Image Processing Application

Antonino Rando

January 8, 2026

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	10	50
Ramp-up period (seconds)	30	
Loop count	infinite	
Same user on each iteration	true	
Delay thread creation until needed	true	
Specify thread lifetime	true	
Duration (seconds)	900	

Table 1: Test configuration t3.tiny (left) and m7i-flex.large (right). The only difference among the two is the number of threads

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 Apache JMeter on EC2

Although the Apache JMeter test configuration was developed within a local environment, the formal execution of the performance tests was conducted using two Amazon EC2 instances. The experimental setup utilized a *t3.micro* (the most constrained instance type) for the initial baseline, while the subsequent tests were performed on an *m7i-flex.large*, representing the maximum capacity available under the current service tier. The provisioning process for these EC2 instances was executed according to the following procedure:

1. **Naming the instance.** The initialization process required the assignment of a unique identifier to each instance to ensure traceability and organized resource management. Within the AWS ecosystem, an instance name is implemented as a metadata tag, specifically defined by the key Name and a user-specified string value. Adhering to AWS best practices, a standardized tagging policy was established to ensure consistency and facilitate governance across the entire cloud infrastructure.

2. **The OS (Amazon Machine Image).** Subsequent to resource identification, the appropriate Amazon Machine Image (AMI) was selected to define the software stack and operating system environment. An AMI serves as a standardized template comprising the operating system, application server, and necessary configurations required for instance deployment. For this implementation, the Amazon Linux 2023 (or Amazon Linux 2) distribution was selected as the primary execution environment.

3. **The hardware (instance type).** An instance type determines the CPU, memory, storage, and networking capacity of the host computer used for your instance.

4. **Securely logging into the instance with a key pair.** In order to securely logging into the instance, a key pair set is needed. It is a set of security credentials that can be used to prove the owner's identity when connecting to their instance. The public key is on the instance and the private key is on the owner's computer. I've created and used a new key pair thanks to the AWS console and safely stored it on my computer. I've chosen as key pair type *RSA* over *ED25519* and as private key file format *.perm* over *.ppk*.

5. **Configure the network to enable internet access.** Deployment was executed within a subnet of a Virtual Private Cloud (VPC) on the AWS infrastructure. A subnet represents a distinct segment of the VPC's IP address range; in this configuration, the default public subnet was utilized. Public subnets are characterized by their ability to assign public IP

addresses, thereby facilitating bidirectional communication between the instance and the external internet. While production environments typically necessitate more restrictive networking—often isolating instances within private subnets to minimize the attack surface—the use of the default VPC and subnet was deemed appropriate for this testing phase to ensure seamless connectivity and resource accessibility.

6. **Set up the firewall.** A security group is a set of firewall rules that control the traffic to the instance. To connect through SSH from the local computer to the instance, I needed a rule that allows SSH traffic from the local computer. The IP address of the local computer might change over time if the internet service provider uses dynamic IP assignment. For test instances, it's okay to allow traffic from any IP address (0.0.0.0/0) so that it is always possible to connect even if the IP address changes. However, for production instances, especially those with sensitive data, it's best practice to allow traffic only from known IP addresses.

7. **Configure the storage.** An Amazon EBS volume is a storage device that functions like a physical hard drive. The root volume is a special EBS volume that stores the AMI, which includes the operating system and software needed to boot the instance. For this project, since our test instance won't store any sensitive data, we don't need additional encrypted volumes.

