Cloud Computing Final Project

Image Preprocessing using AWS services

Beyza Nur Elaslan - 2180876

Marthe Elgawly - 2170201

Murat Huseynov - 2181584

Sapienza Universita di Roma

Table of Contents

Abstract

This documentation report outlines the development and implementation of a serverless image preprocessing application using AWS cloud services, emphasizing scalability, efficiency, and simplicity. The application performs automated image transformations—specifically grayscale conversion and resizing—using the Python Pillow library. Images uploaded to a designated Amazon S3 input bucket trigger an AWS Lambda function, which processes each image and stores the output in a separate S3 output bucket.

The serverless architecture ensures high availability and cost-effectiveness by eliminating the need for server management. AWS Lambda automatically scales in response to the volume of incoming files, making the system suitable for varying workloads without manual intervention. The lightweight processing operations are optimized for fast execution and low resource usage, aligning well with Lambda's constraints.

Monitoring and observability are managed with Amazon CloudWatch, which tracks key metrics such as request count, execution duration and, error rates. These insights help maintain operational health, ensure timely responses to anomalies, and support performance tuning as workload demands evolve.

This approach demonstrates the power of combining AWS Lambda and S3 for real-time, event-driven image processing pipelines. The solution provides a flexible and scalable foundation for image preprocessing tasks, paving the way for integration into broader image analysis or machine learning workflows.

1. Introduction

1.1. Project Overview

The aim of this project is to preprocess images using Computer Vision libraries. When images are uploaded to an S3 bucket, an AWS Lambda function—using the Pillow library in Python—is triggered to convert them to grayscale and resize them. The processed images are then stored in a separate S3 bucket. The system is scalable, cost-effective, and designed for real-time, automated image processing in the cloud.

1.2. Computer Vision Tasks

Computer Vision is a field of computer science that focuses on enabling machines to interpret, analyze, and process visual information from images and videos. In this project, the Pillow library in Python is used to perform basic image manipulation tasks as part of the preprocessing pipeline. Pillow provides a wide range of functionalities such as image format conversion, resizing, color adjustments, and filtering, making it well-suited for preparing images for further analysis or storage. These operations are applied automatically within an AWS Lambda function, allowing images to be processed efficiently and consistently in the cloud environment.

1.3. Cloud Scalability and Infrastructure

Cloud computing enables efficient and scalable processing of workloads by leveraging managed services. In this project, AWS S3 and AWS Lambda are used to create a fully serverless image preprocessing pipeline. AWS S3 provides reliable, scalable object storage for handling large volumes of images, while AWS Lambda automatically executes processing logic in response to image upload events without the need for managing servers.

When a new image is uploaded to the input S3 bucket, it triggers a Lambda function written in Python. This function uses the Pillow library to convert the image to grayscale and resize it, and then stores the processed result in an output S3 bucket. This event-driven design allows the system to scale seamlessly with demand, as Lambda automatically handles concurrency and execution based on incoming events.

Monitoring and observability are handled through Amazon CloudWatch, which tracks key performance and operational metrics which help maintain system reliability, detect performance bottlenecks, and support debugging and optimization efforts.

- **Invocations** – Total number of times the Lambda function is triggered.
- **Duration** – Execution time (milliseconds) per invocation, including cold starts.
- **Error Count** – Number of failed executions (exceptions, timeouts, or crashes).
- **Success Rate** – Percentage of invocations that completed without errors.

- **Throttles** – Number of rejected invocations due to exceeding concurrency limits.
- **Total Concurrent Executions** – Peak number of simultaneous Lambda instances running.
- **Asynchronous Event Age** – Delay (seconds) between event queuing and Lambda processing (for async invocations).
- **Asynchronous Events Received** – Number of events queued for async processing.
- **Asynchronous Events Dropped** – Events discarded due to expiration or queue limits.

2. Application Implementation

2.1. Methods Used

The application is developed using Python, leveraging AWS serverless services to automate image preprocessing in the cloud. The core image processing tasks are handled using the Pillow library, a Python imaging toolkit that supports operations such as format conversion, resizing, and grayscale transformation. When a new image is uploaded to the input S3 bucket, it triggers an AWS Lambda function, which processes the image using Pillow and saves the output to a separate S3 bucket.

This serverless design allows the application to remain lightweight, scalable, and cost-efficient, without the need for managing traditional web servers or containerized applications.

2.2. PILLOW Library

Pillow is an open-source Python imaging library that provides a wide range of image processing capabilities. It is integrated into the application to enable automated image preprocessing tasks, such as format conversion, resizing, and color adjustments. Using Pillow within the AWS Lambda function allows the application to efficiently process image files stored in S3 buckets, ensuring they are standardized and optimized for storage or further analysis.

2.3. Functionalities in this project

The application operates entirely within the AWS cloud environment and is fully serverless, Its key functionalities include:

- **Automated Image Upload Detection**: When an image is uploaded to a designated Amazon S3 input bucket, it automatically triggers the image preprocessing workflow.

