ISSN: 2754-6659

Journal of Artificial Intelligence & Cloud Computing

SCIENTIFIC Research and Community

Review Article

Open Access

Optimizing Observability: A Deep Dive into AWS Lambda Metrics

Balasubrahmanya Balakrishna

Senior Lead Software Engineer, Richmond, VA, USA

ABSTRACT

The importance of observability in AWS Lambda systems is examined in this technical article, which focuses on using Python and the AWS Lambda Powertools to improve monitoring capabilities. With serverless computing, AWS Lambda has emerged as a critical service that lets developers concentrate on writing code rather than maintaining infrastructure. Nonetheless, troubleshooting, performance improvement, and guaranteeing the dependability of serverless apps depend on effective observability. Using specially designed Python Serverless Restful API and AWS Lambda Powertools, this article thoroughly analyzes important AWS Lambda metrics, their interpretation, and techniques to improve observability while running serverless applications in the AWS Lambda environment.

The author will use Python with the AWS Lambda Powertools, implemented as a Lambda layer, to facilitate seamless integration for custom metrics and advanced observability features.

***Corresponding author**

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

Received: November 07, 2022; **Accepted:** November 16, 2022; **Published:** November 25, 2022

Keywords: AWS Lambda, CloudWatch, CloudWatch Metrics, AWS Powertools for Lamda

Background: AWS Cloud Watch Metrics

Two methods are available with AWS Lambda for gathering and examining metrics:

1. Default Option

Without requiring any setup, the default configuration streamlines the monitoring process for users by automatically gathering and presenting critical metrics [1].

2. Custom Option

AWS Lambda allows users to set custom metrics for more customized observability [2]. By allowing the gathering of particular data points, this option makes it possible to monitor and analyze data in a more detailed and unique manner.

Lambda Emits a Default Set of Metrics:

Invocation Metrics

Provides information about the outcome of an Invocation.

E.g., Invocations, Errors, Throttles, Dead Letter Errors, Destination Delivery Failures, Provisioned Concurrency Invocations, Provisioned Concurrency Spillover Invocations.

- Dead Letter Errors and Destination Delivery Failures are applicable for Async invocations

Performance Metrics

Provides data about the performance outcome of a single invocation.

E.g., Duration, Post Runtime Extension Duration, Iterator Age, Offset Lag.

- Duration is rounded to the nearest millisecond
- PostRuntimeExtensionDuration is applicable with Lambda extensions
- IteratorAge applies to stream-based
- Offsetlag applies to self-managed Kafka or MSK

Concurrency Metrics

Provides data about the aggregate count of instances processing events.

E.g., Concurrent Executions, Provisioned Concurrent Executions, Provisioned Concurrency Spillover Invocations, Provisioned Concurrency Utilization, Unreserved Concurrent Executions.

- Information can be at the function, version, alias, or region scope
- Unreserved Concurrent Executions represents a region and provides the number of events that function without reserved concurrency are processing
- Concurrent Executions gives the number of function instances processing requests
- Provisioned Concurrent Executions gives the number of function instances processing events on provisioned concurrency
- Provisioned Concurrency Utilization gives the value Provisioned Concurrent Executions divided by the amount of provisioned concurrency for a version or alias

Metrics come with predefined retention and resolution periods, as shown in the table below:

Resolution Period	Retention Period
<1 min*	3 hours
60 sec	15 days
300 sec	63 days
3600 sec	455 days

Initially, data points published at a shorter interval undergo aggregation for prolonged storage. As an illustration, when data is gathered at a 1-minute frequency, it remains accessible at a 1-minute resolution for 15 days. Following this initial period, the data persists but undergoes further aggregation, retrievable only at a 5-minute resolution.

Beyond the 63-day mark, the aggregated data experiences another level of consolidation, becoming accessible with a 1-hour resolution. These evolving resolutions cater to different needs over time, balancing granularity with long-term storage efficiency.

Furthermore, the metrics maintain a retention period of 15 months before entering a rolling-out phase. This structured approach to data retention ensures a balance between historical depth and resource optimization.

The table below provides the appropriate usage of statistics for each metric type:

Metric Type	Statistic to Use
Invocation Metrics	Sum
Performance Metrics	Average/Max/Percentile Stats
Concurrency Metrics	Max

Introduction

Application Load Balancer (ALB)-fronted Lambda API Architecture

In this exploration, we will spotlight the practical implementation of observability concepts through an Application Load Balancer (ALB)-fronted Lambda API architecture, as depicted in Figure 1. This architectural choice underscores the real-world applicability of the discussed observability strategies.

