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 Logging

Balasubrahmanya Balakrishna

Senior Lead Software Engineer, Richmond, VA, USA

ABSTRACT

Ensuring the reliability and performance of AWS Lambda functions is contingent upon effective observability as serverless computing becomes an essential component of contemporary application development. This paper explores the crucial function of logging in AWS Lambda and clarifies its importance to observability. The importance of serverless systems and the difficulties they provide for conventional monitoring techniques are discussed in the introduction. The paper clarifies the core ideas of observability and places them in the context of the event-driven paradigm of AWS Lambda.

Highlighting the interaction with CloudWatch Logs, the paper delves into the native logging features of AWS Lambda. It offers best practices for optimizing logging strategy, such as using structured logging and taking asynchronous and distributed systems into account.

The study highlights the concrete influence of observability on system reliability by presenting case studies from real-world scenarios in which efficient logging in AWS Lambda was crucial in locating and fixing problems. To address the changing requirements of serverless application development, the conclusion highlights the current issues and potential developments in AWS Lambda logging and calls for more investigation and study.

***Corresponding author**

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

Received: March 04, 2023; **Accepted:** March 14, 2023; **Published:** March 22, 2023

Keywords: AWS Lambda, Cloud Watch, Cloud Watch Logs Insights, AWS Power tools for Lamda

Introduction

AWS Lambda's sophisticated architecture and event-driven execution methodology, inherent to the serverless design, present a novel approach to developing applications. Even though Lambda has a built-in logging interface with AWS Cloud Watch, its proper use necessitates a sophisticated grasp of its advantages and shortcomings. With a particular emphasis on logging, this article explores the landscape of AWS Lambda observability, removing any confusion and providing strategies for success.

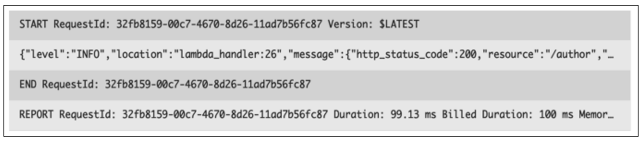
Lambda, by default, seamlessly integrates with AWS Cloud Watch for logging. However, ensuring that functions have proper permissions to emit these logs is critical. Even if a Lambda function doesn't explicitly emit logs, Lambda captures essential data about invocations, a facet that will be explored in detail throughout this series. The platform captures standard output (STDOUT) and standard error (STDERR) streams, providing a comprehensive view of function execution.

Although engineers can utilize pre-existing runtime-logging frameworks, this paper clarifies, certain limitations linked to conventional logging libraries. Crucial concerns arise around how easy it is for humans to read log lines and how best to optimize them for search engines. We address these issues by delving into the application of JMES Path for effective log stream searches and investigating extra data that shows up in log lines in the Lambda space that becomes useful.

This article presents AWS Lambda Power tools, a robust library that streamlines the intricacies of logging in recognition of the necessity for structured logging. Lambda power tools provides a high-level overview of its features and how it contributes to structured and relevant logs in the AWS Lambda environment, freeing engineers to concentrate on business requirements by abstracting away the heavy lifting. This exploration aims to provide developers with the skills and information necessary to fully utilize AWS Lambda logging for reliable observability in serverless architectures.

Logging in AWS Lambda

Logs are produced in a particular format for every function call. START, END, and REPORT log lines are standard for function invocation with RequestId that ties an invocation, as shown in Figure 1

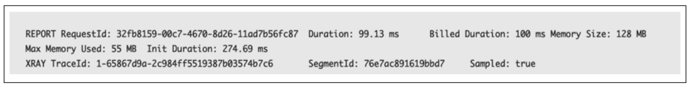


```
START RequestId: 32fb8159-00c7-4670-8d26-11ad7b56fc87 Version: $LATEST
{"level":"INFO","location":"lambda_handler:26","message":{"http_status_code":200,"resource":"/author","…
END RequestId: 32fb8159-00c7-4670-8d26-11ad7b56fc87
REPORT RequestId: 32fb8159-00c7-4670-8d26-11ad7b56fc87 Duration: 99.13 ms Billed Duration: 100 ms Memor…
```

