AWS Adaptive concurrency manipulator.

Project By – Shubham Sunil Chavan

Completed on – January 1, 2023

Email – chavans@uwindsor.ca

linkedIn – https://www.linkedin.com/in/shubhamchavan7

Github – https://github.com/shubham9919

Contents

Introduction 3

Problem Description 3

Project Methodology 4

Block Diagram 4

Flow Diagram of events 5

SNS Configuration 5

Concurrency Update Lambda 7

Flowchart of concurrency_update_lambda 10

DynamoDB configurations 11

SQS Queue Configurations 12

Conclusion 13

Table of Figures

Figure 1: Problem Description 3

Figure 2:Architectural Design of Solution 4

Figure 3: Flowchart of Events 5

Figure 4: SNS Configurations 6

Figure 5: Algorithm of Concurrency_update_lambda 10

Figure 6: DynamoDB Configurations 11

Figure 7:SQS Queue Configurations 13

Tables

Table 1: SNS Configuration 7

Table 2: DynamoDB Configurations. 11

Table 3: SQS Queue Configurations 13

Introduction

I started working on this project to resolve issues I faced and observed during the AWS outages.

The provided design addresses the basic issue of lambda throttling due to uneven concurrency distribution over all the lambdas in a single AWS account.

By default, AWS account comes with 1000 account concurrency which is shared between all the lambda computes. If one of the lambda experiences very high number of concurrent requests, other lambda requests start throttling. Some lambdas are very critical to failures. Such critical lambdas may face failures due to high load faced by some other less critical lambdas.

This issue can be resolved by detecting the throttling and reserving the concurrency for such important lambdas.

The discussed solution detects the lambda throttling and update the lambada concurrency adaptively by sensing AWS CloudWatch metrics data. The CloudWatch metrics data gets updated by the lambda with approximately 1 minute of delay. This data is being used to check the load of requests and updating the function concurrency accordingly.

Problem Description

Consider an EC2 application, consuming the message queue which is external to AWS cloud. The EC2 will consume the queue and call different lambdas using API gateway.

During the AWS outage, the EC2 compute stops consuming the data from the external queue. This increases the number of messages available in queue since producer is out of AWS environment and actively pushing data into the queue.

Once the AWS issue is rectified, the EC2 will start processing the data, due to increased number of messages, the EC2 will call API gateway exposed lambdas with increased concurrency.

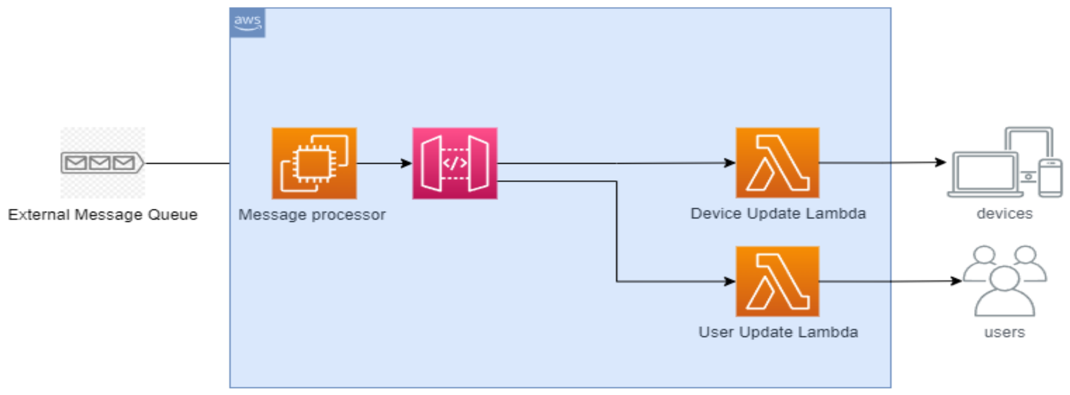


Figure 1: Problem Description

As shown in the above example design, the message processor is calling Device update lambda and user update lambda. The user update lambda is critical since its directly interacting with user but due to stated issue and 1000 concurrency limit, Device update lambda may use the excessive concurrency which results into throttling of user update lambda.

