

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

Deploying AI models in the cloud forces developers to balance cost, latency, and management overhead. This project investigates the reality of serving a Deep Learning workload (MobileNetV2) across Serverless (Lambda), x86 Compute (EC2), and ARM-based Compute (Graviton). To be honest, the results surprised us. We initially assumed that ARM or Serverless would be the clear winners in cost-efficiency. However, our data reveals a stark contrast where the traditional x86 EC2 architecture achieved the lowest cost of \$0.69 per 1 million inferences and the highest throughput thanks to AVX-512 optimization. Meanwhile, the Serverless approach suffered from a massive 21x cold start penalty that we did not anticipate. We also identified a frustrating software gap in ARM instances, where impressive hardware specs were bottlenecked by unoptimized software stacks.

1 Motivation: A Reality Check

We were initially dead set on benchmarking GPUs. Our original proposal was ambitious, aiming to see how high-performance instances would outperform standard CPUs. However, reality hit us hard when we tried to provision the resources; the AWS Academy environment has strict quotas that prohibited launching GPU instances.

We were disappointed at first, but then we questioned whether a GPU is truly necessary for everything. For a lightweight model like MobileNetV2, relying on heavy GPUs might actually be overkill. Consequently, we decided to pivot. Instead of a brute-force performance contest, we turned this into a hunt for the ultimate efficiency on commodity CPUs. We wanted to find out if the Serverless hype actually holds up against a well-tuned traditional server for CPU-bound tasks.

2 Technical Approach: Fighting the Infrastructure

2.1 Standardized Workload

We selected MobileNetV2 as our target. To keep things fair and save our sanity, we enforced strict variable control by using a single, standardized JPEG image for all requests. We know this simplifies the real world, but it was the only way to ensure that latency spikes were caused by the infrastructure rather than random network jitter or different JPEG decoding times. Furthermore, we forged a single Docker image containing Python 3.9 and PyTorch 1.9 to prevent any "it works on my machine" excuses; the runtime had to be identical on both Lambda and EC2.

2.2 The Engineering Struggle

Deploying this was not as smooth as the tutorials suggested. We spent days dealing with two main headaches. Our first major hurdle was the strict 250MB deployment package limit on AWS Lambda. Our PyTorch environment exceeded 700MB, so we had to architect a workaround by mounting an AWS Elastic File System (EFS) to the Lambda function. This allowed our function to access a massive external library, effectively bypassing the limit.

The second nightmare involved the read-only file system. For a long time, our Lambda function kept crashing with generic permission errors. It turned out that PyTorch stubbornly tries to write cache files to a home directory upon initialization. Since Lambda is read-only, this killed the process. We finally resolved this by overriding the environment variables to force PyTorch to use the writable temporary directory or our EFS mount.

3 Experimental Setup

We set up three testbeds in the us-east-1 region. We specifically chose the smallest available instances to mimic a cost-sensitive startup scenario.

Table 1: Testbed Specifications

Paradigm	Instance	Architecture	Hourly Cost
Traditional	c5.large	x86_64 (Intel Xeon)	\$0.085
Cost-Optimized	t4g.medium	arm64 (AWS Graviton2)	\$0.0336
Serverless	1024MB RAM	x86_64 (Lambda)	Duration Based

For testing, we wrote a custom Python script. Initially, we used multiprocessing, but it crashed our own client machine. We switched to threading to simply flood the endpoints with requests and measure the raw throughput and tail latency.

4 Evaluation & "The Plot Twist"

4.1 Throughput and Latency: Old School Wins

We pushed the concurrency from 1 to 16 to find the saturation point. As shown in Figure 1, the EC2 x86 (c5.large) was the clear winner. It maintained a stable 110ms latency and hit 34 inferences per second (IPS). It seems that PyTorch’s optimization for Intel’s AVX-512 instruction set is just too good to beat, even by newer architectures.

