream

Background

A regional concurrency restriction governs Lambda, dictating the maximum number of concurrent instances of Lambda functions, as shown in Figure 3. Regional concurrency soft limit increase can happen by raising a service ticket with AWS.

Each function invocation depletes one unit of concurrency for the duration of the event. When many Lambda invocations run concurrently, all available concurrency units can be exhausting. As a result, each subsequent invocation that exceeds the available concurrent limitation will be throttled, posing a severe problem, particularly for Lambda functions deemed mission-critical.

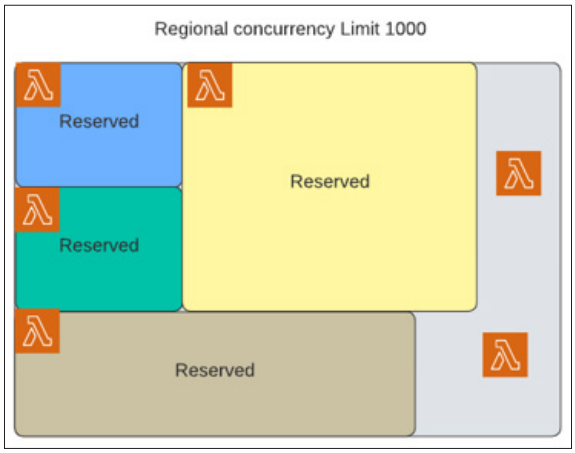

Figure 3: Regional Concurrency Pool and Lambda with Reserved Concurrency

Fine-Tune AWS Lambda Concurrency: Effective Control Strategies

Concurrency Control Through Reserved Concurrency

The most straightforward method for controlling the concurrency of a Lambda function uses the built-in reserved concurrency mechanism.

Reserved concurrency designates a portion of regional concurrency units for a specific function. This allocation ensures that the specified function always has a reserved concurrency capacity for execution. However, it is crucial to note that these reserved concurrency units are withdrawn from the regional pool, reducing the overall concurrency resources available to other services when they require scaling on demand.

Notably, reserved concurrency serves a dual purpose as it also acts as the maximum concurrency threshold, enabling the throttling of concurrent executions of a Lambda function. While setting reserved concurrency is a fundamental and economical approach, it has the following limitations:

1. Deploying reserved concurrency across many functions, in particular, generates incremental consumption of available regional concurrency units. This cumulative effect can reduce the regional pool of concurrency resources.
2. Over time, functions requiring dynamic or on-demand scalability might encounter issues with insufficient concurrency. Inefficiently distributing concurrent units can lead to throttling several invocations, as depicted in Figure 4. Reserved concurrency works well for isolated concerns, and improving scalability becomes crucial as a growing number of functions need concurrency control.

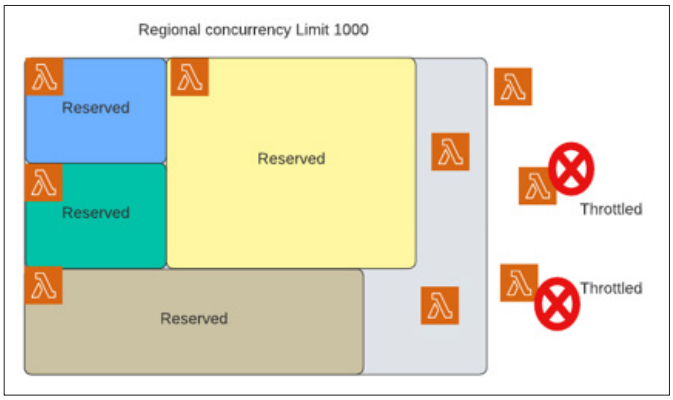

Figure 4: Regional Concurrency Pool, Lambda with Reserved Concurrency and On-demand Traffic Throttling

3. Reserved Concurrency for Lambdas places in a situation that necessitates meticulous oversight of Lambda agreements, an undesirable responsibility likely to be challenging to execute effectively.

Other Approaches to Control Concurrency

Premise

The preferred strategy would be to take advantage of the natural scaling characteristics of Lambda and its event sources. Specific event sources, particularly those requiring polling techniques like SQS and Kinesis, can be configured via event source mapping. These sources have integrated concurrency control techniques, which eliminates the need for human management.

