Deploying an Image-Generation Application on AWS Lambda: Performance and Scalability Evaluation

Riccardo De Sanctis 1937859
Matteo De Sanctis 1937858

October 6, 2025

Abstract

This report describes the design, implementation, deployment and performance evaluation of an image-generation application deployed on AWS Lambda. The system generates PNG images from textual prompts, uploads to S3 and return its url (or its base64-encoding), and was stress-tested with Locust from an EC2 instance. Experiments measured latency, throughput and cost across multiple workload scenarios. The document contains methodology, results placeholders and recommendations.

Contents

1 Introduction

Image generation has become increasingly popular in recent years. As its demand grows, service providers must find effective ways to meet the needs of many customers. Generative image models are computationally expensive and traditionally run on GPUs. This project explores the feasibility of deploying a compact diffusion model in an AWS Lambda environment (CPU-bound in our implementation) and evaluates performance and scalability by subjecting the service to controlled load tests.

2 Objectives

1. Implement an AWS Lambda function that generates images from prompts using a pre-trained `tiny-sd` model, saving the generated images into the cloud with AWS S3.
2. Build the AWS Lambda function by containerizing the function runtime with Docker and uploading the resulting image to AWS ECR.
3. Design and run load tests with (Locust) from an AWS EC2 instance to evaluate metrics, both for server-side and client-side, under different workload patterns.
4. Collect CloudWatch metrics and client-side logs, to assess function feasibility, system performance and scalability, producing analysis and recommendations.

3 System Architecture

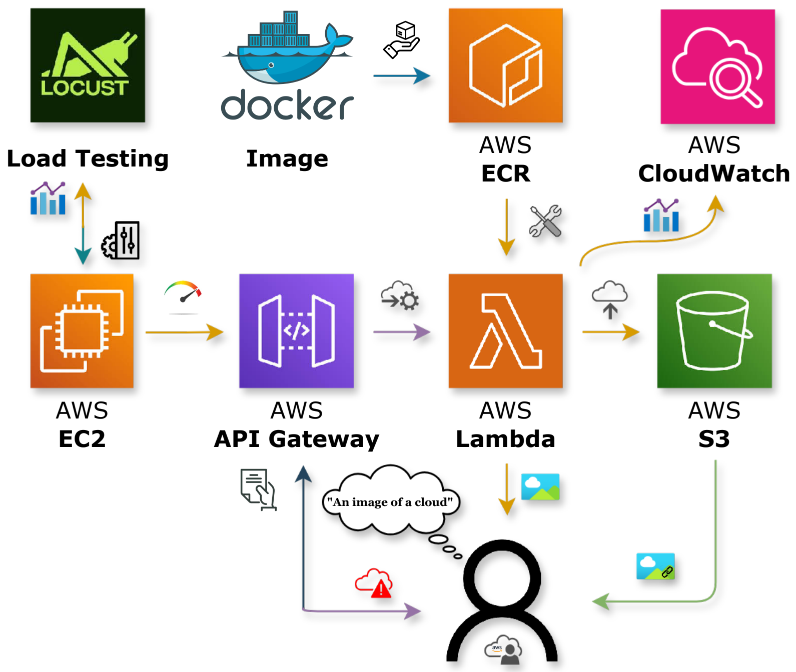


Figure 1: Abstract overview of the system architecture.

3.1 Components

- **AWS Lambda**: the function that generates pictures.

- **AWS ECR**: stores the Lambda container image and its code. The image is generated and built with Docker.
- **S3**: storage for generated images.
- **AWS Lambda URL** (API Gateway): HTTP ingress for the Lambda function.
- **AWS EC2 (Locust)**: load generator running Locust load tests inside the AWS perimeter.
- **AWS CloudWatch**: collects Lambda metrics and statistics of the load test experiments.

4 Implementation

This section summarizes the key implementation details and design decisions.