Project Methodology

Block Diagram

For the load generation, Apache JMeter is being used. JMeter calls the API Gateway with different thread count to trigger the CloudWatch alarm and test the entire flow of Adaptive concurrency manipulator.

Following is the Architectural design of Adaptive concurrency manipulator

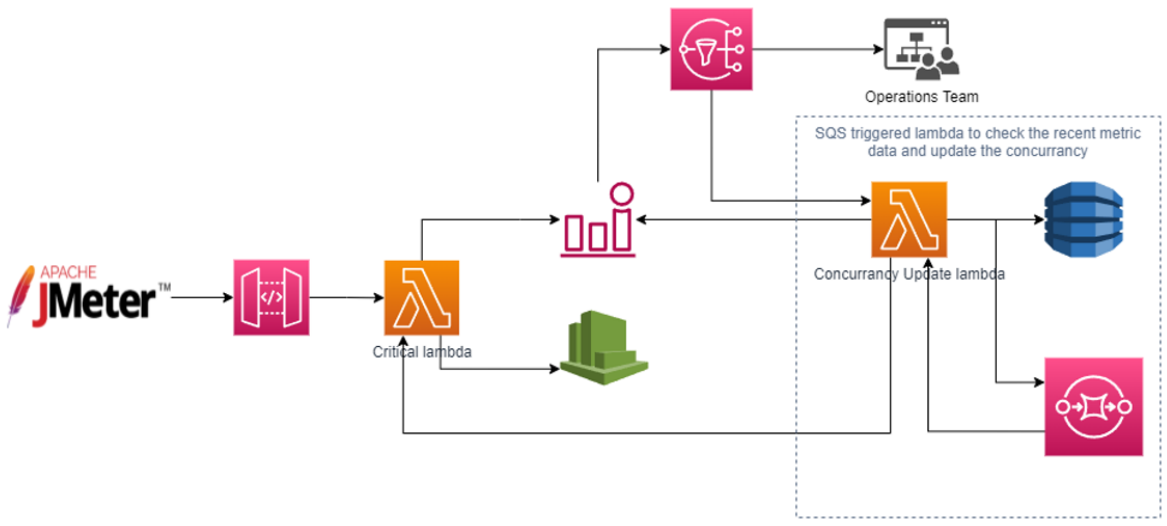

Operations Team
SQS triggered lambda to check the recent metric data and update the concurrancy
APACHE JMeter™
Concurrancy Update lambda
Critical lambda

Figure 2:Architectural Design of Solution

In the above diagram, the JMeter is calling API Gateway to invoke the critical_lambda. This lambda is pushing logs and metrics to AWS CloudWatch. The AWS CloudWatch Alarm is configured over throttles of critical lambda. This alarm is responsible for the SNS trigger. SNS is responsible for:

1. Send Email notification to operations team,
2. Trigger the concurrency update lambda.

The concurrency_update_lambda is responsible for reserving the concurrency of critical lambda and update the same in DynamoDB to keep track on events. The concurrency update lambda is also responsible for pushing updates in SQS which is also responsible for invoking the same concurrency update lambda recursively. The SQS, DynamoDB and concurrency update lambda are components of polling action, where, SQS triggers the lambda with interval of 1 minute to check the current metric data of critical lambda. The polling activity brings the adaptive nature to the design.