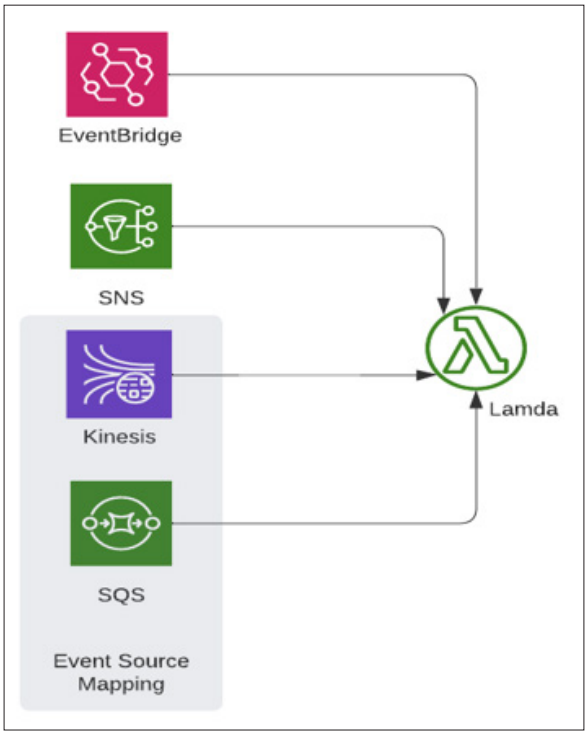

Figure 5: Messaging Service Interaction with Lambda

Concurrency Control: AWS SNS and AWS EventBridge

When asynchronous event sources like SNS and EventBridge receive a message, they immediately attempt to activate a subscriber function [1]. As a result, the function's concurrency increases linearly in tandem with the rate of incoming messages, as depicted in Figure 5 and Figure 6. Assigning reserved concurrency to the function can alter this trajectory.

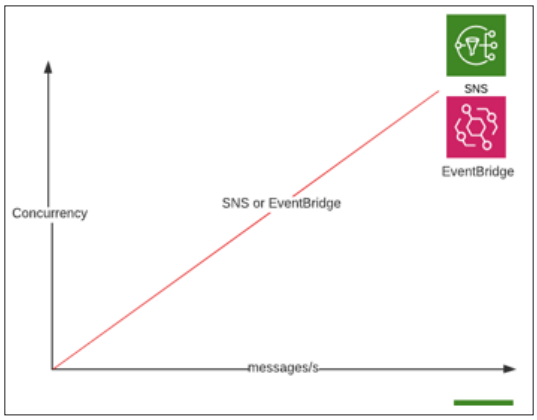

Figure 6: Lambda Concurrency Vs Messages/s for SNS and EventBridge

Concurrency Control: AWS SQS

Conversely, in scenarios where SNS or EventBridge is not mandatory, SQS emerges as a compelling alternative for message ingestion and processing with Lambda [2]. SQS requiring fewer invocations to process an equivalent number of messages, coupled with its inherent concurrency control features, underscores this preference.

Notably, SQS, being a pull-based service, leverages a cluster of pollers managed by the Lambda service [3]. These pollers actively retrieve messages from the queue and transmit them to the function in batch operations. Upon completion of the function invocation, the poller removes the processed messages from the queue. The initial configuration starts with five pollers, facilitating up to 5 concurrent function invocations. Depending on the backlog of messages, the Lambda service dynamically scales the pollers, incrementing by up to 60 instances per minute, with a maximum concurrency cap set at 1000. Each poller independently invokes the associated function, establishing a 1-to-1 mapping between the number of pollers and concurrent function invocations, as shown in Figure 7.

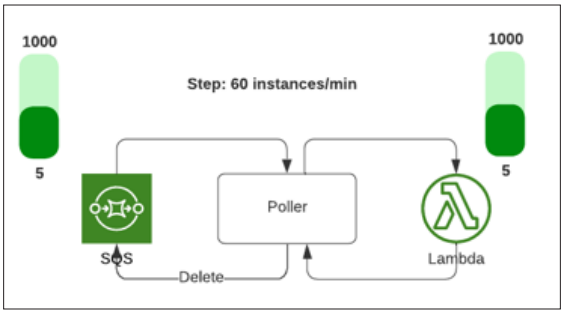

Figure 7: Concurrency Relation: SQS and Lambda

The built-in batching method limit the scaling capacity of pollers to an increment of 60 instances per minute. As a result, in contrast to the quick scaling witnessed in SNS and EventBridge cases, the concurrent augmentation for the function is more gradual, as seen in Figure 8.

