Performance and Cost Comparison of Cloud Services for Deep Learning Workload

Dheeraj Chahal, Mayank Mishra, Surya Palepu, Rekha Singhal
TCS Research, Mumbai, India
{d.chahal, mishra.m, surya.palepu, rekha.singhal}@tcs.com

ABSTRACT

Many organizations are migrating their on-premise artificial intelligence workloads to the cloud due to availability of cost-effective and highly scalable infrastructure, software and platform services. To ease the process of migration, many cloud vendors provide services, frameworks and tools that can be used for deployment of applications on cloud infrastructure. Finding the most appropriate service and infrastructure for a given application that results in a desired performance at minimal cost, is a challenge.

In this work, we present a methodology to migrate a deep learning model based recommender system to ML platform and serverless architecture. Furthermore, we show our experimental evaluation of AWS ML platform called SageMaker and the serverless platform service known as Lambda. In our study, we also discuss performance and cost trade-off while using cloud infrastructure.

CCS CONCEPTS

• **General and reference → Performance**; • **Computer systems organization → Cloud computing**.

KEYWORDS

Recommendation system, ML Platform, AWS SageMaker, cloud performance

ACM Reference Format:
Dheeraj Chahal, Mayank Mishra, Surya Palepu, Rekha Singhal. 2021. Performance and Cost Comparison of Cloud Services for Deep Learning Workload. In *Companion of the 2021 ACM/SPEC International Conference on Performance Engineering (ICPE '21 Companion), April 19–23, 2021, Virtual Event, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3447545.3451184

1 INTRODUCTION

Many enterprises are migrating their artificial intelligence applications to cloud due to the several advantages such as availability of large infrastructure for high scalability, low operational management cost, etc. A user has to make many decisions judiciously while migrating the existing applications to cloud. For example, choosing appropriate cloud services, type of instances, configuration of chosen instances, etc. Such choices are based on multiple factors such as the application's characteristics, its performance requirements, the available budget, etc.

Many cloud vendors provide workflows to deploy machine learning (ML) model based applications with minimal efforts. For example, Amazon Web Services (AWS) SageMaker [1], Microsoft Azure ML [2], TFX [11], and MLFlow [16] are some of the ML platforms that allow automatic training, deployment and inference.

Serverless platform is also emerging as a preferred paradigm for deploying deep learning models. AWS Lambda [3], Microsoft Azure functions [4], Google cloud functions [5], and IBM OpenWhisk [6] are popular platforms for serverless computing. ML platfrom and serverless architecture may not be suitable for all kinds of workloads and requirements. For example, SageMaker instances provide low latency as compared to Lambda, but the pay-per-use model of serverless platform results in cost savings and it is suitable for bursty traffic.

Once a decision is made on cloud platform to deploy the application, the next step is to find the suitable family of instances for deployment. Each vendor provides a range of instances such as AWS provides compute intensive and memory intensive instances with varying configurations. These instances are chosen based on the application workload, expected performance, and the cost. For example, serverless platform is a better choice for serving event driven and bursty traffic at low cost whereas SageMaker endpoint can be used for long running workloads.

One important use case to study the appropriate cloud service and deployment environment for an application is migration of recommender system on cloud. A recommender system enhances customer experience by displaying most relevant items when a request is made. These recommendations are made based on the historical behavior of the customer. The machine learning models used by the recommender system can be deployed using ML platform and serverless architecture. Although, the use of ML platforms such as SageMaker for recommender systems is well known, but its comparison vis-a-vis serverless platform deployment is not explored by its practitioners.

In this work, we compare the performance and cost per inference incurred when a recommender system is deployed using ML platform and serverless architecture. Succintly, our contributions are as follows:

(1) We present a scheme for migrating a deep learning based recommender system to cloud

ICPE '21 Companion, April 19–23, 2021, Virtual Event, France
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8331-8/21/04…$15.00
https://doi.org/10.1145/3447545.3451184

[1] https://aws.amazon.com/sagemaker/
[2] https://azure.microsoft.com/en-in/services/machine-learning/
[3] https://aws.amazon.com/lambda/
[4] https://azure.microsoft.com/en-in/services/functions/
[5] https://cloud.google.com/functions
[6] https://www.ibm.com/in-en/cloud/functions

(2) We study the inference performance and cost using different types of instances available on cloud
(3) We compare the performance and cost when a recommender system is deployed using ML platform versus serverless platform.