- **Event-Driven Processing**: The upload event invokes an AWS Lambda function that performs image preprocessing tasks using the Pillow library, such as converting the image to grayscale and resizing it.
- **Processed Image Storage**: The preprocessed image is saved to a separate Amazon S3 output bucket for further use or archival.
- **Serverless Execution**: All functionalities are handled via AWS services, eliminating the need for a traditional server or web application.
- **Scalability and Reliability**: The application can scale automatically with image upload volume, with monitoring and metrics handled via Amazon CloudWatch.

This setup allows for efficient, automated, and highly scalable image preprocessing without the need for manual intervention or a web-based interface.

3. AWS Tools Used

This project uses AWS services to build a fully serverless, scalable image preprocessing pipeline that automatically processes images without the need for server or container management.

These tools include:

- **AWS Lambda**: A serverless compute service used to automatically process images when they are uploaded to an S3 bucket. The Lambda function handles tasks such as converting images to grayscale and resizing them using the Pillow library.
- **Amazon S3 (Simple Storage Service)**: S3 is used to store both the input images and the processed output images. It also serves as the trigger mechanism to invoke the Lambda function when a new file is uploaded.
- **Amazon CloudWatch**: Used for monitoring and observability of the Lambda function. It tracks metrics such as invocation count, execution duration, error rates, and memory usage, helping maintain system health and performance.
- **AWS IAM (Identity and Access Management)**: Ensures secure access control by assigning permissions to the Lambda function to read from the input S3 bucket and write to the output S3 bucket.

3.1. Key Parameters Configured

- **Amazon S3 (Trigger Configuration)**
    - Source Bucket: Stores incoming images to be preprocessed.
    - Destination Bucket: Stores the processed grayscale and resized images.

- Event Type: s3:ObjectCreated:* (triggers the Lambda function when a new object is uploaded).
- Bucket Permissions: Configured to allow Lambda function to read from source and write to destination bucket.

- **AWS Lambda (Function Configuration)**
  - Runtime: Python 3.12
  - Handler: lambda_function.lambda_handler
  - Memory Allocation: 128 MB (adjustable based on image size and processing needs)
  - Timeout: 3 seconds (configurable depending on expected processing time)
  - Layers: Pillow library is included via a custom Lambda Layer to support image processing.
  - Environment Variables:
    - OUTPUT_BUCKET: Name of the destination S3 bucket for processed images
- **IAM Role (Permissions)**
  - Lambda Execution Role:
    - s3:GetObject and s3:PutObject for accessing the S3 buckets
    - logs:CreateLogGroup, logs:CreateLogStream, logs:PutLogEvents for CloudWatch logging

4. Scalability and Availability Test

To evaluate the performance of the image preprocessing pipeline under different workloads, two main testing phases were conducted: Manual Upload Trigger and High Rate Trigger. The Manual Upload Trigger phase involved manually uploading images to the input bucket to initiate processing events. For the High Rate Trigger phase, a set of preloaded images was initially uploaded manually, then programmatically copied into a specific folder within the input S3 bucket via a Python script to simulate rapid, repeated event triggers. Each test processed varying numbers of images at once—specifically 10, 50, 100, 500, and 1000—to assess system scalability and responsiveness. Images were categorized into three resolution levels: low, mid, and high, enabling analysis of how image quality impacts processing time and overall system performance. Both testing phases covered these resolution categories and

input volumes to provide a comprehensive overview of system behavior under different conditions.

For each combination of image resolution (low, mid, high) and input volume (10, 50, 100, 500, 1000), eight key performance metrics were collected.

4.1. Results

The image preprocessing pipeline's performance across different workloads was monitored using Amazon CloudWatch, which tracked crucial metrics such as Invocations, Duration, Error Count, Success Rate, Throttles, Total Concurrent Executions, Asynchronous Event Age, Asynchronous Events Received, and Asynchronous Events Dropped. The observations highlight how the system manages varying image volumes and resolutions, offering valuable insights into its scalability, reliability, and efficiency under both manual upload and high-rate trigger conditions.
To structure the performance comparison clearly, the results are divided into two key analyses for each test phase (Manual Upload Trigger and High Rate Trigger). The first analysis compares input size by evaluating the system's behavior when processing 100 and 1000 mid-resolution images, highlighting how the pipeline scales with increased image volume. The second analysis focuses on resolution, comparing the system's performance when processing 500 images at low and high resolutions. This dual-approach isolates the effects of input size and image quality, allowing for a focused interpretation of the system's scalability and responsiveness.

4.1.1. Manual Upload Trigger

This section presents the performance results from the Manual Upload Trigger tests, where images were uploaded directly to the input bucket to initiate processing. The graphs illustrate the behavior of the system as it processes different volumes of images (10, 50, 100, 500, 1000) at each resolution level (low, mid, high). Each group of eight side-by-side graphs corresponds to one test scenario, showing the key metrics analyzed. This layout enables an in-depth analysis of system responsiveness and stability during manual upload events.