Figure 1: AWS Log lines - START, END and REPORT

These logs' REPORT line, sample shown in Figure 2, which offers thorough information on the function call, is crucial:

- Duration provides information about the function's runtime and processing time.
- Memory size indicates how much a function has been allotted, affecting Lambda costs.
- Max Memory Used details the exact amount of memory the function uses while running.
- Init Duration is unique to cold starts and represents the time needed for initialization.



```
REPORT RequestId: 32fb8159-00c7-4670-8d26-11ad7b56fc87  Duration: 99.13 ms      Billed Duration: 100 ms Memory Size: 128 MB
Max Memory Used: 55 MB  Init Duration: 274.69 ms
XRAY TraceId: 1-65867d9a-2c984ff5519387b03574b7c6       SegmentId: 76e7ac891619bbd7     Sampled: true
```

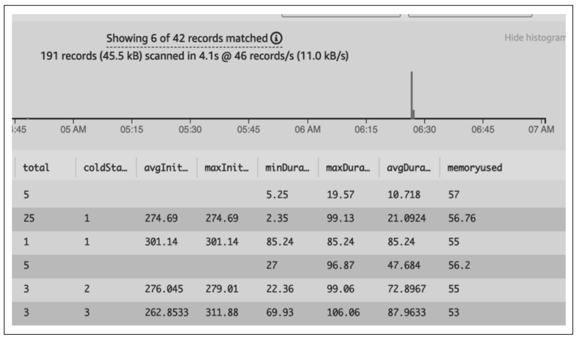
Figure 2: AWS Log REPORT line

Figure 2 shows the function execution lasted 99.13 ms, utilized 55 MB of memory, and had a configured memory setting of 128MB. The presence of Init Duration indicates that this invocation was a cold start.

The scant information from this one invocation emphasizes the necessity of thoroughly analyzing all invocations to identify any patterns. This study aims to identify patterns without the laborious process of going through Lambda logs by hand. Alternatively, use CloudWatch Log Insights methodically sift through all logs and spot patterns [1]. The given query calculates metrics for well-informed decision-making on the function, focusing on cold starts. To simplify statistical calculations, the query, shown in Figure 3, splits the data into 3-minute periods and selectively analyzes the REPORT line, screening for the presence of Init Duration, which is suggestive of cold beginnings.

```
filter @type = "REPORT"
  | parse @message /Init Duration: (?<init>\S+)/
  | stats count() as total,
   count(init) as coldStarts,
   avg(init) as avgInitDuration,
   max(init) as maxInitDuration,
   min(@duration) as minDuration,
   max(@duration) as maxDuration,
   avg(@duration) as avgDuration,
   avg(@maxMemoryUsed)/1000/1000 as memoryused
  by bin (3min)
```

Figure 3: Cloud Watch Log Insights Query for Cold Start



Showing 6 of 42 records matched
191 records (45.5 kB) scanned in 4.1s @ 46 records/s (11.0 kB/s)

total	coldSta...	avgInit...	maxInit...	minDura...	maxDura...	avgDura...	memoryused
5				5.25	19.57	10.718	57
25	1	274.69	274.69	2.35	99.13	21.0924	56.76
1	1	301.14	301.14	85.24	85.24	85.24	55
5				27	96.87	47.684	56.2
3	2	276.045	279.01	22.36	99.06	72.8967	55
3	3	262.8533	311.88	69.93	106.06	87.9633	53

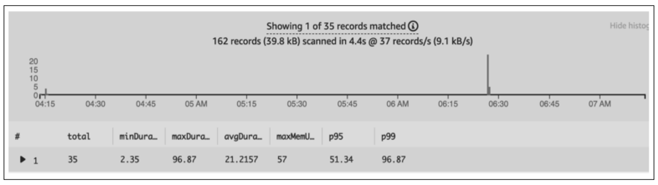
Figure 4: CloudWatch Log Insights Result of Cold Start