Flow Diagram of events

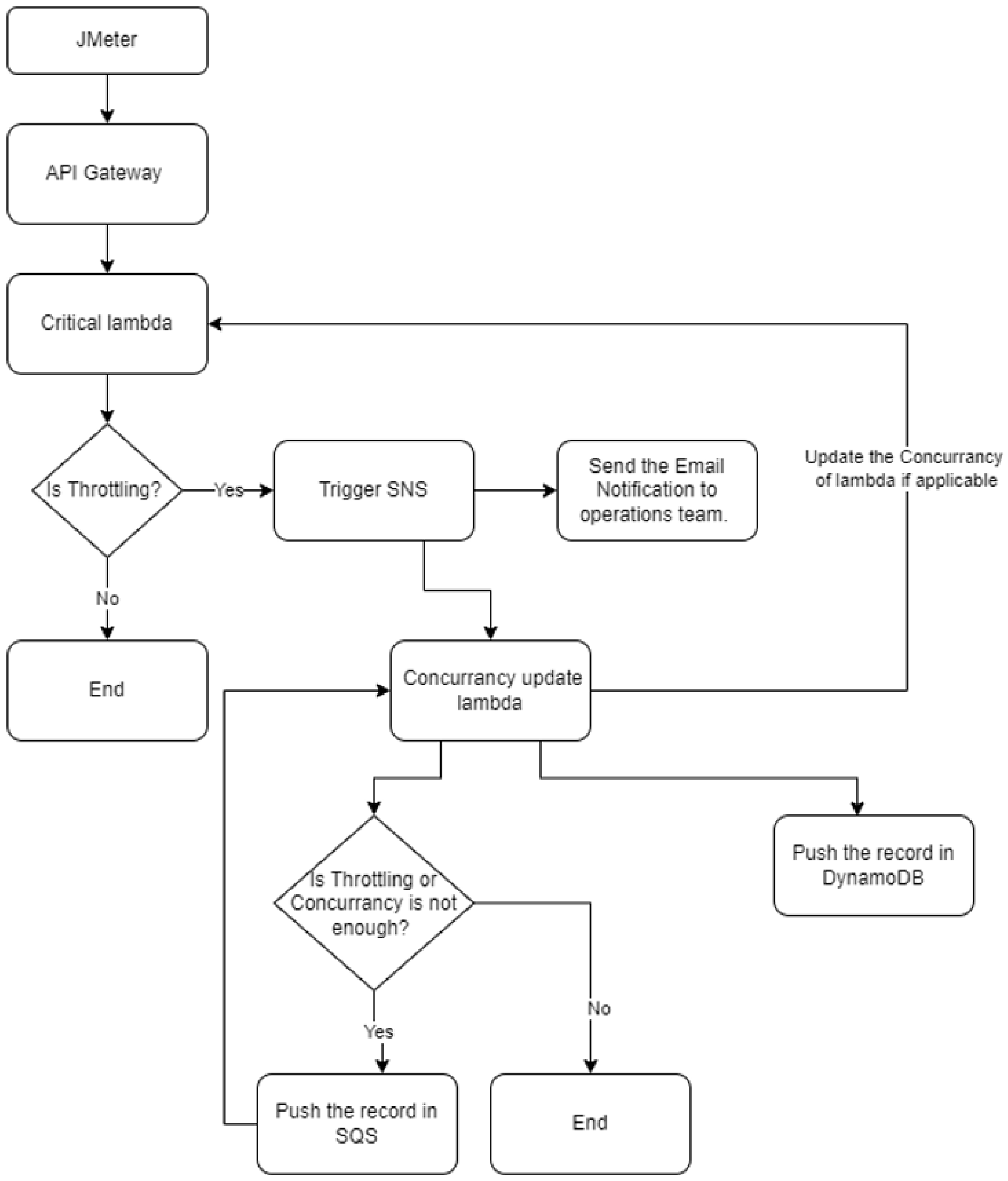
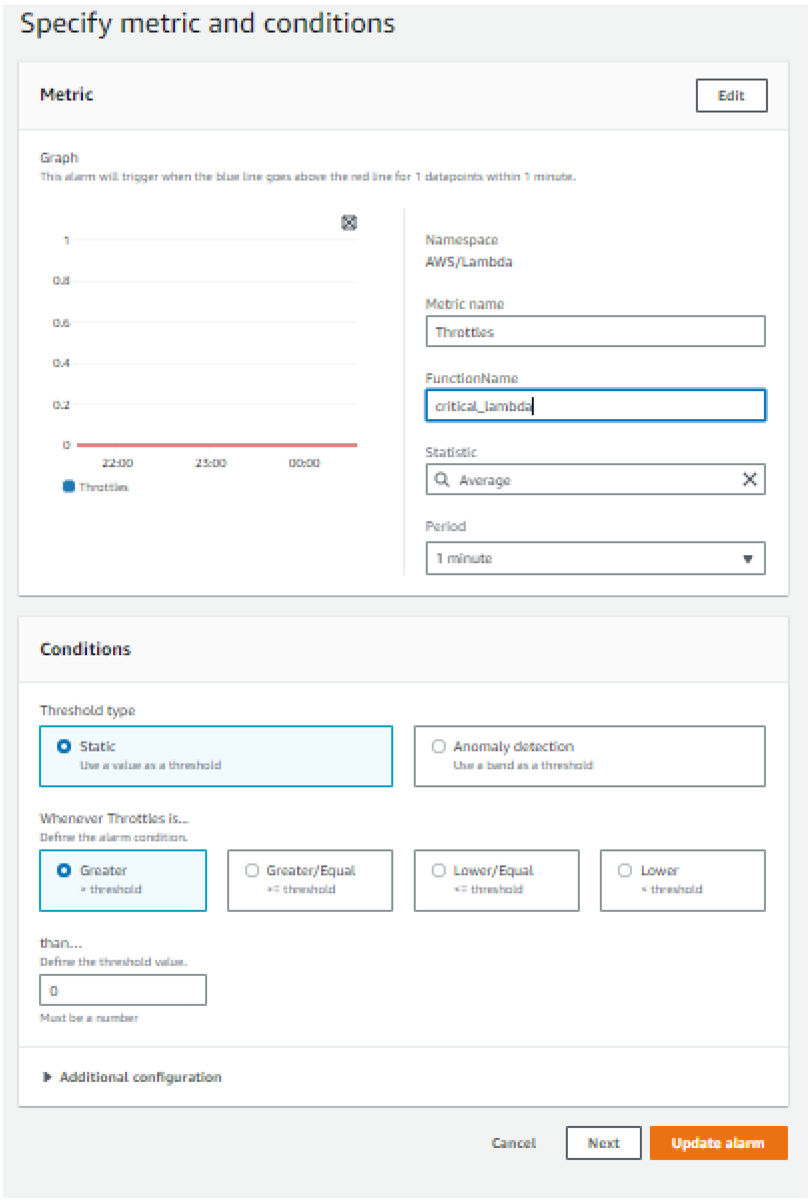