For all resolution levels, tests involving 10 and 50 images do not produce sufficiently detailed trends in the CloudWatch metrics. Owing to the very short processing durations and low number of invocations, the resulting graphs primarily reflect brief, short-term activity rather than sustained patterns. As a result, these cases offer limited analytical value and are less suitable for in-depth performance evaluation. However, starting from 100 images and beyond (100, 500, 1000), the metrics begin to show more pronounced shapes and patterns in the graphs. These visualizations indicate how the system scales with larger input sizes and provide clearer insights into resource utilization and response patterns.

Below are the graphs of the observed metrics when images were manually uploaded to the S3 input bucket, thereby triggering the Lambda function manually.

4.1.1.1. Low Resolution

- 500 images

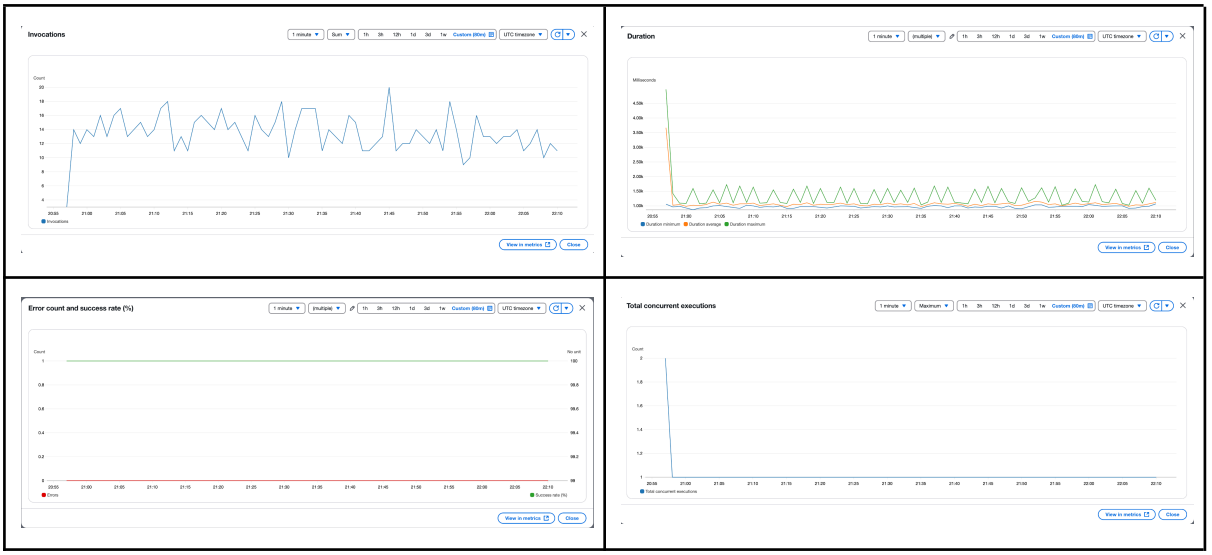
The following graphs illustrate the performance metrics recorded during the processing of 500 low resolution images.

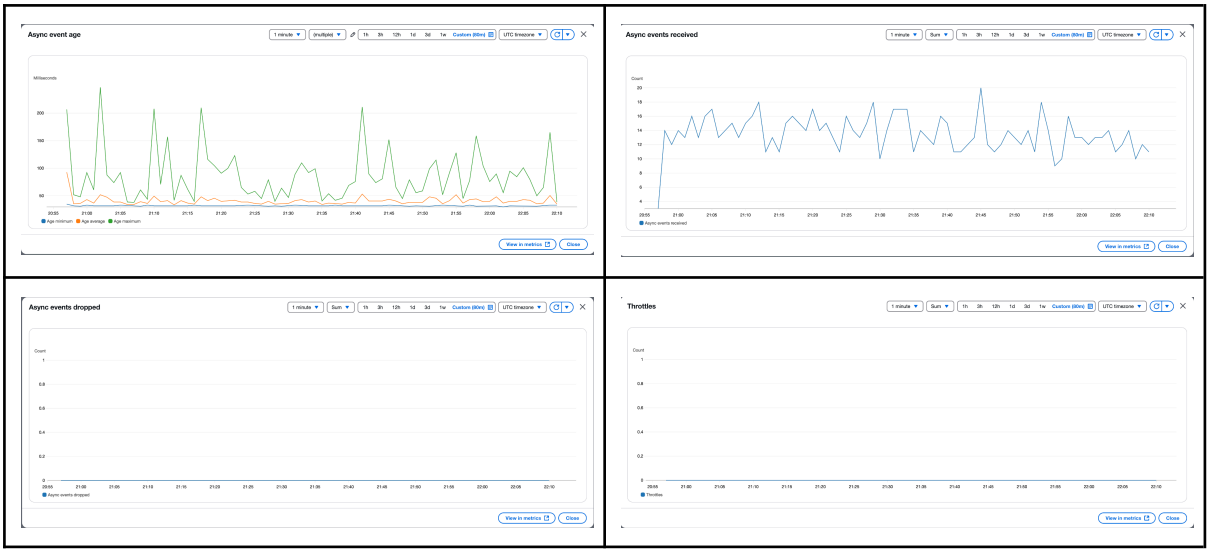
4.1.1.2. Mid Resolution

The following graphs illustrate the performance metrics recorded during the processing of 100, 1000 mid resolution images.