We use well known ML platform called SageMaker [9] and serverless platform called Lambda provided by Amazon Web Services. SageMaker is an end-to-end machine learning platform used for data labeling, model training, deployment, and inference. SageMaker provides features such as GPU acceleration, auto-scaling, AB testing, batch inference but at a steep cost. AWS Lambda provides high scalability at a low cost, especially for functions with small execution time and low resource requirements.

Rest of the paper is structured as follows. Related work is discussed in section 2 followed by discussion on our recommender system, on-premise deployment and migration to cloud in section 3. Experimental setup and evaluation is discussed in section 4 and 5. Conclusion and future scope is discussed in section 6.

2 RELATED WORK

In order to enhance the customer experience, many organizations are using recommender systems based on studies of customer behavior. Lots of research has been done to develop machine learning workflows to deploy these systems, such as Michelangelo [1] from Uber and FBLearn [7] from Facebook with a capability of making 1 million and 6 million predictions per second respectively. Deployment of a recommender system called iPrescribe, using AWS SageMaker, has been studied earlier [4].

Serverless platform is becoming increasingly popular for deployment of deep learning models due to its high scalability and pay-per-use cost model. SPEC RG Cloud working group published 86 serverless use cases in its technical report [15]. The serverless architecture has been studied extensively to deploy deep learning model based applications [5] [8]. Their suitability for recommender systems has been also been explored in [3, 13].

Zheng *et al.* conducted a study on cloud services for cost-effective and SLO aware machine learning inference [17]. We extend this work further and compare the performance of our recommender system on different types of SageMaker endpoint servers. We also compare the performance of SageMaker with the serverless platform called Lambda.

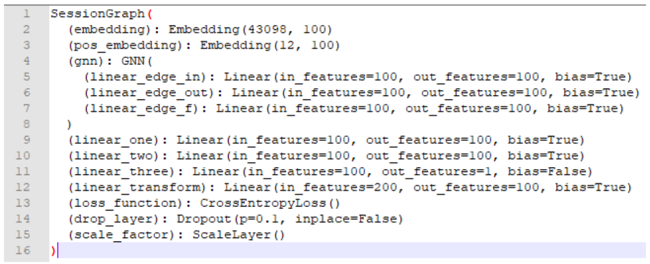
To the best of our knowledge, this work is a first effort to compare ML platform and serverless architecture for deploying deep learning model based recommender system.

3 OUR RECOMMENDER SYSTEM

In this paper, we use our recently developed session based recommendation model known as NISER [6]. NISER is a graph neural network (GNN) [14] based model which utilizes a user's past actions like product or item clicks (known as a session) to recommend the next product or item.

3.1 On-premise recommender system

NISER model consists of several layers of neurons (including the GNN layer) as shown in Figure 1.



```
SessionGraph(
  (embedding): Embedding(43098, 100)
  (pos_embedding): Embedding(12, 100)
  (gnn): GNN(
    (linear_edge_in): Linear(in_features=100, out_features=100, bias=True)
    (linear_edge_out): Linear(in_features=100, out_features=100, bias=True)
    (linear_edge_f): Linear(in_features=100, out_features=100, bias=True)
  )
  (linear_one): Linear(in_features=100, out_features=100, bias=True)
  (linear_two): Linear(in_features=100, out_features=100, bias=True)
  (linear_three): Linear(in_features=100, out_features=1, bias=False)
  (linear_transform): Linear(in_features=200, out_features=100, bias=True)
  (loss_function): CrossEntropyLoss()
  (drop_layer): Dropout(p=0.1, inplace=False)
  (scale_factor): ScaleLayer()
)
```

Figure 1: NISER model layers and arrangement.