JMeter
API Gateway
Critical lambda
Is Throttling?
Yes
No
End
Trigger SNS
Send the Email Notification to operations team.
Update the Concurrancy of lambda if applicable
Concurrancy update lambda
Is Throttling or Concurrancy is not enough?
Yes
No
Push the record in SQS
End
Push the record in DynamoDB

Figure 3: Flowchart of Events

SNS Configuration

A2A and A2P are the two ways in which Amazon Simple Notification Service (SNS) transmits notifications. A2A i.e., application to application offers high-throughput, push-based, many-to-many communications between distributed systems, microservices, and event-driven serverless applications. These programmes include Amazon Simple Queue Service (SQS), Amazon Kinesis Data Firehose, AWS Lambda, and other HTTPS destinations. You can communicate with your clients using A2P i.e., Application to person capability by sending them SMS texts, push alerts, and emails.

Following are the details of SNS configuration:



Specify metric and conditions

Metric

Graph
This alarm will trigger when the blue line goes above the red line for 1 datapoints within 1 minute.

Namespace
AWS/Lambda

Metric name: Throttles

FunctionName: critical_lambda

Statistic: Average

Period: 1 minute

Conditions

Threshold type
[✓ Static — Use a value as a threshold] Anomaly detection — Use a band as a threshold

Whenever Throttles is...
Define the alarm condition.
[✓ Greater — > threshold] Greater/Equal — >= threshold; Lower/Equal — <= threshold; Lower — < threshold

than...
Define the threshold value.
0
Must be a number

Figure 4: SNS Configurations

As shown in the Above image,

Sr. No.	Parameter	Specification
1	Metric Name	Throttles
2	Function Name	Critical_lambda
3	Statistics	Average
4	Period	1 Minute

