Image Processing Application

Antonino Rando

October 6, 2025

Abstract

This paper details the deployment of a serverless image processing application utilizing **Amazon Web Services (AWS)** Lambda and its performance evaluation via **Apache Jmeter**. The core application functionality involves image manipulation, specifically converting a color image to a tone-scale (e.g., grayscale). The methodology covers application design, serverless deployment architecture, configuration of dependencies via Lambda Layers, establishment of a RESTful endpoint using AWS API Gateway, and a formal performance testing methodology to assess system behavior under various load conditions.

1 Introduction: The Application Design

The developed application is an image processing utility implemented in Python. Its primary function leverages the Pillow library to perform fundamental image manipulations. Currently, the supported operation is the conversion of a color image to a tone-scale (such as grayscale). This design was chosen for its simplicity and clear demonstration of serverless function capabilities for computationally moderate tasks.

2 Background

The foundation for this project stems from practical experience gained in Cloud Computing, specifically within the Amazon Web Services (AWS) ecosystem. Prior professional experience, including a role at Avvale, involved the management of a JavaScript-based backend entirely hosted and orchestrated on AWS. This environment was engineered to handle diverse and intensive workloads, which provided foundational knowledge in leveraging cloud-native solutions for robustness and scalability. Key AWS services utilized included AWS Lambda for serverless function execution, Amazon S3 for secure and scalable object storage (with programmatic interaction), and Amazon CloudWatch for comprehensive logging and performance monitoring. This hands-on experience has been instrumental in informing the design, deployment, and rigorous testing of the current serverless application.

3 AWS API Deployment

This section outlines the systematic procedure for deploying the image processing application as a set of serverless APIs on the AWS cloud platform. The strategy prioritizes the use of cloud-native services to ensure a scalable, reliable, and cost-efficient architecture within the constraints of a free-tier/credit-based AWS account.

3.1 Building the Lambda Function

The initial step involved the local development and optimization of the application code in Python. Following successful local testing, the application was packaged for the AWS environment. An AWS Lambda function, named *ImageConvertToColorScale*, was created using the Python 3.13 runtime (or the compatible version available at the time of deployment).

The application logic, including the mandatory `lambda_handler` function, was integrated into the Lambda code editor and subsequently deployed to activate the changes.

3.2 Configuring External Dependencies with Lambda Layers

Due to the application's reliance on the external Pillow library, a Lambda Layer was implemented to manage this dependency, a standard practice for maintaining compact deployment packages. The following process was adopted:

1. The Pillow library was downloaded into a dedicated local directory using `pip3 install Pillow -t .`.
2. The directory was compressed into a zip file, pillow-layer.zip.
3. This zip file was uploaded to a newly provisioned Amazon S3 bucket (e.g., uni-project-pillow-library-9e08dd8c-cdef-434a-abbb-07d3311da229), adhering to S3 bucket naming conventions.
4. The direct S3 URL of the uploaded pillow-layer.zip was retrieved.

An alternative and often faster method, as adopted in other internal deployments, involves utilizing a pre-compiled Amazon Resource Name (ARN) for the library, such as those found in community repositories (e.g., compatible with Python 3.12).

Within the AWS Lambda console, a new Layer, named `pillow-library`, was created. The layer's source was specified as the S3 URL, and crucially, the compatible runtime was set to Python 3.13 to match the function's runtime. The final action was attaching this layer to the ImageConvertToColorScale Lambda function.

3.2.1 Dependency Resolution Refinement

Initial deployment attempts failed due to an incorrect zip file structure and, subsequently, a binary incompatibility issue between the local development environment and the AWS Lambda runtime architecture. To resolve this, the

packaging method was refined based on official AWS documentation. A Docker container was used to mimic the precise AWS Lambda execution environment. The Python packages, including Pillow, were built within this container to ensure that the compiled binaries were compatible with the target runtime architecture, thereby resolving the runtime error.

3.3 Establishing the RESTful API Endpoint

To provide a publicly accessible and testable endpoint for the Lambda function, an Amazon API Gateway endpoint was configured. A new REST API, named "PixelForge API", was created.