Let's enhance the previous query, illustrated in Figure 5, to identify all warm starts within the same log stream and assess the performance of the functions. The addition of p95 and p99 aims to provide a nuanced perspective on performance metrics.

```
filter @type = "REPORT"
  | filter @message not like /Init Duration:/
  | stats count() as total,
  | min(@duration) as minDuration,
  | max(@duration) as maxDuration,
  | avg(@duration) as avgDuration,
  | max(@maxMemoryUsed)/1000/1000 as
maxMemUsed,
  | pct(@duration, 95) as p95,
  | pct(@duration, 99) as p99
```

Figure 5: Cloud Watch Log Insights Query to Measure Performance

Conclusively, the aggregated data from all invocations indicates a mean execution time of 21.2157 ms, with a peak memory usage not exceeding 57 MB. The p95 metric reflects a 51.34 ms duration, while the p99 metric indicates a maximum duration of 96.87 ms.



Showing 1 of 35 records matched
162 records (39.8 kB) scanned in 4.4s @ 37 records/s (9.1 kB/s)

#	total	minDura...	maxDura...	avgDura...	maxMemU...	p95	p99
▶ 1	35	2.35	96.87	21.2157	57	51.34	96.87

Rationale

The above illustration explores the capability of JMES Path within Cloud Watch Log Insights, emphasizing its role in enhancing the search and analysis of log streams. By leveraging JMES Path queries, users can pinpoint relevant data, filter logs based on specific criteria, and extract meaningful information, ultimately facilitating a more efficient and targeted approach to log analysis within the Cloud Watch environment.

Employing a conventional logging library in any programming language for Lambda functions necessitates extra effort to render logs queryable. Inadequate implementation could adversely affect the performance of the Lambda function. To illustrate, Python's standard logging library inherently involves acquiring and releasing a master lock on the thread to process log responses. Manual tuning becomes imperative to guarantee the output adheres to JSON syntax, allowing seamless querying via Cloud Watch Insights. This process also involves auto-discovering fields, ensuring their immediate accessibility for search functionalities [2].

The aws-lambda-power tools for Python Logger simplifies the configuration requirements for producing compliant records while augmenting the features usually associated with the stdlib logging library [3]. All you need to do is give the Logger the native dictionary you want to log and it will take care of the rest, automating the procedures of stream configuration, tracing, and serialization following AWS best practices. This method adheres to recommended principles for AWS Lambda functions and provides a more transparent and effective solution to handle logging tasks.

Following are some visible benefits of using AWS power tools for Lambda:

- Managed by AWS as an open-source solution, creating a solution from scratch is unnecessary.
- It effortlessly captures crucial Lambda fields, including context and cold start information.
- Flexibility is provided to append extra keys to logs as required, facilitating customization.
- The option to include a correlation ID enhances end-to-end

reconciliation and troubleshooting capabilities.
- The tool's automated discovery of fields in logs significantly improves the search experience, ensuring a more efficient and seamless process.
- AWS Powertools for Lambda also supports other Runtimes.

Conclusion

AWS Lambda logging is essential to serverless architectures' successful observability. Although native CloudWatch integrations provide a starting point, problems occur with manual parsing and the best possible log structure. These intricacies are simplified using AWS Lambda Powertools, guaranteeing adherence to AWS standard practices. While Powertools and other tools provide automatic log discovery and serialization, the strength of JMESPath in CloudWatch Log Insights improves search possibilities. Through comprehension of Lambda logging intricacies and the adoption of purpose-built solutions, developers may extract pragmatic insights, execute efficient troubleshooting, and enhance serverless function performance for an application environment that is more robust and expandable.

References

1. Analyzing log data with Cloud Watch Logs Insights. Amazon Cloud Watch Logs-User Guide https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/AnalyzingLogData.html.
2. Supported logs and discovered fields. Amazon Cloud Watch Logs-User Guide https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_AnalyzeLogData-discoverable-fields.html.
3. Logger. Power tools for AWS Lambda (Python) https://docs.powertools.aws.dev/lambda/python/latest/core/logger/.