5	Threshold Type	Static
6	Alarm Condition	Greater > Threshold
7	Threshold Value	0

Table 1: SNS Configuration

Following are events after SNS trigger,

1. A2A: Trigger the concurrency_update_lambda,
2. A2P: Send the email to operations team

Concurrency Update Lambda

The concurrency update lambda is the main processor of the entire workflow. It is invoked by following 2 ways,

1. SNS topic triggered by the CloudWatch alert,
2. SQS queue.

It is responsible for,

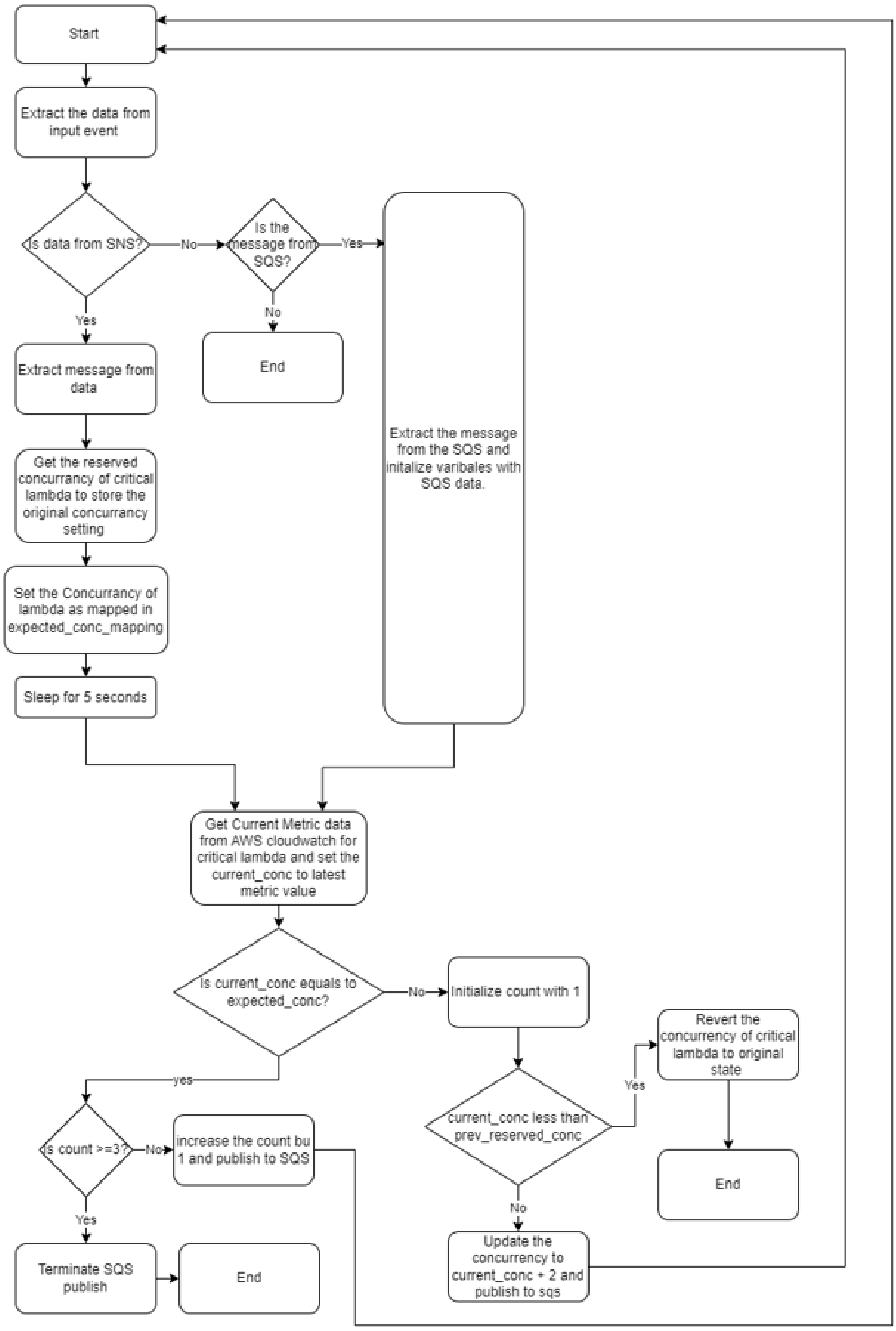
1. Updating the Concurrency of critical lambda,
2. Pushing the record in DynamoDB for record keeping,
3. Pushing the event in SQS queue for adaptively changing the concurrency of critical lambda,
4. Fetching metric records from the AWS CloudWatch to decide if the concurrency must be updated.