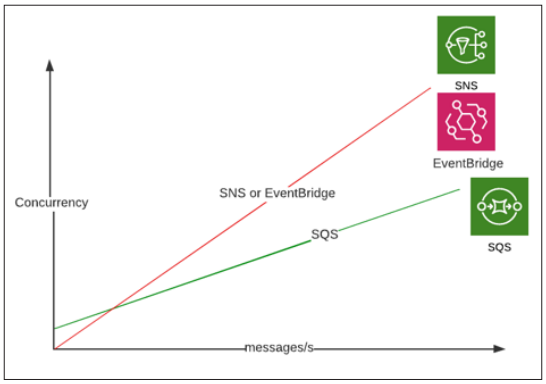

Figure 8: Concurrency Vs. Messages/s Comparison: SNS, EventBridge, and SQS

Configuring the *MaximumConcurrency* parameter in the event source mapping under ScalingConfig allows the set of an upper limit on poller concurrency. This parameter is associated with an integer value representing the specified maximum concurrency. This setting is a practical approach to overseeing an SQS function's concurrency management.

```
aws lambda update-event-source-mapping \
  --uuid "a1b2c3d4-5678-99ab-cdef-11111FOOBAR" \
  --scaling-config '{"MaximumConcurrency":5}'
```

Figure 9: AWS CLI to set MaximumConcurrency

Because of the possible issue known as SQS over-polling, it is not advised to use reserved concurrency in conjunction with an SQS function. This problem develops when there is a mismatch between the function's reserve concurrency setting and the poller's concurrency. Exceeding the limit imposed by the reserve concurrency setting will result in throttling invocations, as shown in Figure 10.

SQS messages are returned to the queue via throttling without being processed by the relevant function. Moreover, the system passes certain messages to the dead letter queue without the function processing them if a dead letter queue (DLQ) is configured and throttling continues. This issue could be mitigated by MaximumConcurrency, setting on the event source mapping, not on the Lambda function.

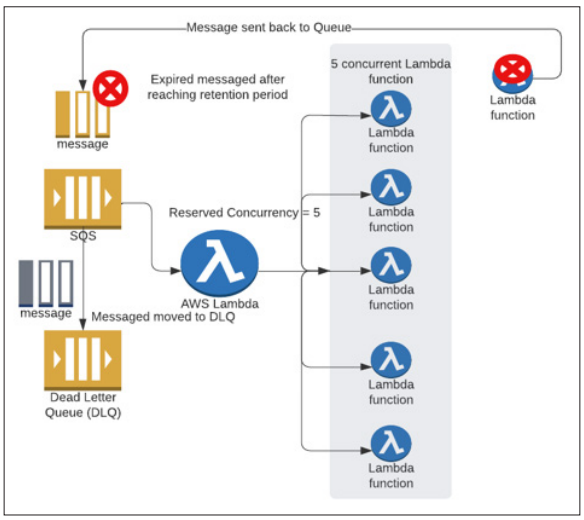

Figure 10: SQS Overpolling

Concurrency Control: AWS Kinesis Data Streams

To wrap up our investigation, let us look at Kinesis data streams, a polling-based service similar to SQS that allows batching [4]. The Lambda service, like SQS, orchestrates a cluster of pollers on our behalf at no extra expense.

Increasing the number of shards in the data stream distinguishes Kinesis due to its scalability. A poller is assigned to each shard, establishing a 1-to-1 ratio between shards and pollers, as shown in Figure 11. Each poller individually invokes a function, ensuring a 1-to-1 ratio between the number of shards and concurrent Lambda invocations. This architecture highlights a clear link between the scaling of Kinesis data streams and the concurrent execution of Lambda functions.

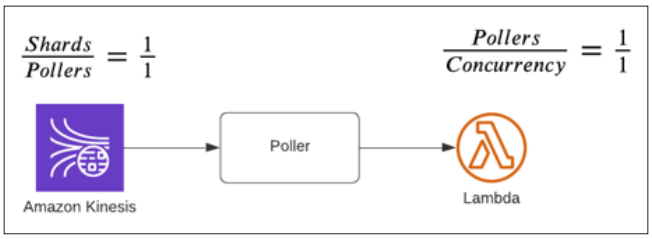

Figure 11: Concurrency Relation: Kinesis and Lambda

However, it is essential to clarify this assumption-*ParallelizationFactor* within the event source mapping for a Kinesis function, ranging from 1 to 10. This factor dictates that pollers process messages from the same shard simultaneously. Importantly, this rule applies while preserving the ordering of messages based on the partition key.

```
aws lambda create-event-source-mapping -
-function-name my-function \
--parallelization-factor 2 --batch-size 50 -
-starting-position AT_TIMESTAMP -
-starting-position-timestamp 1541139178 \
--event-source-arn arn:aws:kinesis:us-east-
2:123456789099:stream/lambda-stream
```

Figure 12: AWS CLI to Set ParallelizationFactor

As a result, the correlation between Kinesis shards and Lambda concurrency varies between 1-to-1 and 1-to-10, depending on the configuration used, as shown in Figure 13.