4.1 Lambda function

- Supports per-request `width`, `height` (clamped to safe limits) and returns either a base64 image encoding or uploads to S3 when `STORE` environmental variable is `true`.
- It saves generated images into a S3 bucket. Saving and bucket endpoint can be specified as Lambda environment variables.
- It uses `DPMSolverMultistepScheduler` and a configurable amount of denoising pipeline steps that can be set with Lambda environment variables (steps parametrize the image quality and the resulting generation time required).
- Enforces safe limits to prevent extremely large CPU/memory usage (e.g., max img size or pipeline steps) or concurrency limit break (10 concurrencies-AWS student limit).
- Caches the Stable Diffusion pipeline in a global variable `pipe` to mitigate repeated cold initialization.

4.2 Dockerfile and model packaging

A container based on `public.ecr.aws/lambda/python:3.12` was used. The tiny-sd model was downloaded and saved into `/opt/models/tiny-sd` during image build so the runtime load does not fetch model weights at invocation time.

4.3 API contract

The Lambda accepts a JSON body as follows:

```
{
  "prompt": "mountains and rivers at sunset",
  "width": 256,
  "height": 256
}
```

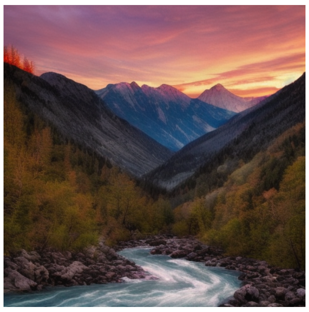


Figure 2: Generated image for given prompt.

The response is a JSON array of objects either containing `image_base64` (base-64 image byte encoding) or (when images are saved to the S3 bucket) `image_url` pointing to the bucket.

5 Testbed and Methodology

5.1 Load generator

Locust is run on an EC2 instance performing the synthetic test workloads. The EC2 instance is based on an Amazon Linux AMI, the instance type is t3.small with 2vCPU and 2GiB of memory . The representative type of users were implemented (Small/Medium/Large) targeting 64×64, 128×128 and 256×256 image sizes respectively. For the final test, a warm-up/scale-up/steady/scale-down test was performed to test scalability and different load level intensities.

5.2 Workload patterns

EC2 executes a ramp-up scenario followed by a steady and ramp-down per experiment. Steps used in experiments are as follows:

- **Warm-up** and **ramp-up:** increasing users every 40 seconds, up to 10 users.
- Long **steady-state**: 14 minutes with 10 concurrent users.
- Short **ramp-down** steps to observe recovery and cool-down behaviour. Decreasing one user per minute for 10 minutes.
- Repeated experiments (5 iterations) to reduce noise and to improve statistical confidence.

Note: we also conducted experiments with up to 5 users under the same settings; nonetheless, we omit those results as the 10-user experiments generalize them and to keep the presentation concise. Also, maximum 10 concurrent running Lambda execution instances are permitted by AWS Academy Learner Lab policy, therefore we set our Lambda function reserved concurrency to 10 and limited the Locust users to 10.

5.3 Metrics collected

Client-side (Locust): Requests and request rate (RPS), response time (RT) and response time average, maximum, minimum and percentiles (p50/p90/p99), failures, and average content size.

Server-side (CloudWatch): Invocations, errors, throttles, duration, and concurrent executions.

6 Reproducibility

All code used is attached to this report; reproducibility pipeline can be summarized as follows:

1. Build and push the Docker image to AWS ECR.
2. Create Lambda from the container image. Set environment variables: `PIPELINE_STEPS` to specify the denoising steps required, `S3_BUCKET` with the name of the bucket and `STORE` (a flag) to save images to the S3 bucket. Set reserved concurrency to 10 (to be sure to avoid surpassing AWS learner lab limit and get banned). Set timeout at least to 3 minutes. Set memory size at least to 4096MB, ephemeral storage can stay as default. These settings mainly account for cold starts.
3. Expose the Lambda via Lambda URL (or API Gateway) and note the endpoint.
4. Launch an EC2 instance, install pip3, Locust, Boto3 and AWS CLI. Update the `ENDPOINT` variable in `locustfile.py`. For convenience, we recommend connecting to the EC2 instance via Remote SSH from an editor such as VS Code.