Following is the algorithm for Concurrency Update Lambda

```
Start
Import libraries
import lambda, cloudwatch, sqs and dynamoDB client
# initial value of conc to set on critical lambda
expected_conc_mapping = {
    'critical_lambda' : 50
}
data = Extact the Message data from the input event
count = 1
if data is from SNS:
    message = extract message from data
    prev_reserved_conc = get the original cocurrancy of lambda
    Set the Concurrency of lambda as mapped in expected_conc_mapping
    #allow some time for changes to get reflected
    Sleep(5)

else if Message is from SQS:
    lambda_name = message['lambda_name']
    expected_conc = message['expected_conc']
    prev_reserved_conc = message['prev_reserved_conc'] or prev_reserved_conc
    # override count
    count = message['count'] if 'count' in message.keys() else count
else
    print as unknown data source event and exit
current_metric = Get the current metrics data with start date as yesterday and end date as
today
current_conc = round(latest concurrency value from the current_metric data)
if current_conc != expected_conc
    #override count.
    count = 1
    if current_conc < prev_reserved_conc
            # This case corroborates that the load over critical lambda is less as compared
            # to the allotted concurrency
            revert the concurrency of critical lambda to original concurrency value
            no sqs publish
    else:
            # adding two to ensure adequate upper limit to avoid frequent changes
            Update the concurrency to current_conc + 2
            publish to sqs with delay of 1 minute and payload as {
                    'lambda_name': lambda_name,
                    'prev_reserved_conc': prev_reserved_conc,
                    'expected_conc': current_con,
                    'count': count
            }
```

```
else:
    #stop the sqs publish if the concurrency is stable for more than 2 minutes
    if count >= 3:
            terminate the SQS publish
    else
            # Increase the count before publishing it to SQS
            count = count + 1
            publish to SQS with delay of 1 minute and payload as {
                    'lambda_name': lambda_name,
                    'prev_reserved_conc': prev_reserved_conc,
                    'expected_conc': current_con,
                    'count': count,
    }

ddb_item = Search in the dynamoDB with primary key as lambda_name and value of primary key as lambda_name
if ddb_tem is present
    update the updated_reserved_conc with recent change in value of reserved concurrency(expected_conc+2)
else
    Add the new entry of lambda in DynamoDB
Print the current metric data for tracking
Return the current_metric_data

End
```

