Final Report: A Comparative Analysis of Cost-Performance for AI Model Serving on AWS

Rukai Xiao (124090729) Peixin Yao (122090663) Hezhe Wang (122040066)

December 14, 2025

Abstract

Putting AI models into the cloud means you have to balance three things: cost, speed, and how much headache you can tolerate managing servers. In this project, we tried to run a standard Deep Learning task (MobileNetV2) on three different AWS setups: Serverless (Lambda), normal x86 servers (EC2), and the newer ARM-based ones (Graviton). Honestly, the results weren't what we expected. We thought ARM or Serverless would save us the most money. But the data showed something else entirely. The old-school x86 EC2 instances were actually the cheapest at roughly \$0.69 per 1M inferences and the fastest, mostly because the software optimization is just better. On the other hand, Serverless had this massive cold start delay of around 13 seconds that we didn't see coming, making it way more expensive than we thought. We also found that while ARM chips are good on paper, the software support isn't quite there yet for heavy parallel processing.

1 Motivation: A Reality Check

At the start, we just wanted to benchmark GPUs. Our plan was pretty ambitious as we wanted to see how high-performance instances would crush standard CPUs. But reality hit us pretty hard when we tried to start the instances because the AWS Academy account has strict limits, and we simply weren't allowed to launch any GPUs.

We were pretty bummed out at first. But then we thought, do we really need a GPU for everything? For a small model like MobileNetV2, maybe a big GPU is actually a waste of money. So, we changed our plan. Instead of a brute-force speed contest, we decided to see how much performance we could squeeze out of regular CPUs. We wanted to find out if the Serverless hype is actually worth it compared to just running a well-tuned traditional server.

2 Technical Approach: Fighting the Infrastructure

2.1 Standardized Workload

We picked MobileNetV2 as our test subject. To keep the comparison fair and to keep us sane, we controlled the variables tightly. We used one standard JPEG image for every single request. We know this isn't exactly how the real world works, but it was the only way to make sure that any lag we saw was coming from the server, not from network jitter or different file sizes. Furthermore, we built a single Docker image with Python 3.9 and PyTorch 1.9 inside. We used this same image for both Lambda and EC2, so we knew the code running was exactly the same.

2.2 The Engineering Struggle

Getting this to run wasn't as easy as the tutorials make it look. We spent a couple of days dealing with two big headaches. The first hurdle was the strict deployment package size limit on AWS Lambda. Our PyTorch setup was over 700MB, which is way over the limit. We couldn't shrink it enough, so we had to cheat a little. We used AWS Elastic File System (EFS) to mount a network drive to the Lambda function. This let our code access the huge PyTorch libraries stored outside the function itself.

The second nightmare involved the read-only error. For a long time, our Lambda function kept crashing immediately. It turned out that PyTorch tries to create a cache folder in the home directory when it starts. But Lambda's file system is read-only. We finally fixed this by forcing the TORCH_HOME environment variable to point to the temporary folder, which is the only place we are allowed to write files.

3 Experimental Setup

We set up our testbeds in the us-east-1 region. We picked the smallest usable instances to simulate a startup trying to save money.

Table 1: Testbed Specifications

Setup	Instance Type	CPU Arch	Price (Hourly)
Traditional	`c5.large`	x86_64 (Intel)	$0.085
Budget Option	`t4g.medium`	arm64 (Graviton2)	$0.0336
Serverless	1024MB RAM	x86_64 (Lambda)	Pay-per-ms

For the testing tool, we wrote a Python script. We started with multiprocessing, but it kept freezing our own laptop. We switched to threading, which worked better to flood the servers with requests so we could measure the raw speed and latency.

4 Evaluation & "The Plot Twist"

4.1 Throughput: Old School Wins

We ramped up the concurrent users from 1 to 16 to see when the servers would break. As you can see in Figure 1, the EC2 x86 (c5.large) was the clear winner. It kept a steady latency around 110ms and handled 34 requests per second. It seems that PyTorch is just really well optimized for Intel's AVX-512 instructions, and it is hard to beat that optimization.