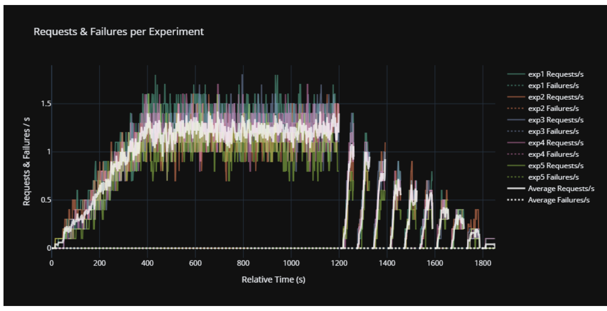
5. Run the provided bash test script to execute Locust scenarios and fetch CloudWatch metrics.

7 Results

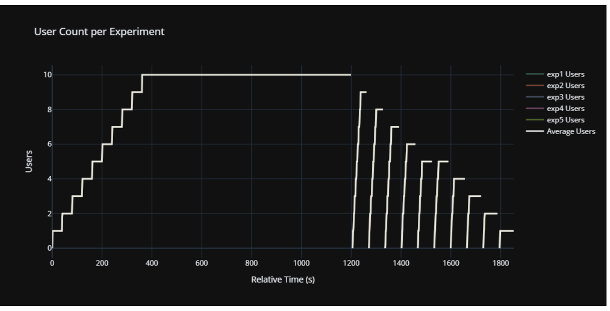
7.1 Locust - Client side metrics

Table 1: Statistics summary per image size average across the five experiments.

Image size	Requests	Requests/s	Failures	Min RT	Median RT	Avg RT	Max RT	Avg Content Size	p50 (ms)	p90 (ms)	p99 (ms)
64x64	698.8	0.24	0.0	2048.08	2608.65	3929.82	24796.29	132.43	2651.30	6455.83	23223.50
128x128	699.6	0.23	0.0	3464.77	4198.34	5998.71	29616.81	132.06	4229.71	10213.78	21621.21
256x256	82.4	0.05	0.0	9445.61	10743.46	13695.71	43950.69	133.10	10973.60	19834.52	36852.66
Aggregated	1480.8	0.48	0.0	2048.08	3959.93	5443.14	43.950.69	132.29	3970.93	10883.94	36156.31

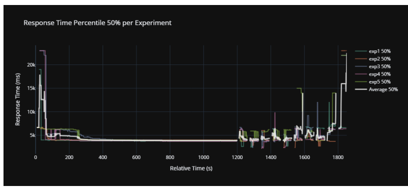


(a) Client side - Requests & Failures.

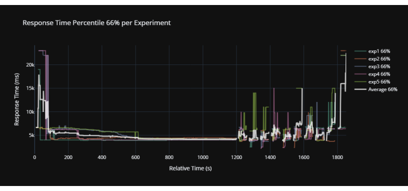


(b) Client side - User Count.

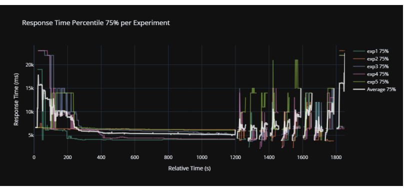
Figure 3: Client side - Throughput vs Time.



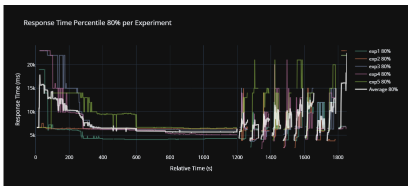
(a) Percentile 50%.



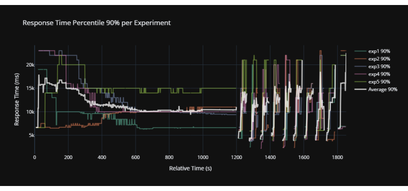
(b) Percentile 66%.