Flowchart of concurrency_update_lambda



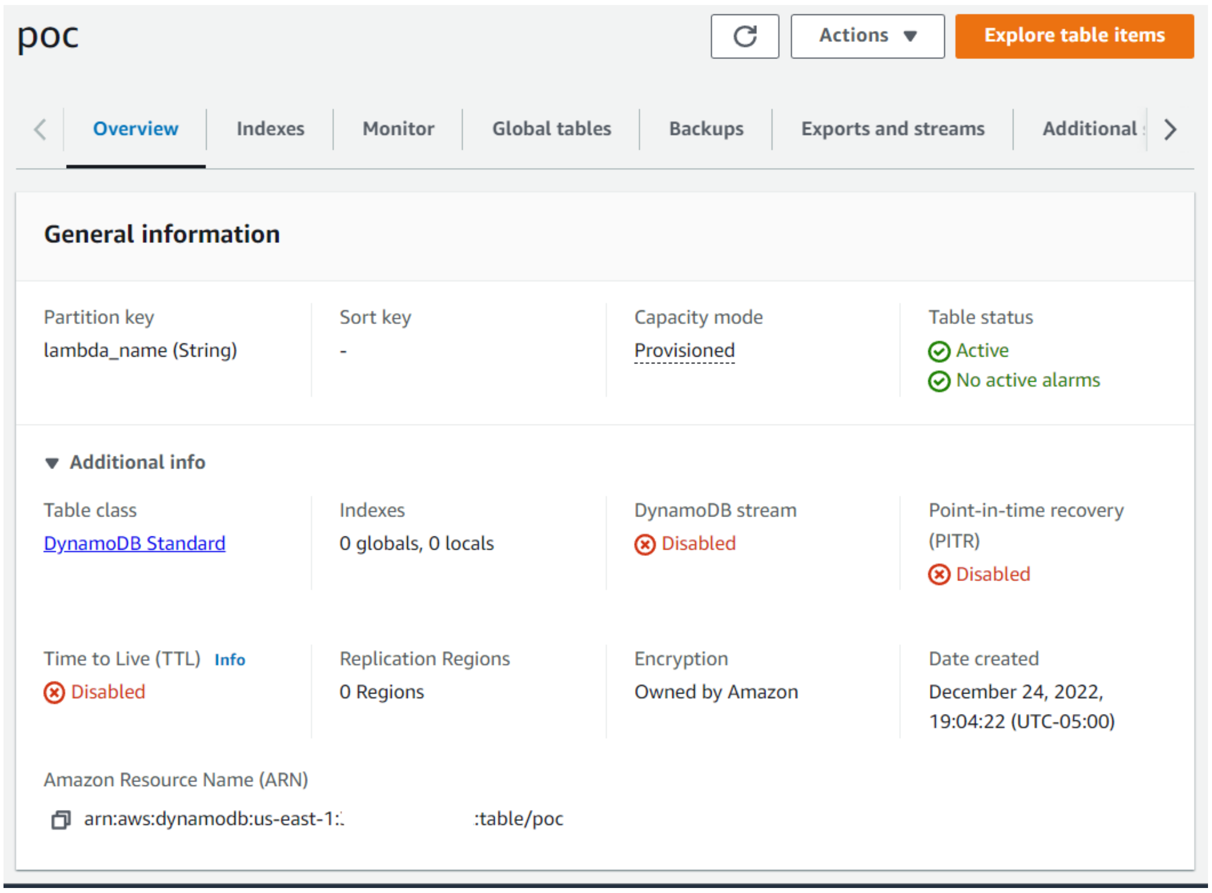
Start
Extract the data from input event
Is data from SNS?
No
Is the message from SQS?
Yes
No
End
Yes
Extract message from data
Extract the message from the SQS and initalize varibales with SQS data.
Get the reserved concurrancy of critical lambda to store the original concurrancy setting
Set the Concurrancy of lambda as mapped in expected_conc_mapping
Sleep for 5 seconds
Get Current Metric data from AWS cloudwatch for critical lambda and set the current_conc to latest metric value
Is current_conc equals to expected_conc?
No
Initialize count with 1
Revert the concurrency of critical lambda to original state
Yes
current_conc less than prev_reserved_conc
End
yes
Is count >=3?
No
increase the count bu 1 and publish to SQS
Yes
No
Terminate SQS publish
End
Update the concurrency to current_conc + 2 and publish to sqs

Figure 5: Algorithm of Concurrency_update_lambda

DynamoDB configurations

A key-value and document database from Amazon, DynamoDB offers performance at any size of one millisecond or less. It is a fully managed, multiregion, multimaster, persistent database for internet-scale applications that has built-in security, backup and restore, and in-memory caching. Over 10 quadrillion queries may be processed daily using DynamoDB, with peak throughput rates of over 20 million requests per second.