- 100 images

- 1000 images





4.1.1.3. High Resolution

The following graphs illustrate the performance metrics recorded during the processing of 500 high resolution images.

- 500 images

Performance Comparison: Mid-Resolution – 100 vs 1000 Images

This analysis examines the effect of input size on system performance by comparing two test cases involving mid-resolution images processed via manual upload triggers: one with a batch size of 100 images and the other with 1000. The objective is to evaluate how the image processing pipeline scales in response to increasing data volume and whether it maintains reliability, efficiency, and responsiveness under heavier workloads.

1. Invocation Patterns

The number of Lambda invocations directly reflects the input size. The 100-image test produced a short burst of invocations, whereas the 1000-image scenario exhibited a more sustained and evenly distributed invocation pattern over time. This extended invocation period indicates the system's ability to process a larger batch progressively while maintaining throughput.

2. Execution Duration

In both cases, the average execution durations remained within an acceptable range, demonstrating consistent performance. While the 100-image batch showed minimal variation, the 1000-image batch introduced higher peaks in maximum duration—suggesting occasional delays likely due to resource contention or parallel execution overhead. Nonetheless, average durations remained stable, underscoring the Lambda function’s efficiency.

3. Concurrency

The total number of concurrent executions remained low for the 100-image batch, typically not exceeding two parallel instances. In contrast, the 1000-image test prompted Lambda to utilize higher concurrency levels to maintain responsiveness under increased load. This confirms that the system leverages AWS Lambda's concurrency model effectively during high-volume processing.

4. Asynchronous Events Received

Although the trigger was configured for synchronous execution, the CloudWatch logs captured asynchronous events in the 1000-image case, likely due to the S3 service internally batching uploads in rapid succession. This behavior was minimal for the 100-image batch but more noticeable for 1000 images, without impacting performance.

5. Throttles and Dropped Events

No throttling or dropped events were observed in either test case. This indicates that the current configuration (memory, timeout, and concurrency limits) is well-provisioned for the tested workloads, even when scaling to 1000 images.

6. Success Rate and Error Count

Both scenarios achieved a 100% success rate, with no recorded invocation failures or processing errors. This result confirms the stability and robustness of the pipeline across varying input sizes.

7. Asynchronous Event Age

The event age remained negligible for the 100-image test. For 1000 images, there were slight increases in event age, though still within the millisecond range—indicating that Lambda functions were triggered almost immediately after object creation, even at scale.

In summary, the comparison clearly demonstrates that the system scales effectively with increased input volume. While larger batches introduce minor fluctuations in execution time and concurrency, the processing remains stable and efficient. No throttling or reliability issues were encountered, underscoring the architecture's suitability for scalable image processing tasks using AWS Lambda and S3.

Performance Comparison: 500 Low-Resolution vs 500 High-Resolution Images

This analysis evaluates the impact of image resolution on system performance by comparing two test cases processed via manual upload triggers: one involving 500 low-resolution images and the other 500 high-resolution images. The objective is to assess how resolution affects processing efficiency, resource utilization, and overall system stability under identical input size.

1. Invocation Patterns

The 500 low-resolution image test exhibited a rapid burst of invocations, with counts peaking around 60 invocations per minute. This high-frequency triggering reflects the shorter processing time per image, allowing for fast sequential or parallel execution.

In contrast, the 500 high-resolution test displayed a much more gradual and consistent invocation rate, typically ranging between 1 and 6 invocations per minute. This pattern indicates increased processing time per image, spreading invocations over a longer duration.

2. Execution Duration

The execution times for low-resolution images were relatively stable, with average durations around 900–1200 milliseconds. This consistency suggests minimal variability in workload per invocation.

On the other hand, high-resolution image processing resulted in higher and more variable durations. The average execution time exceeded 2000 milliseconds, with peaks reaching up to

6000 milliseconds, indicating greater computational demand and potential variation in image complexity.

3. Concurrency

Concurrency levels were higher for low-resolution images, peaking at three parallel executions. This indicates that the Lambda function was able to process multiple images simultaneously, capitalizing on the short runtime.
In the high-resolution scenario, concurrency remained consistently low—typically at one or two—suggesting that the longer function durations limited the system's ability to scale concurrent executions efficiently within the same time window.