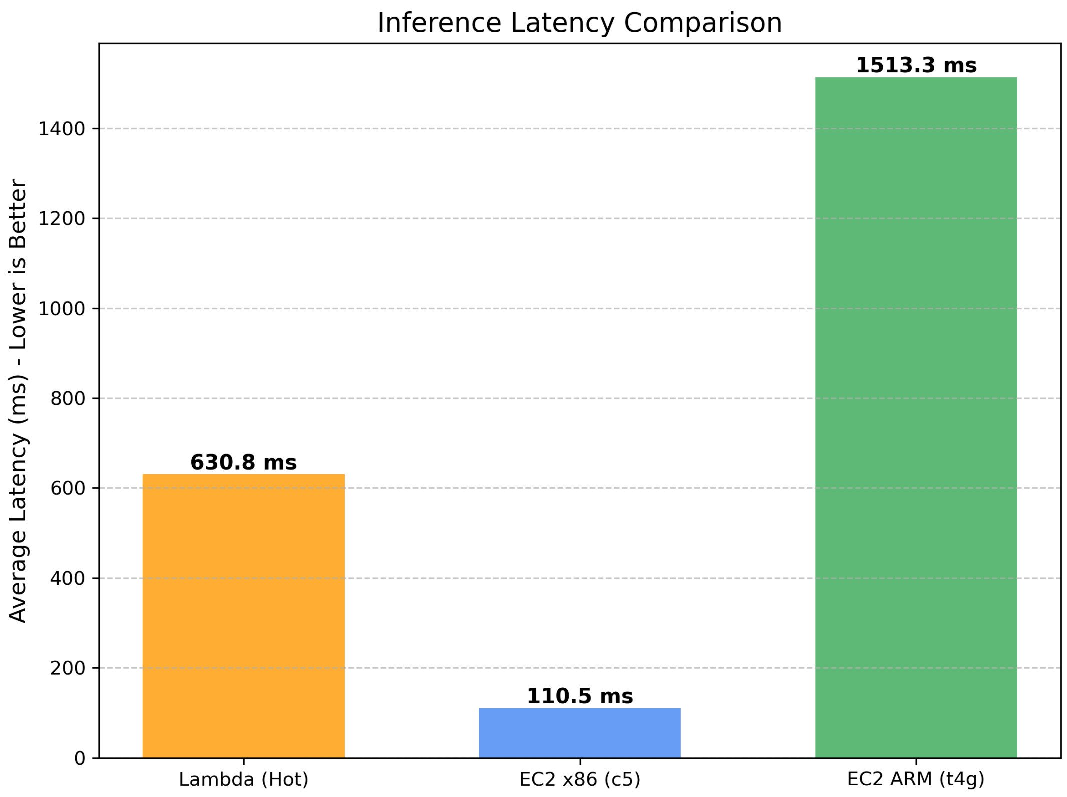

Figure 1: Latency Comparison. The x86 EC2 instance is way faster.

4.2 The ARM Letdown

We had high hopes for the t4g.medium instance. It is newer and cheaper. The weird thing is, its minimum latency was actually great at 92ms, which was faster than x86. But as soon as we added more concurrent requests, it fell apart. The latency spiked to over 1500ms. This looks like a software issue. The hardware is fast, but the PyTorch code for ARM probably isn't managing the threads as well as it does on x86. It taught us that specs are not everything.

4.3 The Cold Start Tax

We knew Serverless had cold starts, but we didn't expect it to be this bad. Our measurements showed a cold start latency of 13.27 seconds compared to a warm latency of just 0.63 seconds. This 21x difference is a huge problem. It basically means you have to pay for Provisioned Concurrency to keep it warm, which kind of defeats the purpose of using Serverless to save money.

4.4 Cost Verdict

We did the math to find the cost per 1 million inferences. The EC2 x86 instance emerged as the winner at roughly $0.69 because the high speed dilutes the cost per request. The EC2 ARM instance came in at $3.55 since it is cheap to rent but slow to finish the job. Lambda was the most expensive option at $10.70 because you pay for time, and CPU inference on Lambda is just too slow.

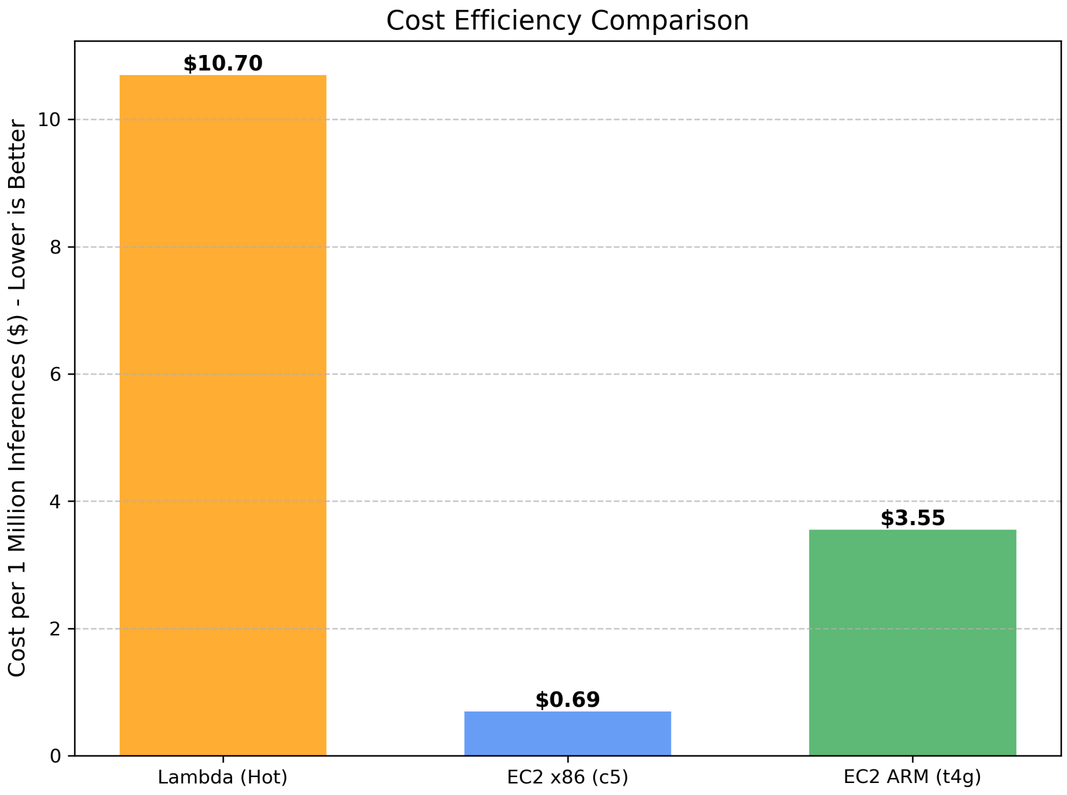

Figure 2: Cost Efficiency. x86 ended up being the cheapest.