A POST method was defined for this API, deemed appropriate as the function generates a new resource (the processed image) upon request. The integration was configured as follows:

- **Integration type**: Set to Lambda Function, indicating that the API Gateway will proxy requests directly to an AWS Lambda function.
- **Lambda Proxy Integration**: Enabled (True), allowing for a simplified integration where the entire request is passed to the Lambda function and the Lambda function's response is returned as-is.
- **Region**: Specified as eu-north-1, the default one.
- **Lambda Function**: Designated as ImageConvertToColorScale, pointing to the specific Lambda function responsible for image conversion.

To ensure seamless handling of various image formats, the API's global settings were adjusted to include support for all binary media types:

```
API settings > binary media types: */*
```

The final step involved deploying the API to a designated stage, named dev (development). Upon successful deployment, the stage's Invoke URL was retrieved, providing the public endpoint for direct testing and interaction with the deployed image processing function.

4 Performance Evaluation Methodology

The primary objective of the performance evaluation was to conduct a systematic analysis of the serverless application's performance and scalability under varying simulated workload conditions.

The general testing procedure followed a structured, sequential approach:

1. To specify the purpose and scope of the evaluation;
2. To identify the features/aspects of the Cloud services that are to be evaluated;
3. To determine the performance metrics that will be analyzed;
4. To select appropriate tool for evaluating performance;
5. To design performance evaluation experiments;

6. To setup an experimental environment;

7. To run performance evaluation experiments.

4.1 To Specify the Purpose and Scope of the Evaluation

The scope of this evaluation is to demonstrate a complete and robust testing capability on the AWS cloud environment. While the specific image manipulation implementation is a well-studied problem, the focus here is on the system-level performance of the serverless architecture. The evaluation aims to answer critical performance questions from a user-centric perspective, simulating the Lambda function's invocation as a public, accessible service.

The study aims to address the following key questions:

- Is there any bottleneck (hot spot functions)?
- Are the resource utilized efficiently or not? How many resources are needed to guarantee a certain level of service when the workload increase
- Does the system grant high availability or not? What is the throughput and the response time?
- Is the system capable of scale when the workload increase?

We try to evaluate the performance of the system from a user perspective, as if the lambda function was to be called as an accessible website.

4.2 To Identify the Features/Aspects of the Cloud Services that are to be Evaluated & To Determine the Performance Metrics that Will be Analyzed

The feature under evaluation is the AWS Lambda function implementing the image processing logic.

The analysis focuses on both user-oriented and system-oriented performance metrics.

User-Oriented Performance Metrics:

- **Response Time**: Measured directly via the Lambda metric "Duration" (i.e., time the answer is received minus time the request is sent).
- **Availability**: Calculated as the ratio of successfully processed requests to the total number of requests, with system throttles indicating unavailability episodes.

System-Oriented Performance Metrics:

- **Throughput**: Evaluated via three composite metrics: NetworkOut/second, Disk*Ops/second, and (Invocation - Errors - Throttles)/second.
- **Resource Utilization** (e.g., memory usage).
- **Scalability** (the system's ability to handle concurrent load).

4.3 To Design Performance Evaluation Experiments

The experiments involve generating requests for the API, studying various request types and intensity (requests per second). Requests were sent directly to the API Gateway endpoint, bypassing any manual image upload interface. Prior to testing, the Lambda function's configuration was updated to allow for increased timeout windows and memory allocation to accommodate potentially longer processing times under load.

Apache JMeter was selected as the appropriate tool for generating and measuring the simulated load. The Apache JMeter™ application is an essential, open-source, 100% pure Java-based framework designed for the quantitative analysis of application and server performance. It serves primarily as a load testing tool to simulate a heavy, concurrent load on a server, group of servers, network, or object, thereby measuring system stability, scalability, and resource utilization under various conditions.

The execution of these experiments provides the empirical data required to assess the application's performance characteristics, which will be detailed in subsequent analysis.

4.3.1 Experimental Setup and Configuration with Apache JMeter

As JMeter is a pure Java application, the first prerequisite was ensuring a compatible Java Development Kit (JDK) environment was installed on the testing host. Subsequently, the JMeter binary distribution was downloaded and extracted.