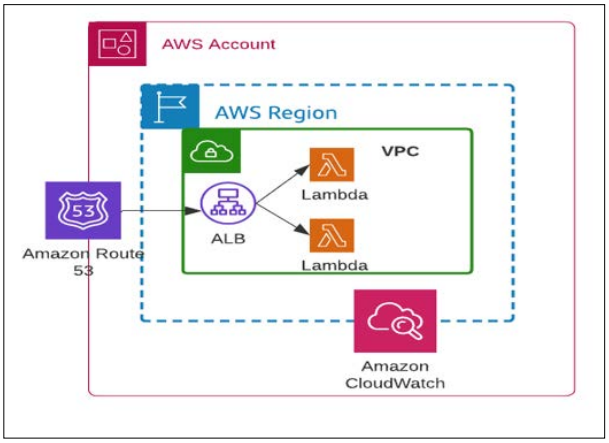

Figure 1: High-Level Architecture of ALB-Fronted Lambda API

Purpose-Built Lambda Function with AWS Powertools

The key AWS Lambda metrics, their interpretation, and strategies to elevate observability throughout this paper. By utilizing Python and AWS Lambda Powertools, the aim is to empower developers and system administrators with practical insights into building resilient, high-performing, serverless applications. The chosen architectural context and purpose-built function serve as tangible examples of the efficacy of these observability strategies in real-world scenarios.

Optimizing Metric Generation: Leveraging Cloudwatch EMF and Lambda Powertools

Leveraging CloudWatch EMF

Custom metrics can be generated inside a chosen namespace by facilitating the ingestion of logs containing application data using the CloudWatch Embedded Metrics Format (EMF)[3]. These customized metrics are crucially produced asynchronously. A single EMF object can aggregate 100 metrics within a custom namespace. It is not recommended to synchronously report a statistic to CloudWatch Metrics since it hurts function scalability and code performance. Steer clear of such methods.

```
        "CloudWatchMetrics": [
            {
                "Namespace": "BooksService",
                "Dimensions": [
                    []
                ],
                "Metrics": [
                    {
                        "Name": "BooksApiSuccess",
                        "Unit": "Count"
                    }
                ]
            }
        ]
    },
    "BooksApiSuccess": [
        1
    ]
```

Figure 2: EMF Output in CloudWatch Logs

It is highly advised for developers working with Python, Typescript, or Java to utilize AWS Lambda Powertools [4]. This toolbox offers a lot of functionality, including analytics, tracing, logging, and other features. Most notably, it makes it possible to generate logs in the proper EMF format quickly.

Exploring Metric Math Capabilities

Although Metric Math is a feature often disregarded, it is more comprehensive than AWS Lambda and Amazon CloudWatch [5]. Once a measure is recorded in CloudWatch, you can use mathematical expressions to create new time series based on the existing metrics, whether custom or built-in. One well-known example in AWS Lambdas is calculating the mistake rate by dividing the Errors metric by the Invocation metric. Metric analysis gains further adaptability by allowing alarms to be created using metrics-related math calculations.

Lambda Function with AWS Powertools Producing Default Metrics on Invocation

The default metrics (Figure 4) are automatically generated as part of the default behavior when the Application Load Balancer (ALB) executes the sample API using AWS Powertools for Python, shown in Figure 3. As the code shows, custom metrics have yet to be expressly defined.

```python
import json
import random
from aws_lambda_powertools import Logger, Tracer
from aws_lambda_powertools.event_handler import ALBResolver, Response, content_types
from aws_lambda_powertools.event_handler.exceptions import NotFoundError
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths

logger = Logger()
tracer = Tracer()

app = ALBResolver()

SUCCESS_HTTP_STATUS_CODES = [200, 202, 204]
FAILURE_HTTP_STATUS_CODES = [401, 400]

class Status:
    SUCCESS = 'success'
    ERROR = 'error'
    UNKNOWN = 'unknown'

@app.get("/books")
@tracer.capture_method
def get_books() -> Response:
    status = random.choice([Status.SUCCESS, Status.ERROR, Status.UNKNOWN])

    if status == Status.SUCCESS:
        http_status_code = random.choice(SUCCESS_HTTP_STATUS_CODES)
        result = {
            'http_status_code': http_status_code,
            'resource': app.current_event.path,
            'books': ['book1', 'book2', 'book3']
        }
        logger.info("Successfully retrieved books", extra=result)
        return Response(
            status_code=http_status_code,
            content_type=content_types.APPLICATION_JSON,
            body=json.dumps(result))

    elif status == Status.ERROR:
        http_status_code = random.choice(FAILURE_HTTP_STATUS_CODES)
        result = {
            'http_status_code': http_status_code,
            'resource': app.current_event.path,
            'books': []
        }
        logger.error("Error retrieving books", extra=result)
        return Response(
            status_code=http_status_code,
            content_type=content_types.APPLICATION_JSON,
            body=json.dumps(result))

    else:
        try:
            raise Exception('Unknown Exception')
        except Exception as e:
            logger.exception("Exception occurred", extra={'exception_message': str(e)})

@app.not_found
@tracer.capture_method
def handle_not_found_errors(exc: NotFoundError) -> Response:
    logger.info(f"Not found route: {app.current_event.path}")
    return Response(status_code=418, content_type=content_types.TEXT_PLAIN, body="I'm a teapot!")


@logger.inject_lambda_context(correlation_id_path=correlation_paths.APPLICATION_LOAD_BALANCER,
log_event=True, clear_state=True)
@tracer.capture_lambda_handler
def lambda_handler(event: dict, context: LambdaContext) -> Response:
    return app.resolve(event, context)
```