5 Discussion: The Big Picture

This project showed us that just picking the cheapest instance like t4g can actually lose you money if your software doesn't run well on it.

Also, we listened to the guest lecture by Dr. Lingyun Yang from Alibaba [1], and it gave us a new perspective. Our struggle came from having to rent a whole c5.large just to get a little bit of CPU power. The future isn't about choosing instance types; it's about Resource Disaggregation where you can grab CPU and memory separately over the network. That would fix the waste we saw in our tests.

6 Conclusion

We started this wanting to test GPUs, but we ended up learning a lot about CPUs. Our conclusion is simple. For a model like MobileNetV2, stick to Traditional x86 EC2 instances. They give the best bang for your buck. Serverless is easy but too expensive for this kind of work, and ARM is promising but the software needs to catch up.

References

[1] Lingyun Yang. *High-Efficiency and Cost-Effective AI Model Serving Systems.* Guest Lecture at CUHK-SZ, Alibaba Group, 2025.

A Appendix: Artifacts

A.1 Code

We believe in open science. All our scripts, including the Dockerfiles and the messy testing logs, are uploaded here:

`https://github.com/[YOUR_GITHUB_ID]/CSC4160-Project`

A.2 Raw Data

Figure 3 sums up everything we found.

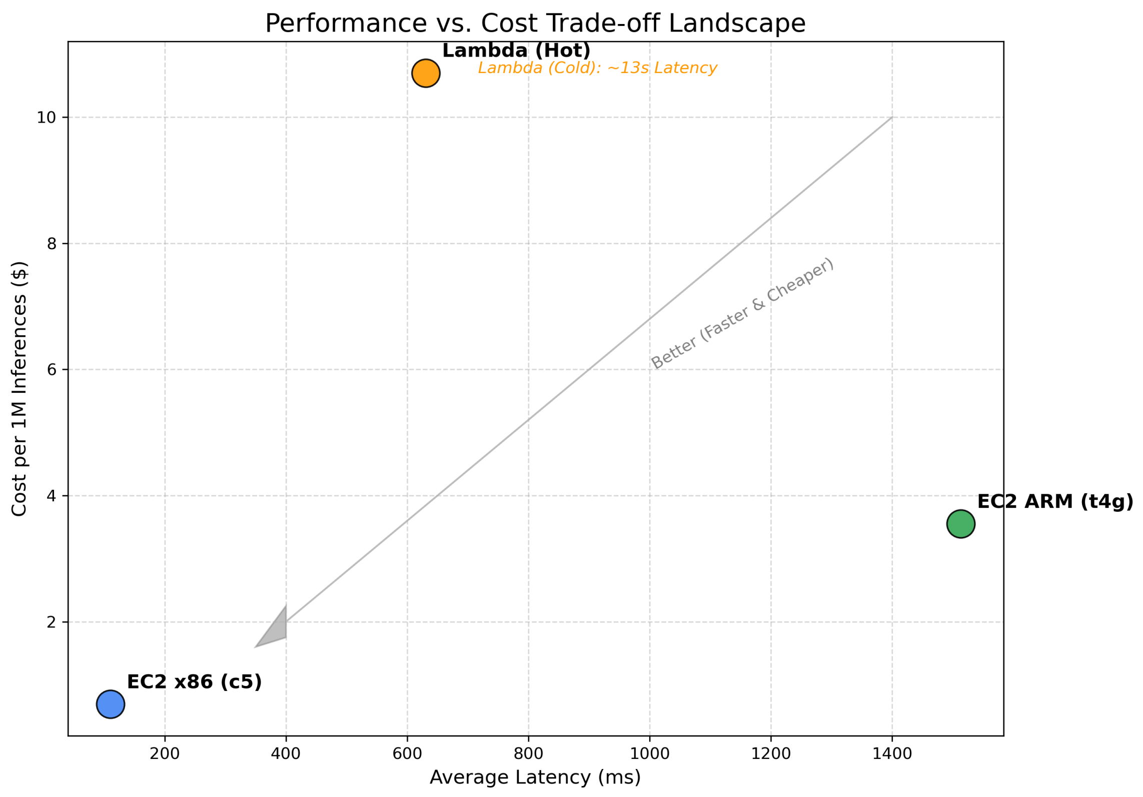

Figure 3: The Trade-off Map. Ideally, you want to be in the bottom-left. EC2 x86 is the only one that made it there.