To configure the test parameters, the JMeter application was initialized in its Graphical User Interface (GUI) mode by executing the startup script (./jmeter.sh in the bin directory) from the command shell. The GUI environment was exclusively used for the Test Plan creation and configuration, a standard practice to minimize resource consumption during the actual load execution.

The Test Plan was meticulously configured with the parameters defined in Table 1:

Number of threads	50
Ramp-up period (seconds)	70
Loop count	infinite
Same user on each iteration	true
Delay thread creation until needed	true
Specify thread lifetime	true
Duration (seconds)	1800

Table 1: Test configuration

To simulate realistic and varied workloads, a CSV Data Set Config element was integrated into the Test Plan. This element was populated with approximately twelve base64-encoded image strings, representing a broad range of input file sizes—from a small 25x25 pixel resolution up to an intensive 8K resolution. This diversity in payload size was necessary to observe the application's performance characteristics and identify any size-dependent processing bottlenecks within the Lambda function.

The final execution of the load test was performed in Non-GUI (Command-Line Interface) mode by invoking the startup script directly in the terminal (e.g., ./jmeter.sh -n -t Test.jmx). This method is the established best practice for load testing, as it prevents the resource overhead of the GUI from skewing the performance metrics, ensuring the results are a true reflection of the system under test.

5 Experimental Results and Performance Analysis

The performance evaluation utilized Amazon CloudWatch metrics captured during the Apache JMeter load test to assess the serverless application's behavior. The test spanned approximately 17 minutes, and the following subsections detail the analysis of the system and user-oriented performance metrics.

5.1 Analysis of User-Oriented Performance Metrics

Duration (Response Time) The stability of the average duration, despite variable input image sizes, suggests that the Lambda runtime successfully hot-started or efficiently scaled to handle the consistent request rate.

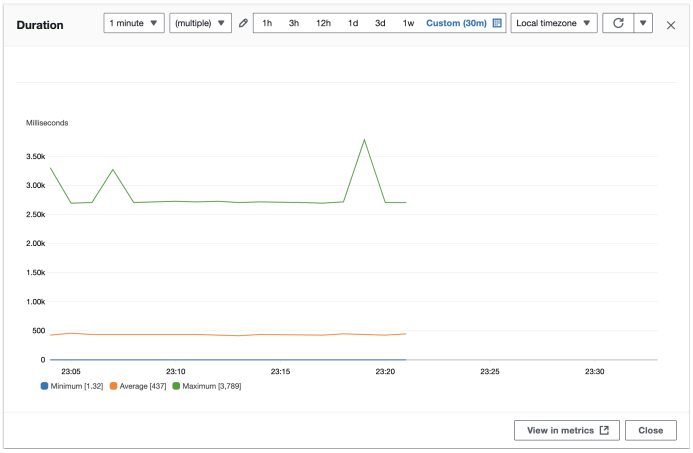

Figure 1: Duration

Availability and Error Rate The Error Count and Success Rate metric (Figure 2) shows the system's reliability. The count of errors was consistently zero throughout the test. The Success rate remained at 100%. This confirms that every request processed by the system was handled successfully by the Lambda function, demonstrating high functional availability and robustness in the core image processing logic.

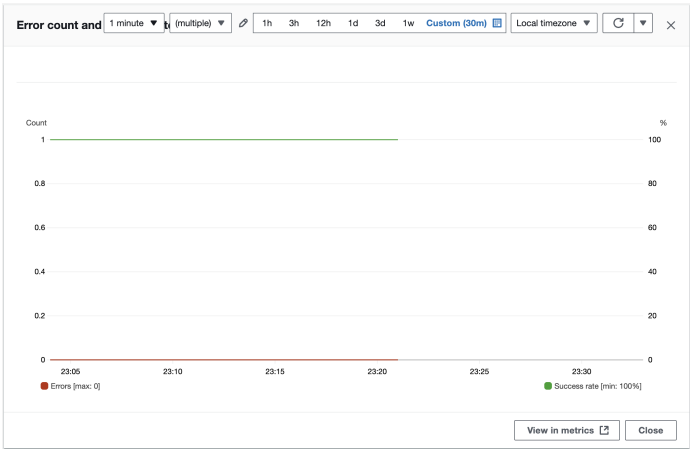

Figure 2: Errors

5.2 Analysis of System-Oriented Performance Metrics

Invocations (Throughput) The Invocations metric (Figure 3) provides insight into the request throughput.