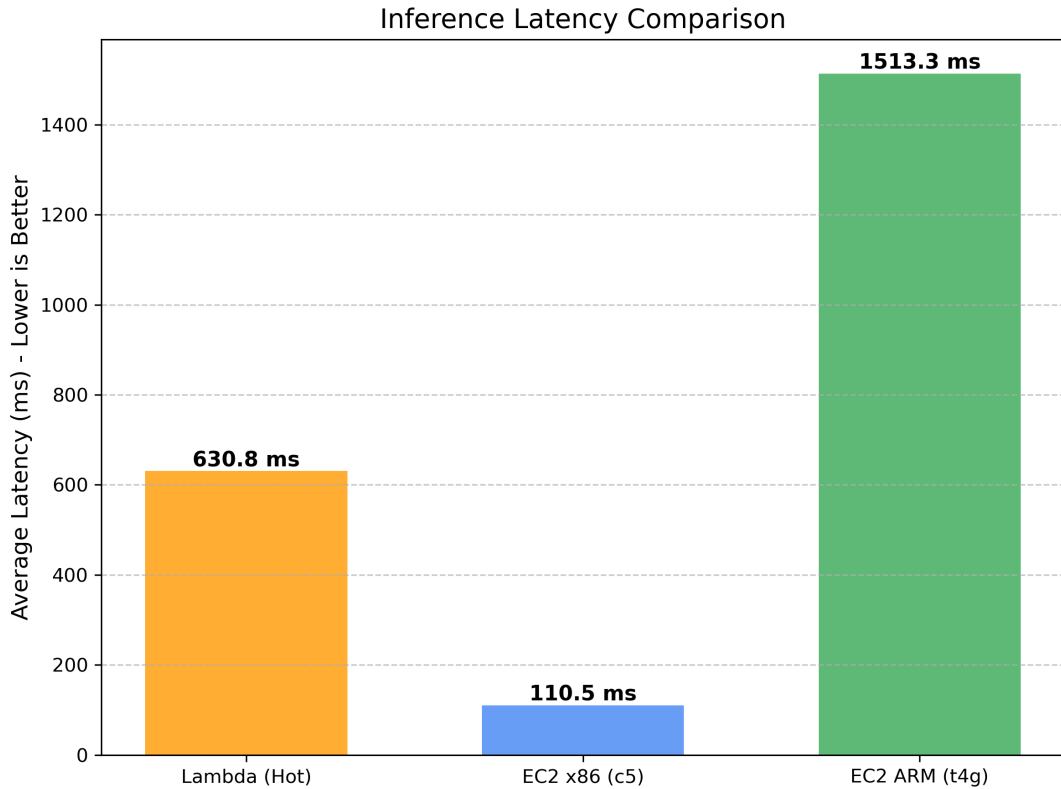


Figure 1: Average Latency Comparison. EC2 x86 wins by a large margin.

4.2 The ARM Disappointment

We had high hopes for the t4g.medium instance. It is cheaper, newer, and hyped as the future of cloud compute. However, while its minimum latency was an impressive 92ms, which was actually faster than x86, its performance completely fell apart under load, dropping to over 1500ms. This feels like a software gap. The hardware is ready, but the PyTorch backend for ARM just does not seem to handle concurrency as gracefully as x86. It was a lesson for us that hardware specs are not everything.

4.3 The Cold Start Tax

We knew Serverless had cold starts, but we did not 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 driven by the overhead of mounting EFS and loading PyTorch over the network. It is a deal-breaker for real-time apps and effectively means one has to pay for Provisioned Concurrency to keep it warm, which defeats the purpose of the pay-as-you-go model.

4.4 Cost-Efficiency Verdict

We normalized everything to the cost per 1 million inferences.

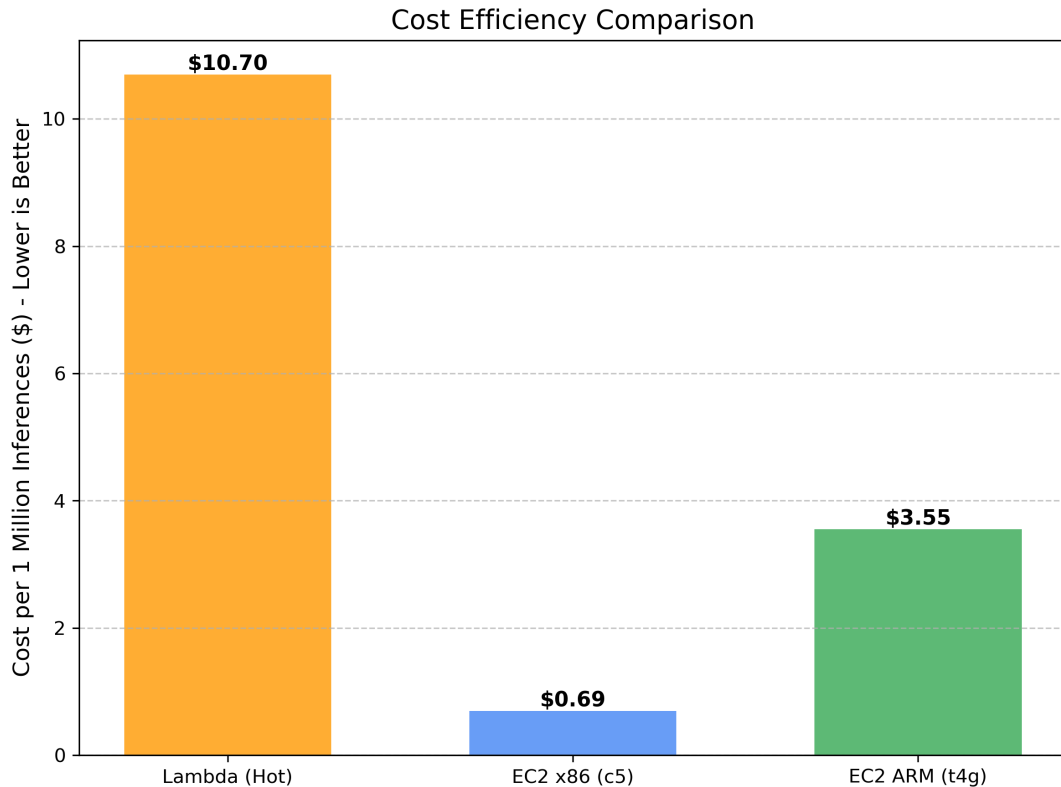


Figure 2: Cost Efficiency. x86 is surprisingly the cheapest option.