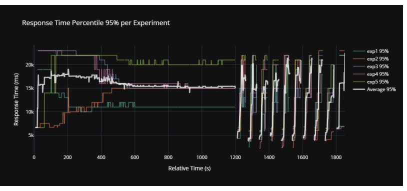
(c) Percentile 75%.



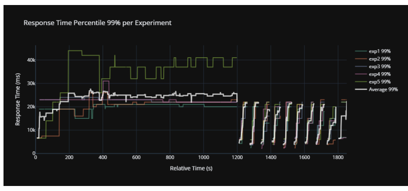
(d) Percentile 80%.



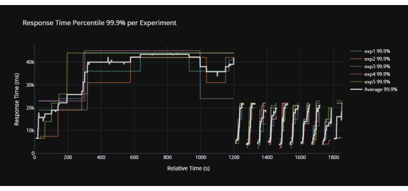
(e) Percentile 90%.



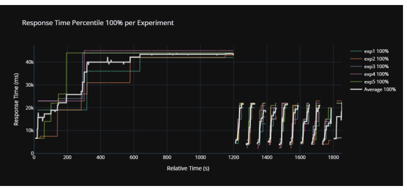
(f) Percentile 95%.



(g) Percentile 99%.



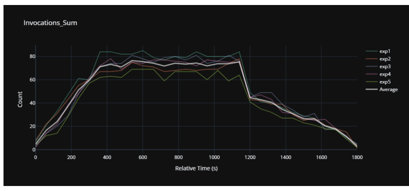
(h) Percentile 99.9%.



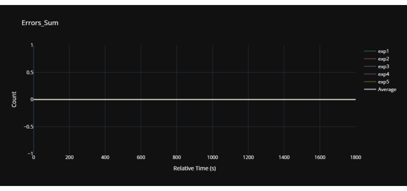
(i) Percentile 100%.

Figure 4: Client side - Response Time Percentiles.

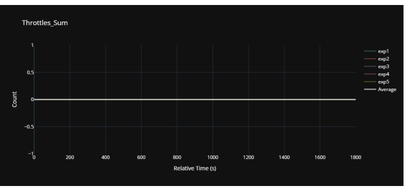
7.2 CloudWatch - Server side metrics



(a) Invocations.

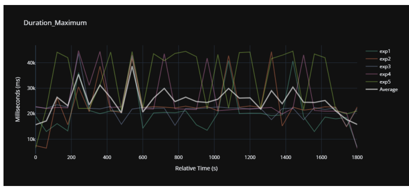


(b) Errors.

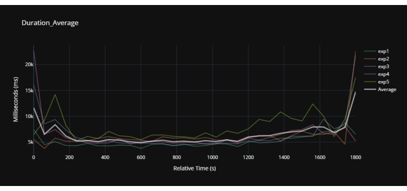


(c) Throttles.

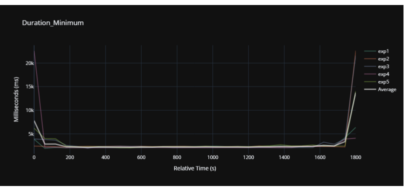
Figure 5: Server side - Count metrics.



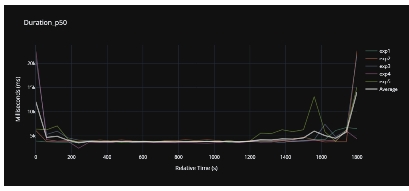
(a) Maximum.



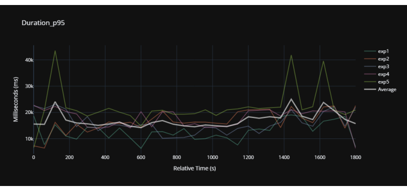
(b) Average.



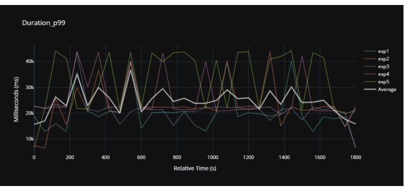
(c) Minimum.



(d) Percentile 50%.