- **Total Invocations**: The test resulted in a sum of 19,388 total function executions.
- **Invocation Rate**: The system maintained a highly consistent invocation rate of approximately 1,000 requests per minute (≈16.7 invocations/second) for the majority of the test.
- **System Capacity**: The sustained high throughput without service-breaking errors confirms the API Gateway and Lambda configuration efficiently managed the simulated load.

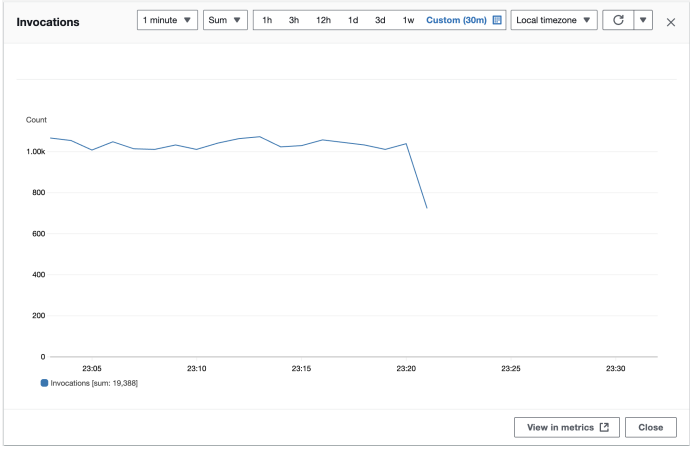

Figure 3: Invocations

Throttles and Scalability The Throttles metric (Figure 4) is critical for assessing scalability and resource limits. The system experienced a significant and consistent pattern of throttling, with peak counts reaching 193 per minute. The consistent fluctuation suggests that the request rate generated by JMeter exceeded the configured concurrency limit of the Lambda function or the default AWS account limits.

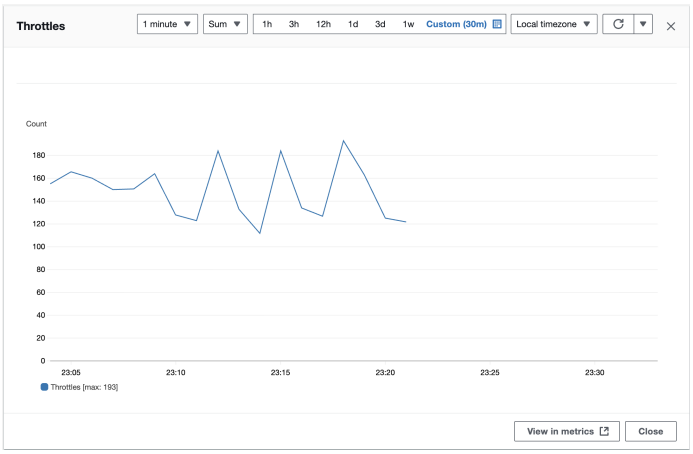

Figure 4: Throttles

5.3 Conclusion of Test Results

The evaluation successfully demonstrated the following performance characteristics of the serverless application:

- **Reliability and Availability**: The system achieved a 100% success rate, confirming the functional stability and robustness of the image processing logic under continuous load.
- **Performance Consistency**: The average response time (Duration) was consistently low (≈437 ms), indicating efficient processing once the Lambda function was initialized.
- **Scalability Bottleneck**: The system's scalability was constrained by the maximum concurrent execution limit (max 10) enforced on the AWS Lambda function. The high rate of throttles demonstrates the system's ability to gracefully degrade service by rejecting excess load, rather than collapsing under stress, which confirms its ability to grant high availability up to its configured capacity.

To improve the overall throughput and eliminate throttling, the recommended action would be to increase the provisioned concurrency limit for the Lambda function.