The EC2 x86 instance emerged as the winner at \$0.69 per million inferences because the high throughput dilutes the hourly cost. The EC2 ARM instance came in at \$3.55; it is cheap to rent, but too slow to run efficiently. Lambda was the most expensive option at \$10.70 because the billing model charges for duration, and CPU inference is simply too slow on Lambda without GPU acceleration.

5 Discussion: A Bigger Picture

This project taught us that blindly picking the cheapest instance like t4g can actually lose you money if the software isn't optimized for it.

Also, attending the guest lecture by Dr. Lingyun Yang from Alibaba [1] gave us a new perspective. Our struggle came from having to provision a whole c5.large just to get a bit of CPU power. The future is likely not about choosing instance types but about Resource Disaggregation, where we can grab CPU and memory independently over the network. That would solve the resource waste we saw in our experiments.

6 Conclusion

We started this project wanting to benchmark GPUs, but ended up learning a lot more about the nuances of CPU inference. Our conclusion is simple: for MobileNetV2, stick to traditional x86 EC2 instances. They offer the best price-performance ratio. Serverless is convenient but too expensive for this heavy workload, and ARM is promising but not quite there yet software-wise.

References

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

A Appendix: Reproducibility

A.1 Source Code

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

[https://github.com/\[YOUR_GITHUB_ID\]/CSC4160-Project](https://github.com/[YOUR_GITHUB_ID]/CSC4160-Project)

A.2 Raw Data Evidence

Figure 3 summarizes our entire journey in one chart.

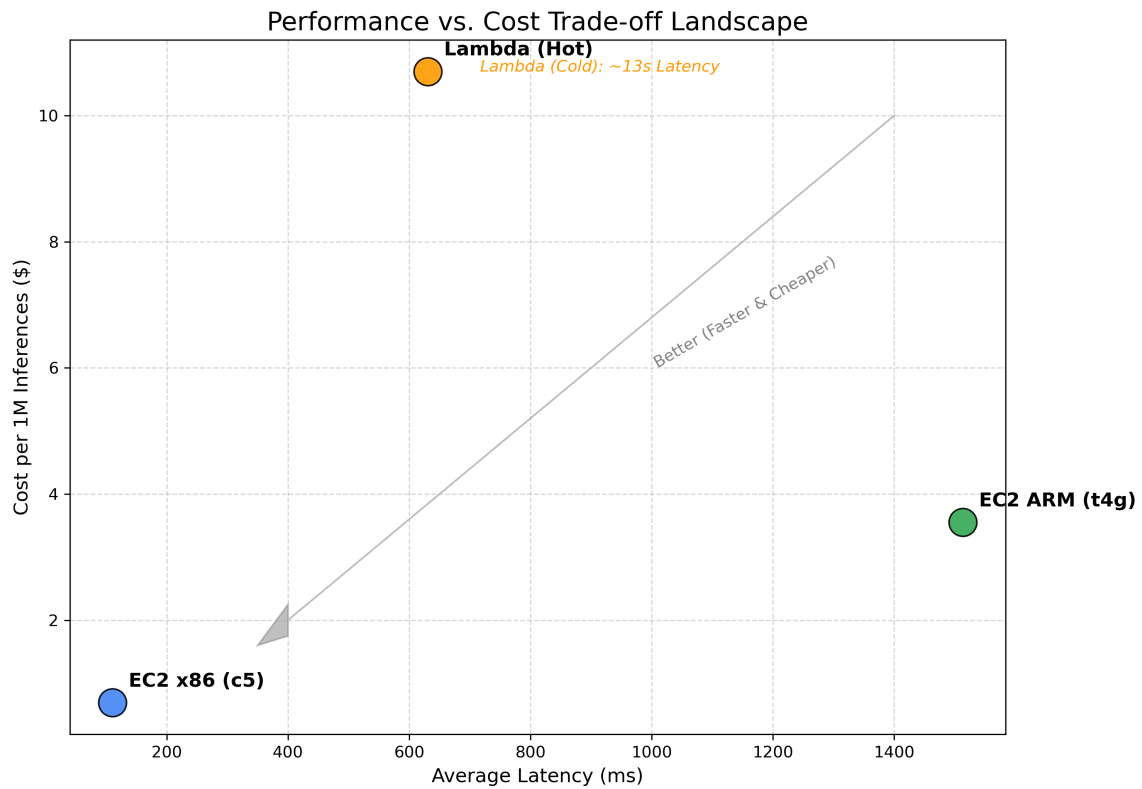


Figure 3: The Trade-off Map. Ideally, we want to be in the bottom-left corner. EC2 x86 is the only one that made it there.