4. Asynchronous Events Received

In both scenarios, the number of asynchronous events closely mirrored the invocation patterns. The low-resolution batch showed a dense and bursty event reception pattern, while the high-resolution batch exhibited a slower, more linear influx.
This behavior is consistent with the input image resolution affecting the rate at which S3 events were generated and processed.

5. Throttles and Dropped Events

Minor dropped events were observed: one in the low-resolution test and two in the high-resolution test. No throttling was recorded in either case, suggesting that the provisioned concurrency and function limits were sufficient for the given workloads.
These dropped events did not affect success rates, indicating resilience in event handling.

6. Success Rate and Error Count

Both scenarios encountered a few transient errors at the beginning of the processing window—likely due to cold starts or initial load—but quickly stabilized, achieving a 100% success rate throughout the remainder of the execution period.
This result reflects strong system reliability regardless of image resolution.

7. Asynchronous Event Age

The event age in the low-resolution test showed initial spikes but quickly stabilized to near-zero values. The high-resolution test had similar early peaks, with maximum event age reaching around 190,000 milliseconds, but also leveled off shortly thereafter.

This indicates that Lambda functions were effectively invoked without long queuing delays, even under heavier per-image load.

In summary, the comparison clearly highlights that image resolution significantly impacts processing behavior and resource utilization. Low-resolution images result in faster execution, higher concurrency, and dense invocation patterns, whereas high-resolution images demand more compute time, leading to longer durations and steadier invocation rates. Despite these differences, the system successfully maintained performance and reliability across both test cases, confirming its robustness and adaptability for variable image processing workloads using AWS Lambda and S3.

4.1.2. High Rate Upload Trigger

This section presents the performance results from the High Rate Trigger tests, where a set of preloaded images—initially uploaded manually—was programmatically copied into a designated folder within the input S3 bucket using a Python script. This approach simulates a rapid and continuous input of images, triggering a high volume of events in a short period and allowing evaluation of the system's responsiveness under sustained load.

As in the Manual Upload Trigger tests, the system was evaluated across varying input volumes (10, 50, 100, 500, and 1000 images) and image resolutions (low, mid, high).

As previously observed in the Manual Upload Trigger section, tests involving 10 and 50 images across all resolution levels result in minimal fluctuations within the CloudWatch metrics. Due to the short-lived processing times and low invocation volumes, the generated graphs typically reflect only short-term activity rather than continuous trends. This limitation reduces the depth of analysis that can be performed for these smaller-scale scenarios. However, at higher volumes (100 images and above), the graphs begin to reveal more significant metric fluctuations and trends, offering valuable insights into how the system handles concurrent executions, async event handling, and scaling behavior under high-rate input conditions.

The same layout is used here to enable direct comparison with the Manual Trigger results, while emphasizing the differences introduced by the higher event frequency and concurrency.

Below are the graphs of the observed metrics when images were uploaded to the S3 input bucket via a script, triggering the Lambda function at a high rate.

4.1.2.1. Low Resolution

- 500 images

The following graphs illustrate the performance metrics recorded during the processing of 500 low resolution images.

4.1.2.2. Mid Resolution

The following graphs illustrate the performance metrics recorded during the processing of 100, 1000 mid resolution images.

- 100 images


Invocations
Duration
Error count and success rate (%)
Total concurrent executions
Async event age
Async events received
Async events dropped
Throttles

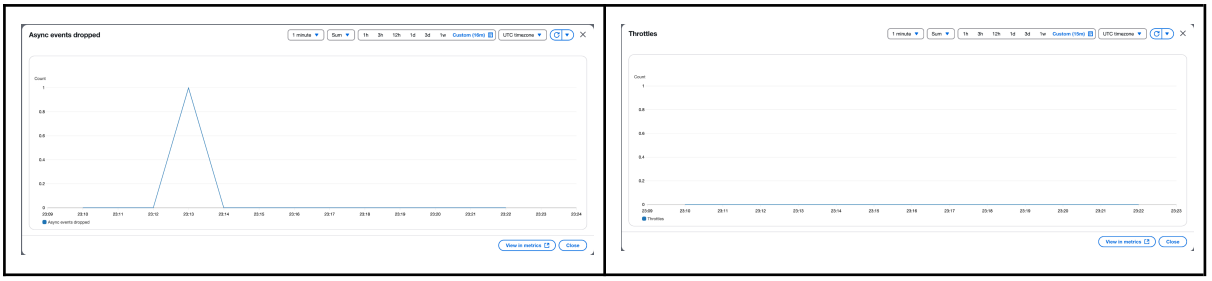
- 1000 images


Invocations
Duration
Error count and success rate (%)
Total concurrent executions

4.1.2.3. High Resolution

The following graphs illustrate the performance metrics recorded during the processing of 500 high resolution images.