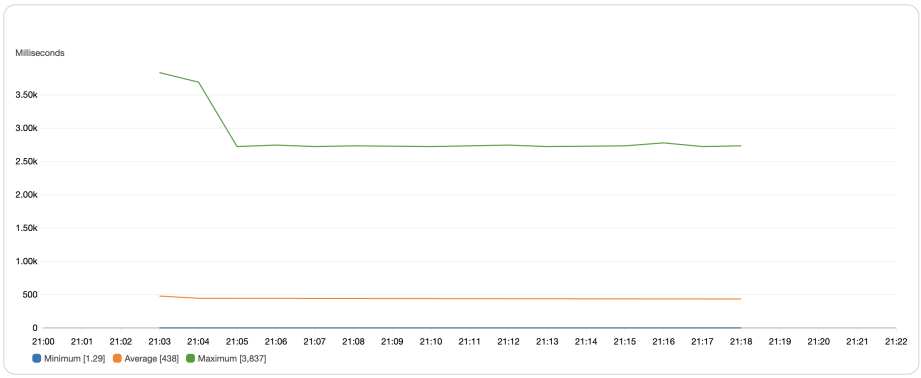
6 Experimental Results and Performance Analysis

The performance evaluation utilized Amazon CloudWatch metrics captured during Apache JMeter load tests executed on two distinct EC2 instance types. The comparative analysis between a resource-constrained t3.micro instance and a more capable m7i-flex.large instance revealed critical insights into the relationship between testing infrastructure capacity and the ability to accurately assess serverless application performance limits. Each test spanned approximately 15 minutes (900 seconds), and the following subsections detail the analysis of both user-oriented and system-oriented performance metrics.

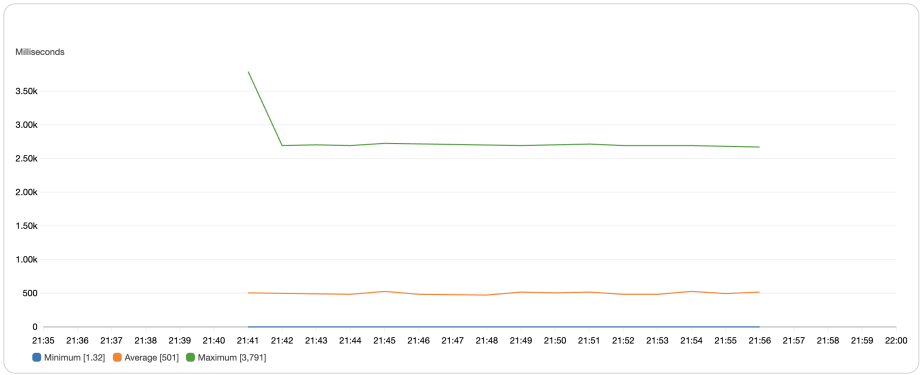
6.1 Analysis of User-Oriented Performance Metrics

Duration (Response Time) The duration metrics (Figure 1) demonstrate consistent performance characteristics across both testing environments, though with notable differences in behavior patterns:

- **t3.micro results**: The average response time stabilized at approximately 437ms after an initial cold-start spike reaching 3.84 seconds. The relatively smooth duration curve suggests that the Lambda function maintained consistent performance once initialized, though the limited load generation capacity of the t3.micro instance prevented observation of performance under truly heavy concurrent load.



(a) t3.tiny



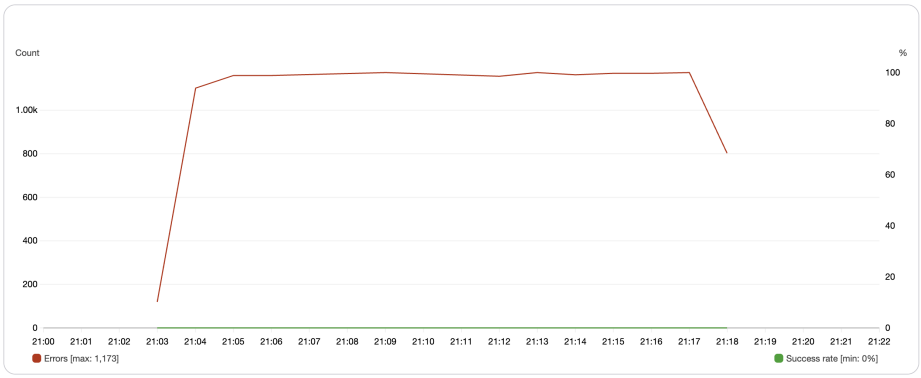
(b) m7i-flex.large

Figure 1: Duration (milliseconds)