Following are the details of DynamoDB configuration:



poc

General information

Partition key	Sort key	Capacity mode	Table status
lambda_name (String)	-	Provisioned	✓ Active ✓ No active alarms

▼ Additional info

Table class	Indexes	DynamoDB stream	Point-in-time recovery (PITR)
DynamoDB Standard	0 globals, 0 locals	⊗ Disabled	⊗ Disabled
Time to Live (TTL) Info	**Replication Regions**	**Encryption**	**Date created**
⊗ Disabled	0 Regions	Owned by Amazon	December 24, 2022, 19:04:22 (UTC-05:00)

Amazon Resource Name (ARN)

arn:aws:dynamodb:us-east-1:. :table/poc

Figure 6: DynamoDB Configurations

As shown in the above image,

Sr. No	Parameter	Value
1	Table Name	poc
2	Partition Key	lambda_name
3	Sort Key	NA
4	Capacity Mode	Provisioned

Table 2: DynamoDB Configurations.

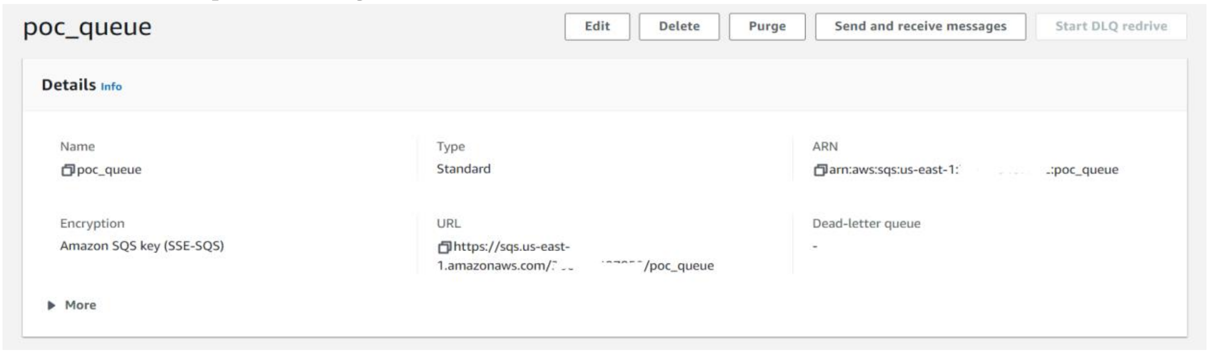
The DynamoDB is being used to store the lambda name along with original capacity and the list of updates in the concurrency. Following is the format of item stored in the DynamoDB in DynamoDB JSON format.

```
{
    "lambda_name": {
            "S": "critical_lambda"
    },
    "prev_conc": {
           "N": "5"
    },
    "updated_reserved_conc": {
            "L": [
                {
                 "N": "10"
                },
                {
                 "N": "15"
                },
                {
                 "N": "30"
                }
        ]
    }
}
```

SQS Queue Configurations

One can send, store, and receive messages across software components using Amazon Simple Queue Service (SQS) at any volume without worrying about message loss or needing other services to be available.

Details of SQS queue configuration are as follows,


poc_queue

Details Info

Name: poc_queue

Type: Standard

ARN: arn:aws:sqs:us-east-1: :poc_queue

Encryption: Amazon SQS key (SSE-SQS)

URL: https://sqs.us-east-1.amazonaws.com/ /poc_queue

Dead-letter queue: -

Figure 7:SQS Queue Configurations

As shown in the above image

Sr. No	Parameter	Value
1	Name	poc_queue
2	Type	Standard
3	Message retention period	4 days
4	Dead-letter queue	NA

Table 3: SQS Queue Configurations

The SQS queue is to bring the adaptiveness in the design. The produces and consumer of the SQS queue is same lambda i.e., concurrency_update_lambda. This lambda pushes the data in SQS until the load is stabilized or back to the normal state.

The counter is maintained and tracked though SQS payload, where, if the concurrency is stable and adequate for more than 4 minutes or 4 invocations of lambda, The SQS queue publish is terminated.

There are two cases in which SQS publish is terminated:

1. When concurrency metric is stable for more than 4 minutes,
2. When the concurrency is less than the original value of concurrency before the load.

Conclusion

As discussed above, the design can be used efficiently in cloud outage scenarios as well as when there is an unexpected load on any endpoint. The solution can be integrated with other services of AWS for overall tracking of the environment.