- 500 images



Performance Comparison: 100 vs 1000 Mid-Resolution Images

This analysis focuses on evaluating the system's performance under high-frequency event triggers by comparing two test cases: one processing 100 mid-resolution images and the other processing 1000 images. In both scenarios, the images were uploaded to the S3 bucket in rapid succession using a scripted trigger to simulate high-throughput workloads. The objective is to assess the system's scalability, responsiveness, and robustness under intensified load conditions.

1. Invocation Patterns

The 100-image test produced a brief spike in Lambda invocations, peaking quickly before declining within a short window. In contrast, the 1000-image scenario demonstrated a more consistent and sustained invocation pattern, maintaining steady invocation rates over an extended period. This pattern reflects effective batching and distribution of incoming events, showcasing the system's capability to handle prolonged high-load scenarios without interruption.

2. Execution Duration

Both test cases maintained stable average execution durations, indicating consistent function performance under varying input volumes. However, maximum durations were higher in the 1000-image test, suggesting minor delays possibly related to queuing or resource contention. Notably, the variation in execution time was more controlled compared to the manual trigger tests, demonstrating improved consistency in handling event bursts.

3. Concurrency

The total concurrent executions for both scenarios remained within low-to-moderate levels (ranging between 2 to 3), with the 1000-image case displaying a more sustained concurrency. This indicates that the Lambda service appropriately adjusted parallel execution to match the increased load, thereby maintaining throughput without overutilization.

4. Asynchronous Events Received and Dropped

In both tests, asynchronous events were recorded despite the synchronous nature of the trigger configuration. The 1000-image case showed a steady and sustained inflow of async

events, while the 100-image case presented a short spike. Additionally, dropped async events were observed in both cases, albeit minimally (only one dropped event per scenario), indicating that while the system is largely reliable, momentary queuing pressure or processing limits may lead to infrequent loss during high burst periods.

5. Throttling

No throttling was observed in either test case. This affirms that the configured concurrency and memory settings were adequate to process the load without exceeding AWS Lambda's service limits, even under rapid, high-frequency invocations.

6. Success Rate and Error Count

The 100-image scenario experienced a brief moment of reduced success rate, but quickly stabilized to 100%. The 1000-image test also encountered transient errors during the initial phase of processing, followed by full recovery to a consistent 100% success rate. These observations highlight the system's ability to recover autonomously from short-lived errors in high-load conditions.

7. Async Event Age

In both cases, a noticeable increase in event age was recorded at the start of the test, especially in the 1000-image case. Maximum event ages exceeded 150,000 milliseconds before sharply declining and stabilizing. This delay reflects initial backlogs in the processing queue but does not persist, indicating that the system successfully catches up after the initial spike in traffic.

In summary, the comparison demonstrates that the image processing pipeline can scale effectively under high trigger rates, with stable invocation patterns and consistent performance across increasing input volumes. While the 1000-image scenario introduced more pronounced variation in execution duration and event age, these effects were transient and self-correcting. The architecture remained resilient, avoiding throttling, minimizing dropped events, and achieving full processing success—affirming its suitability for high-throughput use cases involving AWS Lambda and S3.

Performance Comparison: 500 Low-Resolution vs. 500 High-Resolution Images

This analysis evaluates the system's performance under high-frequency event triggers by comparing two test scenarios: one processing 500 low-resolution images and the other processing 500 high-resolution images. In both cases, the images were uploaded to an S3 bucket in rapid succession using a scripted trigger to simulate intensive load conditions. The

purpose of this analysis is to assess the system's responsiveness, throughput, and fault tolerance in handling varying payload sizes under uniform burst pressure.

1. Invocation Patterns

The low-resolution test displayed a sharp and stable spike in Lambda invocations, peaking at approximately 50 invocations per minute and maintaining this rate consistently during the peak window. The high-resolution test, by contrast, showed a slightly lower peak of around 42 invocations per minute with more fluctuation throughout the execution window. This difference likely results from increased processing latency per event caused by larger image sizes, which limits how frequently new events can be concurrently invoked.

2. Execution Duration

Execution duration metrics showed clear divergence between the two configurations. In the low-resolution test, average durations were around 1300 to 1600 milliseconds, with a maximum of roughly 4500 milliseconds. The high-resolution test exhibited significantly higher averages ranging from 2500 to 2700 milliseconds, and a peak duration reaching up to 6400 milliseconds. This increase reflects the heavier computational demands required to process high-resolution image data but remains within acceptable thresholds, indicating sustained performance without timeout risks.

3. Concurrency

In terms of concurrent executions, the low-resolution test maintained a peak concurrency level between 2 and 3. For the high-resolution test, concurrency peaked at 4 during the initial high-load period before stabilizing at 3. These results demonstrate AWS Lambda's ability to scale execution threads in response to payload complexity while avoiding resource saturation.