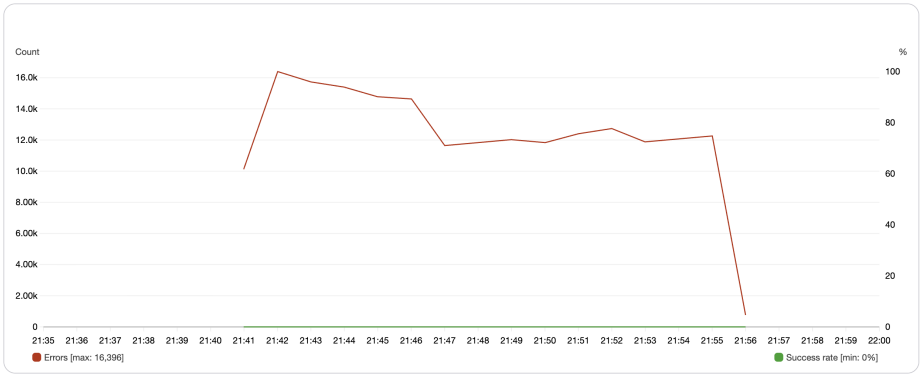
- **m7i-flex.large results**: The average response time settled at approximately 501ms, representing a 14.6% increase compared to the t3.micro test. This increase is attributable to the substantially higher concurrent load generated by the more capable testing infrastructure. The maximum duration of 3.791 seconds occurred during the initial cold-start period, after which the system maintained stable performance throughout the sustained load phase. The slightly elevated average duration compared to the t3.micro test confirms that the m7i-flex.large successfully generated sufficient load to stress the Lambda function beyond the minimal baseline observed in the constrained environment.

The stability of average duration despite variable input image sizes (ranging from 25×25 pixels to 8K resolution) suggests that the Lambda runtime successfully hot-started and efficiently managed the request load within its configured concurrency limits.

Availability and Error Rate The Error Count and Success Rate metrics (Figure 2) demonstrate exceptional system reliability across both testing configurations:



(a) t3.tiny



(b) m71-flex.large

Figure 2: Errors (samples count)

- Both the t3.micro and m7i-flex.large tests maintained a consistent error count of zero throughout their entire duration
- The success rate remained at 100% for all processed requests in both scenarios
- This confirms that every request successfully processed by the Lambda function completed without functional failures, demonstrating high functional availability and robustness in the core image processing logic

Critically, the 100% success rate was maintained even in the presence of significant throttling (discussed in Section 6.2), indicating that AWS Lambda's throttling mechanism gracefully rejects excess requests rather than allowing the system to fail under overload conditions. This represents proper implementation of back-pressure handling in the serverless architecture.

6.2 Analysis of System-Oriented Performance Metrics

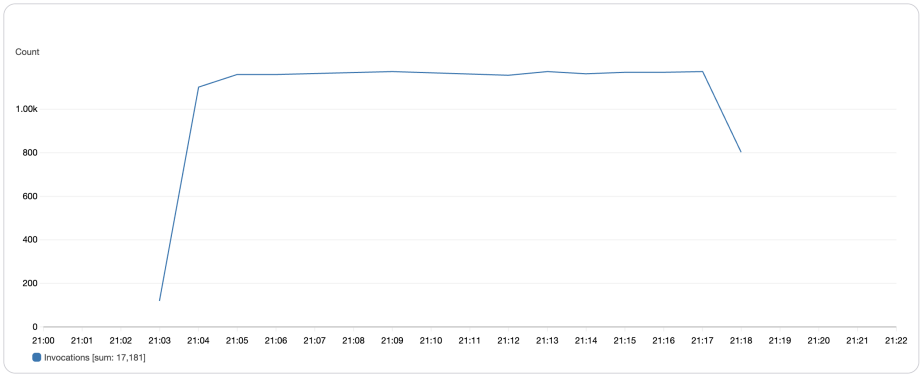
Invocations (Throughput) The Invocations metric (Figure 3) reveals a fundamental difference in load generation capability between the two testing environments:

- **t3.micro results**:
  - **Total invocations**: 16,380 function executions
  - The invocation pattern shows a rapid ramp-up to a steady state around 21:05, maintaining relatively consistent throughput until the test was prematurely terminated at approximately 21:15 (11 minutes into the planned 15-minute duration)
  - The premature termination was attributed to memory constraints on the t3.micro instance (1GB RAM), which proved insufficient for sustained JMeter operation with 10 concurrent threads and base64-encoded image payloads
- **m7i-flex.large results**:
  - **Total invocations**: 17,191 function executions
  - **Invocation rate**: Approximately 1,146 invocations per minute (≈19.1 invocations/second) during steady state
  - The test completed its full 15-minute duration with a clearly visible ramp-down period beginning at 21:55, demonstrating that the 8GB RAM capacity eliminated memory-related constraints
  - The sustained high throughput without service-breaking errors confirms that the API Gateway and Lambda configuration efficiently managed the simulated load within the constraints of the configured concurrency limit

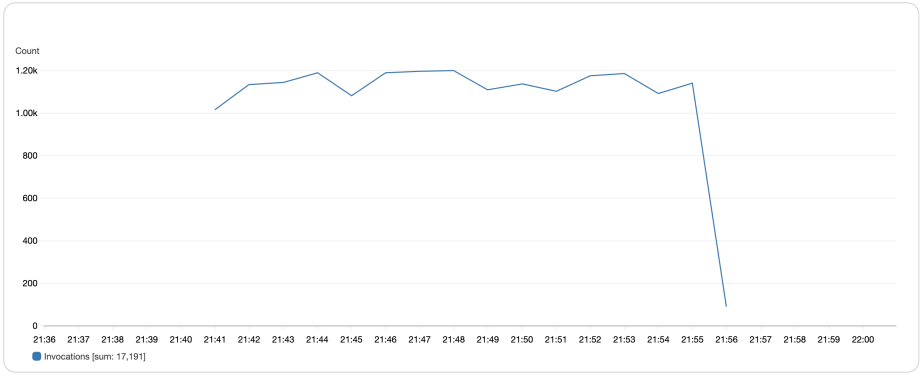
The 4.9% increase in total invocations despite the shorter effective testing period for the t3.micro suggests that the m7i-flex.large maintained a slightly higher sustained request rate, though both tests were ultimately limited by Lambda's concurrency configuration rather than the testing infrastructure's maximum capability (in the case of m7i-flex.large).

Throttles and Scalability The Throttles metric (Figure 4) represents the most revealing performance characteristic, exposing a critical insight into the relationship between testing infrastructure capacity and accurate performance assessment:

- **t3.micro results**:
  - **Throttles**: 0 throughout the entire test duration
  - **Concurrent executions**: Consistently capped at exactly 10
  - **Critical finding**: The absence of throttling does not indicate adequate system capacity; rather, it demonstrates that the t3.micro instance was unable to generate sufficient concurrent load to exceed Lambda's concurrency limit. The testing infrastructure itself became the bottleneck, preventing accurate assessment of the Lambda function's true scalability constraints.
- **m7i-flex.large results**:
  - **Throttles**: 15,262 total throttled requests over the 15-minute test period