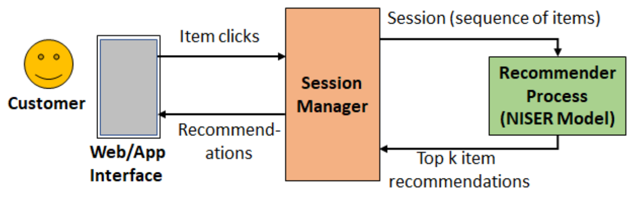

Customer
Web/App Interface
Item clicks
Recommend-ations
Session Manager
Session (sequence of items)
Recommender Process (NISER Model)
Top k item recommendations

Figure 2: Overall scenario and placement of different components in session based recommendations involving NISER.

Session based recommendation models like NISER require only the recent actions of users for recommendations, and hence are useful in scenarios where the users, for whom the recommendations are being made, are not known. For example, first time visitors on an e-retail website who are browsing products (or a non logged-in user).

NISER represents an item as embedding (a way of representation in form of a vector of length 100). The model training involves learning these item embeddings in such a manner that items which are often clicked one after another (or in close vicinity), have embeddings which are similar to each other. The data set employed for training consists of past sessions of item clicks by users seen by the system. During prediction, the sequence of items present in current session are used to predict the top k items to be recommended next (Figure 2).

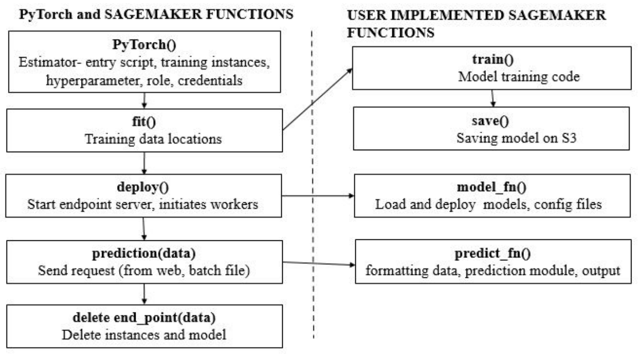

PyTorch and SAGEMAKER FUNCTIONS
USER IMPLEMENTED SAGEMAKER FUNCTIONS
PyTorch()
Estimator- entry script, training instances, hyperparameter, role, credentials
fit()
Training data locations
deploy()
Start endpoint server, initiates workers
prediction(data)
Send request (from web, batch file)
delete end_point(data)
Delete instances and model
train()
Model training code
save()
Saving model on S3
model_fn()
Load and deploy models, config files
predict_fn()
formatting data, prediction module, output

Figure 3: Recommender system deployment using AWS SageMaker APIs

Figure 2 shows the scenario where a user clicks the items (session) over a web or app based interface. There is a session manager to identify individual users' sessions as well as to keep track of item clicks per session. The NISER model is embedded in the recommender process which recommends the top k items for a particular session.

3.2 Cloud based implementaion of recommender system

In this section, we discuss migration of our recommender system to cloud. First we discuss the deployment scheme which we used to migrate it to AWS SageMaker followed by a discussion on AWS Lambda migration.

3.2.1 Using AWS SageMaker. We use AWS SageMaker APIs to deploy our recommender system on cloud as shown in Figure 3. A call to *fit* API invokes the *train* method in the entry script defined in the PyTorch [12] estimator. We pre-load the training data on S3 which is invoked inside the train function for training the model. At the completion of the the training, model is returned by *train*. Trained model is saved in S3 along with the parameter values.

Once the traing process is over, save API is invoked and model is saved on S3. A call to deploy function invokes *model_fn* method of SageMaker and model is accessed from S3. A SageMaker PyTorch model server is created to host the model. The model server runs inside the SageMaker endpoint. The end-point created to serve the model is accessed using the predictor object returned by the deploy API. The predictor object has predict method to do inference on the endpoint hosting our model. Predict returns the result of the inference which is NumPy array by default.

3.2.2 Using AWS Lambda. In another deployment strategy, we use AWS serverless platform Lambda to deploy the recommender system (Figure 3). Serverless platform for recommender system seems a good choice due to a small execution time per inference.