4. Asynchronous Events Received and Dropped

Both test cases showed a steady pattern of asynchronous event receipt aligned with the invocation timeline. The number of async events received matched the expected throughput for both scenarios. Each test recorded one dropped asynchronous event, suggesting that minor event loss can occur during burst peaks, possibly due to temporary queue congestion. Despite this, the system demonstrated reliable handling of high event volumes with minimal data loss.

5. Throttling

No throttling events were observed in either test scenario. This confirms that the configured concurrency, memory, and execution time settings were sufficient to absorb and process the

entire event stream without exceeding service limits imposed by AWS Lambda. The absence of throttling is a strong indicator of an adequately tuned serverless infrastructure.

6. Success Rate and Error Count

Both tests experienced brief instances of error spikes at the beginning of the execution window. However, these were transient and quickly resolved. The success rate recovered to a stable 100% in each case. This behavior is consistent with typical Lambda cold start characteristics and demonstrates the platform's resilience and automatic error recovery under pressure.

7. Async Event Age

The maximum async event age for both tests exceeded 170,000 milliseconds during the early minutes of execution, indicating a short backlog of queued events at the onset of the trigger spike. This delay rapidly decreased as the system caught up with processing demand. The ability to quickly reduce event age after a surge confirms the system's capability to recover from initial bursts and return to steady-state operation without persistent lag.

In summary, this comparison illustrates that the Lambda-based image processing pipeline is capable of handling large, high-frequency workloads with both low and high-resolution images. While the high-resolution scenario exhibited longer execution durations and slightly reduced invocation throughput, the system adjusted effectively through concurrency scaling and preserved overall service reliability. Neither test case encountered throttling, and error recovery was autonomous and swift. These results affirm that the architecture is robust and scalable, suitable for real-time or near-real-time processing of variable-sized media inputs under heavy, burst-style loads.

Performance Comparison: 500 High-Resolution Manual vs. High-Rate Trigger

This analysis examines the performance of the serverless image processing pipeline under two distinct execution conditions, each involving the upload of 500 high-resolution images to an AWS S3 bucket. The first scenario uses a high-rate scripted trigger to simulate rapid event generation, while the second scenario employs a manual upload strategy spread over a longer time window. The objective is to assess and contrast system behavior in terms of throughput, concurrency, latency, and overall resilience.

1. Invocation Patterns

The high-rate trigger test generated an intense and concentrated burst of Lambda invocations, reaching and sustaining over 40 invocations per minute across a short duration of

approximately 13 minutes. This behavior aligns with a scripted mass upload scenario and demonstrates Lambda's ability to scale aggressively in response to rapid input.

In contrast, the manual trigger test exhibited a more fragmented and prolonged invocation pattern, with invocation counts fluctuating between 1 and 6 per minute and spread across a longer period of more than 1.5 hours. This staggered arrival allowed the system to process events more gradually but introduced variability in load management.

2. Execution Duration

Both scenarios maintained relatively stable average execution durations between 2000 and 2800 milliseconds, suggesting consistent processing efficiency. However, the manual trigger test exhibited greater overall volatility across the time window, with multiple minor spikes in maximum duration values.

The high-rate trigger test, on the other hand, showed a brief spike in execution duration at the start, with maximum durations exceeding 6000 milliseconds, followed by a period of stabilization. This spike likely corresponds to initial queuing or cold-start overhead caused by the rapid influx of events.

3. Concurrency

The high-rate trigger scenario reached a peak concurrency of 4 simultaneous Lambda executions, reflecting the system's adaptive scaling in response to the concentrated invocation load.

The manual trigger scenario maintained a lower concurrency level, never exceeding 2 simultaneous executions. This is consistent with the test's staggered event submission pattern, which allowed AWS Lambda to process events sequentially with minimal parallelism.

4. Asynchronous Events Received and Dropped

The high-rate test displayed a pronounced burst in asynchronous events received, mirroring the elevated invocation rate. In contrast, the manual test showed smaller, less frequent spikes in async event reception.

Both scenarios experienced minimal event loss. The manual trigger test recorded 2 dropped async events, while the high-rate trigger test recorded 1 dropped event. These drops occurred early in each test, likely due to brief pressure on the event queue during peak load.

5. Throttling

No throttling events were observed in either test. This indicates that the system's provisioned concurrency and memory configurations were sufficient to support both usage patterns without exceeding service limits.

6. Success Rate and Error Count

In the manual trigger test, 2 transient errors were recorded during the initial phase, after which the success rate recovered to 100%.

The high-rate test showed similar behavior, with sporadic errors in the first few minutes, followed by full recovery. These brief drops in success rate are likely tied to cold starts or initialization delays during sudden invocation spikes.

7. Async Event Age

The manual trigger test exhibited a maximum async event age of approximately 190,000 milliseconds, indicating an initial backlog in event processing that gradually resolved.

In the high-rate scenario, the maximum async event age peaked around 165,000 milliseconds, followed by immediate normalization. These age spikes were short-lived and correlated with the respective peaks in async event intake.