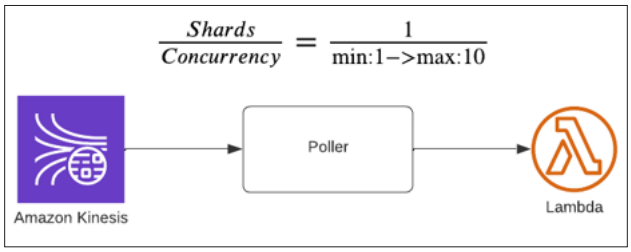

Figure 13: Kinesis Shards and Lambda Concurrency Varies between 1-to-1 and 1-to-10

When *ParallelizationFactor* equals 1 for a Kinesis function, the function's concurrency gradually increases. In contrast, concurrency can grow faster when a ParallelizationFactor equals 10. The essential component is the accuracy provided by this method, which allows for precise control over Lambda concurrency by adjusting the number of shards in the Kinesis data stream.

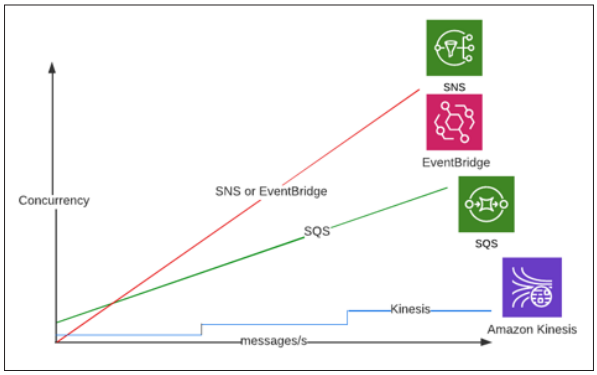

Figure 14: Concurrency Vs. Messages/s Comparison: SNS, EventBridge, SQS, Kinesis

Conclusion

However, concurrency management is only one factor when deciding which messaging services to use in AWS whenever we need to protect the downstream with overwhelming requests from Lambda functions, or we need to control the concurrent invocation of Lambda. Other aspects include cost implications, message replay feature support, error handling and retry system evaluation, and message processing order adherence. A complete table describing essential considerations to help select the best messaging service for serverless applications is as follows:

	Kenisis	SQS	SNS	EventBridge
Subscribers	1:N	1:1	1:N	1:N
Ordering	By shard	None (standard) By group (FIFO)	None	None
Replay event	1-365 days	Batched (upto 10)	Singular	Singular
Retry	Retry until success + DLQ	Retry + DLQ	Retry + DLQ	Retry + DLQ
Concurrency	1-10/shard	auto-scale	fan-out	fan-out

References

1. AWS (n.d) (2024) What is Amazon SNS? Amazon Simple Notification Service. https://docs.aws.amazon.com/sns/latest/dg/welcome.html.
2. Julian Wood (2023) Introducing maximum concurrency of AWS Lambda functions when using Amazon SQS as an event source. AWS Compute Blog. AWS https://aws.amazon.com/blogs/compute/introducing-maximum-concurrency-of-aws-lambda-functions-when-using-amazon-sqs-as-an-event-source/.
3. Tushar Sharma, Shaun Wang (2023) Understanding Amazon SQS and AWS Lambda Event Source Mapping for Efficient Message Processing. AWS Partner Network (APN) Blog. AWS https://aws.amazon.com/blogs/apn/understanding-amazon-sqs-and-aws-lambda-event-source-mapping-for-efficient-message-processing/.
4. Moheeb Zara (2019) New AWS Lambda scaling controls for Kinesis and DynamoDB event sources. AWS Compute Blog. AWS https://aws.amazon.com/blogs/compute/new-aws-lambda-scaling-controls-for-kinesis-and-dynamodb-event-sources/.