One of the challenges in serverless platform is the statelessness of Lambda functions. Lambda provides a total of 500 MB storage and only 250 MB is available for deployment packages. The PyTorch packages and NumPy required for recommender system is approximately 900 MB in size. We use CPU version of PyTorch requiring only 200 MB and hence satisfies the storage constraint of the Lambda. Another challenge in serverless platform is a cold start which results in very high latency for the first request. There are techniques proposed by researchers to mitigate the problem in serverless platform [2] [10]. As discussed in section 3.2.1, we used SageMaker APIs for training, deploying and to infer the model. However, in this study we used Lambda for deployment and inference only. As and when the first request arrives, an event is triggered and NISER model is loaded in the memory of Lambda function from S3 along with other packages and code from the local storage. This results in a cold start and with a significant latency overhead. The event handler code in the Lambda function contains the predict function which gets executed for each request.

4 EXPERIMENTAL SETUP

We conducted on-premise experiments on an Intel server consisting of 28 physical cores and 256 GB memory. We used Python 3.6 with PyTorch version 17.1 over CentOS 7. For deploying and inference using SageMaker, we used ml.c5.large and ml.c5.xlarge compute

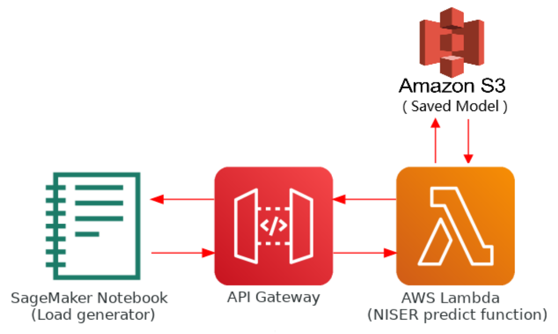


Figure 4: Recommender system deployment on AWS Lambda

family CPU instances with gdn.ml.xlarge and gdn.ml.2xlarge family of GPU instances. Also, for serverless deployment, each instance used was configured with 3GB memory, hence resulting in allocation of 2 cores for each Lambda function. Load was generated using Jupyter notebook instance for SageMaker as as well Lambda. For the performance measurements, response time was measured with an increasing concurrency (number of concurrent users). The response time shown in the experiments is round trip time for the requests. For all the serverless experiments, we used provisioned concurrency, and response time is collected while Lambda is warm. All data points are collected in the steady state of the system.

5 EXPERIMENTAL EVALUATION

In this section, we first discuss the experiments that we performed on-premise and then compare the performance with AWS SageMaker and AWS Lambda.

5.1 On-premise

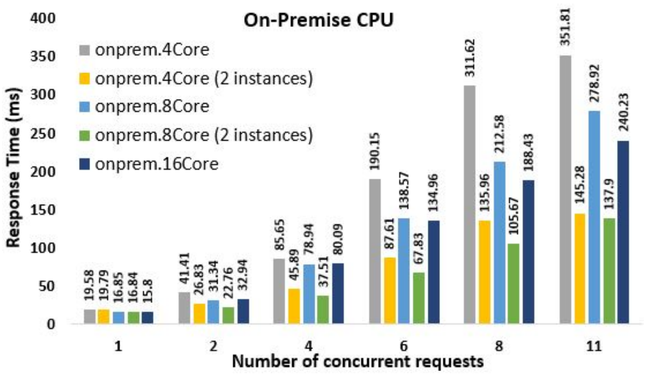


Figure 5: Response time of recommender system with on-premise server

We performed experiments by allocating different number of cores (4, 8, 16) to the NISER recommendation process instance. We also experimented with 2 instances of NISER recommendation process. We used Python 3.6 with PyTorch version 17.1 over CentOS 7. The performance graph displaying the mean response times in figure 5 shows that as we increase the number of concurrent requests, we observe an increase in the response time of the requests in each of the instance types irrespective of the number of cores allotted to the recommendation process. For example, consider the "light blue" bar representing 8 core allocation, the response time increases from 16ms for a single request at a time to 278ms when 11 concurrent requests are made. This increase is primarily due to the queuing delay.

An interesting observation is that, at higher concurrency situations (6,8,11), the response time of a one 8 core instance is approximately twice that of two 4 cores instances. The same observation is true for response times observed for a single 16 core instance versus two 8 core instances.