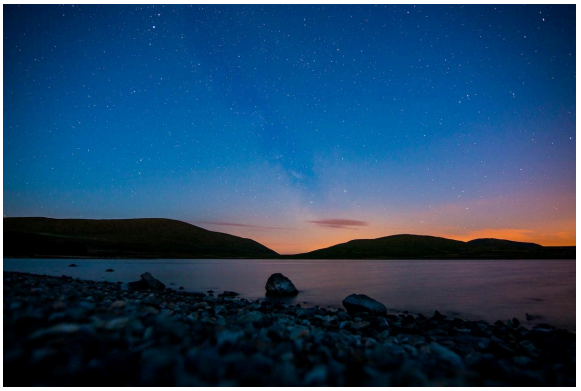
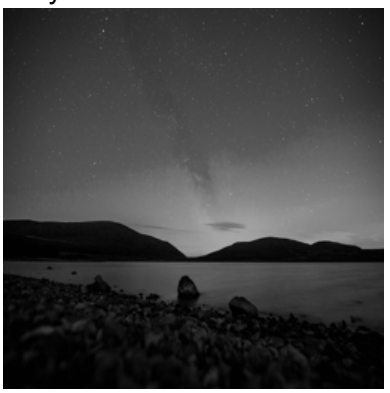
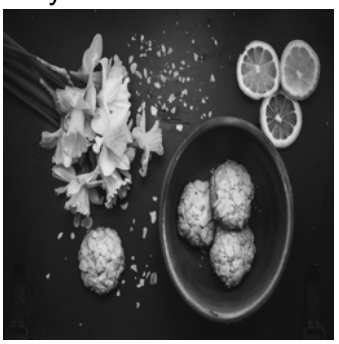
In summary, the results reveal distinct operational behaviors influenced by trigger strategy. The high-rate trigger executed a rapid, high-throughput workload that AWS Lambda scaled efficiently to accommodate, showing higher concurrency, consistent invocation rates, and faster overall test completion. The brief but intense load resulted in minimal event age delay and transient errors, both of which were promptly resolved.

Conversely, the manual trigger test induced a prolonged, low-concurrency processing period. While this spread the system load over time, it led to greater variability in execution duration and slightly higher async event age and event loss.

Ultimately, both approaches demonstrated strong performance and error recovery capabilities. However, the high-rate trigger scenario yielded faster throughput and better resource utilization, validating its efficiency in time-sensitive, high-volume workloads. The manual trigger offers a more conservative alternative suited to use cases where controlled pacing is preferred over peak throughput.

4.1.3. Outputs

This section presents examples of input images alongside their corresponding outputs generated by the project pipeline. Samples are provided for each resolution category—low, mid, and high—to demonstrate how the system processes images of varying quality. These examples illustrate the effectiveness and consistency of the preprocessing across different image resolutions.

Resolution	Input	Output
Low	Size: 1200 x 800 RGB 	Size : 256 x 256 Grayscale 
Mid	Size: 1500 x 997 RGB 	Size : 256 x 256 Grayscale 
High	Size: 2040 x 1536 RGB 	Size : 256 x 256 Grayscale 

5. Conclusion

This project showcases the development of a fully serverless image preprocessing workflow using cloud computing services. Utilizing AWS Lambda and Amazon S3, the system performs automatic image processing operations—such as grayscale conversion and resizing—whenever new images are uploaded, eliminating the need for managing servers or containers.

The solution is built with Python and employs the Pillow library for handling image transformations. Amazon S3 is used to store both the original and processed images, offering a scalable and secure storage mechanism. AWS Lambda handles the compute layer, automatically scaling based on event triggers and offering a cost-effective execution model.

Operational performance is tracked using Amazon CloudWatch, which monitors essential metrics like execution time, errors, and function invocations, enabling visibility and prompt issue resolution.

Overall, the project effectively demonstrates how serverless architecture and managed AWS services can be combined to build a flexible, efficient, and scalable image processing pipeline. This lays a strong groundwork for future improvements or scaling to handle more advanced processing requirements.

Further Notes

The following link contains all images corresponding to our testing process. Each innermost folder includes 8 graph images representing the 8 performance metrics observed during project implementation. The folder structure is organized as follows:

Link: https://drive.google.com/drive/folders/190IXUecx4ZmLunxUyIfHdl-hzswfxg6B?usp=sharing

Manual Upload Trigger

- **Low Resolution**
  - 10 images – 8 Graph Images
  - 50 images – 8 Graph Images
  - 100 images – 8 Graph Images
  - 500 images – 8 Graph Images
  - 1000 images – 8 Graph Images

- **Mid Resolution**
  (Same structure as above)
- **High Resolution**
  (Same structure as above)

High Rate Trigger

- **Low Resolution**
  (Same structure as above)
- **Mid Resolution**
  (Same structure as above)
- **High Resolution**
  (Same structure as above)