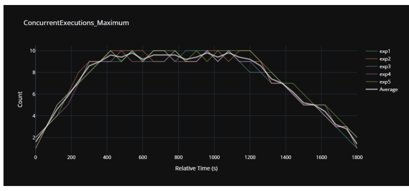


(e) Percentile 95%.

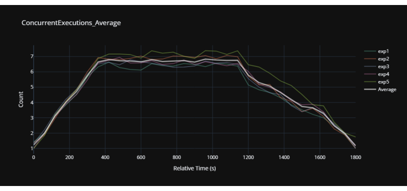


(f) Percentile 99%.

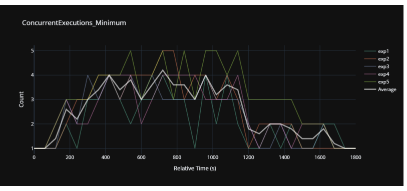
Figure 6: Server side - Duration.



(a) Maximum.



(b) Average.



(c) Minimum.

Figure 7: Server side - Concurrent Executions.

Note that other interesting detailed metrics such as cpu and memory utilization, network-in and network-out (or disk-read-ops and disk-write-ops) are not available since Lambda Insights is disabled for AWS Academy Learner Lab accounts. Nonetheless, we can thoroughly analyse the behaviour and performance of the system.

8 Analysis

From the results we clearly see the stress load test scenario taking place and the system scaling properly to the amount of requests received, indeed at different workload conditions Figure 3, the time needed to carry out the requests remains the same Figure 6. Error and throttles are not present throughout the experiments, further indicating the correct Lambda function working and Locust setup. This indicates the ability of our Lambda function to scale up according to the needed intensity.
Note that the evident spike up of duration times at the end of the experiment (approaching 30

minutes) may be caused by AWS deallocating resources from the lambda function and causing multiple cold starts for the running users. This happens for 3 out of 5 experiments.
From Table 1 it is evident how increasing image dimension leads to heavier CPU workload, and hence a diminished amount of requests possible due to the bigger response time.
Note that the average content size of the Lambda response is similar across the three different image sizes, this because the S3 URL is returned in the tests and not the raw image byte-encoding.

As we can see from the time duration metrics, cold start was indeed an issue in experiments, as it significantly slows down the execution time of the first request, leading to a 40-seconds increased response time, due to the model being loaded into memory.

This setup demonstrates good scalability; however, the computationally intensive nature of image generation inevitably causes performance slowdowns on a CPU-based infrastructure. In a real-world scenario with potentially thousands of concurrent user requests, a GPU-backed environment would be essential to handle the workload efficiently. GPUs are designed for parallel processing, making them far better suited for the large number of image operations that would overwhelm CPUs at large-scales.

The cost of running the five experiments with 10 users and five experiments with 5 users for a total of 5 hours of Lambda computing (30 minutes per test) is about 5.3$ of the AWS credit.

9 Future Work

- Consider running GPU-backed inference on EC2/GPU instances, moving heavier models off Lambda; keeping Lambda for lightweight or orchestrating tasks while using another computational back-end.
- Cache or memorize results for repeated prompts to avoid redundant computation.
- Test lambda function with larger and better image generation models, more suitable for a real production scenario. Thus generating higher resolutions images.
- Test with a higher computational workload with more users and requests per second.

10 Conclusion

AWS proved to be suitable for our target workload, effectively handling variable demand and scaling to meet incoming requests. Cold start latency was minimal, allowing subsequent requests to be processed efficiently. However, the current concurrency limit of 10 (imposed by AWS Academy Learner Lab environment policies) restricts testing under realistic production conditions. In addition, the CPU-bound nature of our implementation limits overall performance; a GPU-based setup would be more appropriate for large-scale image generation. Overall, the cost remained low, suggesting that with appropriate scaling and optimization, this approach could be both feasible and economically viable and profitable for real-world deployment. These findings demonstrate that AWS Lambda can support small-scale image generation workloads efficiently, achieving true scalability and good performance given the CPU-backed infrastructure.