Figure 3: Purpose-Built Lambda Function with AWS Powertools

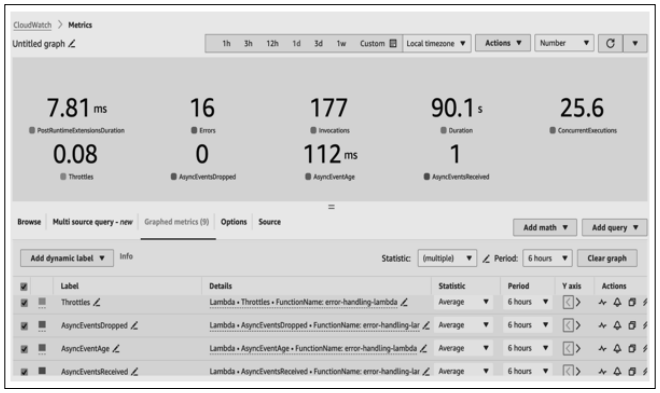

Figure 4: CloudWatch Metrics Generated By Default

Lambda Function with AWS Powertools Produces Custom Metrics on Invocation

Augment the code mentioned above by introducing custom metrics to gain deeper insights into business-level metrics. The modified code snippet below (Figure 5) demonstrates the incorporation of custom metrics. It's important to note that this example merely scratches the surface of the extensive capabilities that Powertools offers for accomplishing a wide range of tasks.

```python
import json
import random
from aws_lambda_powertools import Logger, Tracer, Metrics
from aws_lambda_powertools.metrics import MetricUnit
from aws_lambda_powertools.event_handler import ALBResolver, Response, content_types
from aws_lambda_powertools.event_handler.exceptions import NotFoundError
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths

logger = Logger(service="BooksService")
tracer = Tracer(service="BooksService")
metrics = Metrics(namespace="BooksService")

app = ALBResolver()

SUCCESS_HTTP_STATUS_CODES = [200, 202, 204]
FAILURE_HTTP_STATUS_CODES = [401, 400]

class Status:
    SUCCESS = 'success'
    ERROR = 'error'
    UNKNOWN = 'unknown'

@app.get("/books")
@tracer.capture_method
def get_books() -> Response:
    status = random.choice([Status.SUCCESS, Status.ERROR, Status.UNKNOWN])

    if status == Status.SUCCESS:
        http_status_code = random.choice(SUCCESS_HTTP_STATUS_CODES)
        result = {
            'http_status_code': http_status_code,
            'resource': app.current_event.path,
            'books': ['book1', 'book2', 'book3']
        }
        logger.info("Successfully retrieved books", extra=result)
        metrics.add_metric(name="BooksApiSuccess", unit=MetricUnit.Count, value=1)
        return Response(
            status_code=http_status_code,
            content_type=content_types.APPLICATION_JSON,
            body=json.dumps(result))

    elif status == Status.ERROR:
        http_status_code = random.choice(FAILURE_HTTP_STATUS_CODES)
        result = {
            'http_status_code': http_status_code,
            'resource': app.current_event.path,
            'books': []
        }
        logger.error("Error retrieving books", extra=result)
        metrics.add_metric(name="BooksApiFailure", unit=MetricUnit.Count, value=1)
        return Response(
            status_code=http_status_code,
            content_type=content_types.APPLICATION_JSON,
            body=json.dumps(result))

    else:
        try:
            raise Exception('Unknown Exception')
        except Exception as e:
            metrics.add_metric(name="BooksApiException", unit=MetricUnit.Count, value=1)
            logger.exception("Exception occurred", extra={'exception_message': str(e)})

@app.not_found
@tracer.capture_method
def handle_not_found_errors(exc: NotFoundError) -> Response:
    logger.info(f"Not found route: {app.current_event.path}")
    return Response(status_code=418, content_type=content_types.TEXT_PLAIN, body="I'm a teapot!")


@logger.inject_lambda_context(correlation_id_path=correlation_paths.APPLICATION_LOAD_BALANCER,
log_event=True, clear_state=True)
@tracer.capture_lambda_handler
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> Response:
    return app.resolve(event, context)
```