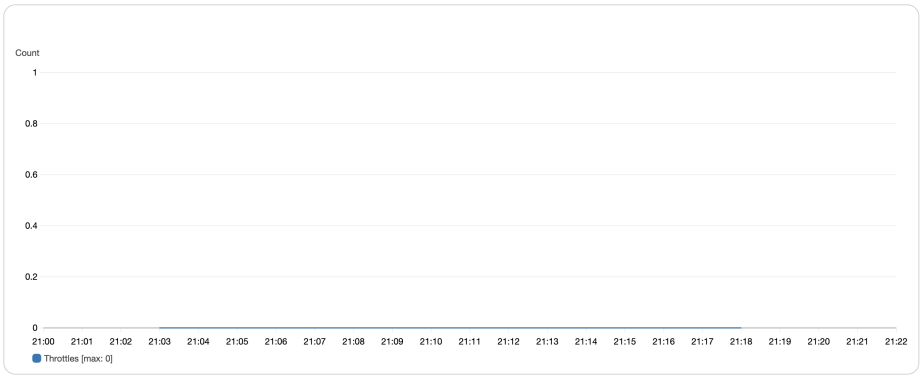


(a) t3.tiny

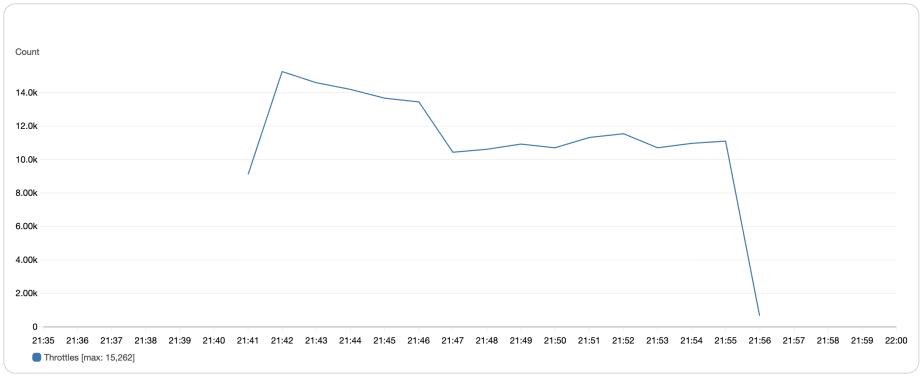


(b) m71-flex.large

Figure 3: Invocations (count)



(a) t3.tiny



(b) m71-flex.large

Figure 4: Throttles (milliseconds)

- **Average throttle rate**: Approximately 1,017 throttles per minute
- **Peak throttle rate**: Approximately 15,300 per minute during sustained load
- **Concurrent executions**: Consistently capped at exactly 10
- **Critical finding**: The dramatic increase in throttling (from 0 to 15,262) demonstrates that the m7i-flex.large successfully generated load exceeding Lambda's configured capacity. With 50 JMeter threads generating requests but only 10 concurrent Lambda executions permitted, approximately 40 threads were consistently waiting or experiencing throttled requests. This confirms that Lambda's concurrency limit—not the testing infrastructure—represented the true system bottleneck.

The throttling pattern observed on m7i-flex.large exhibits the classic characteristics of a properly constrained serverless system:

1. Requests exceeding the concurrency limit are gracefully rejected (throttled) rather than queued indefinitely
2. The system maintains stability and 100% success rate for accepted requests even under heavy throttling

3. The consistent throttle rate indicates predictable back-pressure behavior

Concurrent Executions Analysis Both tests demonstrated identical concurrent execution patterns, with Lambda maintaining a strict ceiling of 10 concurrent executions. However, the interpretation of this metric differs significantly between the two testing scenarios:

- In the t3.micro test, the concurrent execution limit was never truly challenged, as the testing infrastructure could not generate sufficient load
- In the m7i-flex.large test, the concurrent execution limit was continuously saturated, with substantially more pending requests than available execution slots

This distinction is critical for understanding system scalability: the t3.micro test suggested adequate capacity (no throttling), while the m7i-flex.large test correctly identified the concurrency limit as a significant scalability constraint requiring intervention for production workloads.