Figure 5: Modified API with Custom Metrics

Within the code, we have established a custom namespace labeled BooksService and populated it with business-level metrics. Figure 6 and Figure 7 below illustrate the specific metric within this custom namespace.

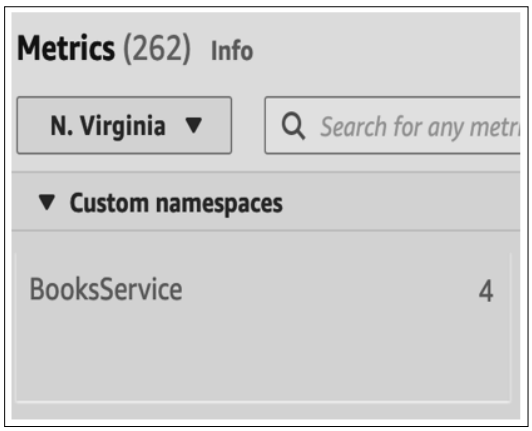

Figure 6: Custom Namespace

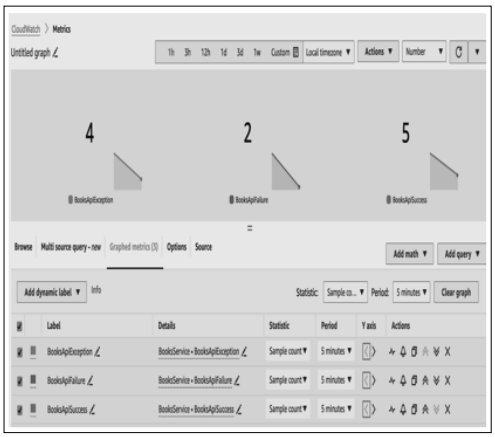

Figure 7: Custom Metrics

Conclusion

AWS Lambda is a critical player in the serverless computing space, freeing developers from the burden of managing infrastructure in favor of writing code. This exploration of observability optimization in AWS Lambda environments has revealed a variety of tactics meant to improve accuracy and efficiency. Exploring default metrics categories like Performance, Concurrency, and Invocation has helped us establish a solid foundation for debugging and monitoring.

Furthermore, a potentially has appeared with the advent of AWS Lambda Powertools, which are smoothly linked with Python. One helpful step toward a more sophisticated observability framework is using the CloudWatch Embedded Metrics Format (EMF) and the advice to use Powertools for metric creation, tracing, and logging.

Beyond the technical details, investigating Metric Math capabilities and adding custom metrics under a namespace designated explicitly for them (such as "BooksService") highlight the breadth of analysis possible with AWS Lambda and CloudWatch.

As we go through the observability's complexities, it is clear that this journey is about more than just measurements and codes-it's about making decisions based on intelligent facts. Refinements for optimal system knowledge and responsiveness are prioritized, whether avoiding anti-patterns like synchronous metric publication or utilizing Metric Math for meaningful computations.

Essentially, the cooperative synergy of AWS Lambda, CloudWatch, Python, and Powertools provides a comprehensive approach to observability, which guarantees the identification of problems and proactive optimization for continued performance. Developers and system administrators can create a serverless environment that functions effectively and changes dynamically in response to the needs of the applications it supports by following best practices, avoiding hazards, and utilizing the capabilities offered. Pursuing increased observability is a process rather than a destination, and this investigation is the first step toward a day when AWS Lambda applications function at their best.

References

1. Metrics collected by Lambda Insights. Amazon CloudWatch User Guide https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/Lambda-Insights-metrics.html.
2. Publish custom metrics. Amazon CloudWatch User Guide https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/publishingMetrics.html.
3. Setting alarms on metrics created with the embedded metric format. Amazon CloudWatch User Guide https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch_Embedded_Metric_Format_Alarms.html
4. Metrics. AWS Powertools for AWS Lambda (Python) https://docs.powertools.aws.dev/lambda/python/latest/core/metrics/.
5. Use metric math. Amazon CloudWatch User Guide https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/using-metric-math.html.