5.2 Cloud deployment using AWS SageMaker

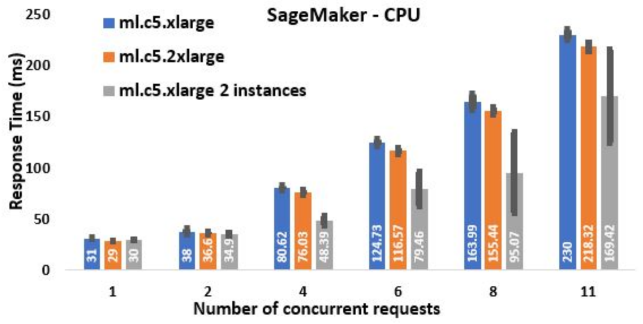


Figure 6: Response time of recommender system on two different types of CPU based SageMaker instances

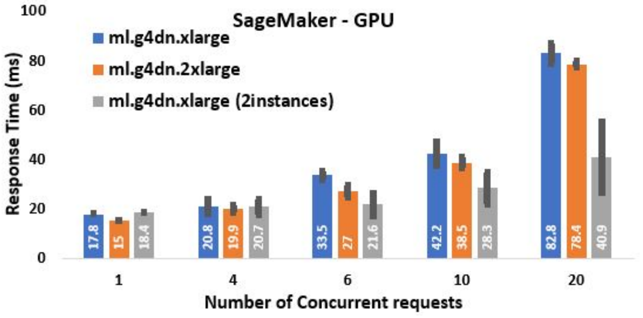


Figure 7: Response time of recommender system on two different types of GPU based SageMaker instances

In this experiment, we evaluate the effect of scaling on the response time and cost per inference. We run our experiments on *ml.c5.xlarge* (4 cores) and *ml.c5.2xlarge* (8 cores) and 2 instances of *ml.c5.xlarge* (2*4 cores) family instances. As shown in figure 6, as we increase the number of concurrent requests, we observe an increase in the response time of the requests in each of the instance types. As expected, we observe that response time of ml.c5.2xlarge instance is slightly lower than ml.c5.xlarge instance at higher concurrencies due to availability of more cores. Similar behavior is observed in the experiments using SageMaker GPUs as shown in figure 7.

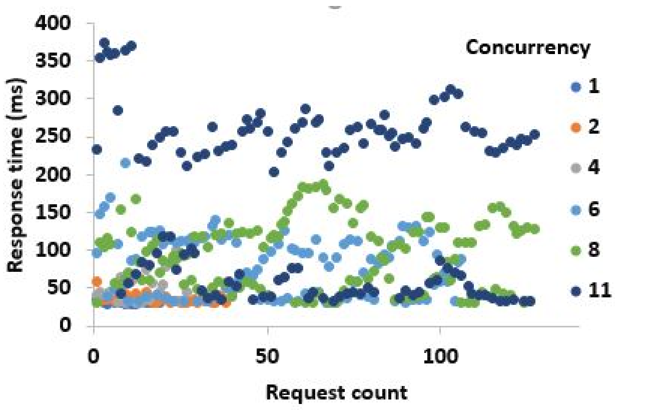


Figure 8: Response time with increasing workload when using multiple CPU instances with AWS SageMaker

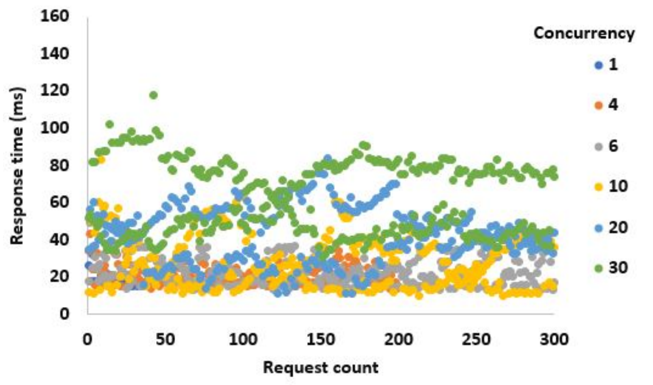


Figure 9: Response time with increasing workload when using multiple GPU instances with AWS SageMaker

Another interesting observation is that, average response time observed using 2 instances of *ml.c5.xlarge* with total 8 cores is significantly less than on one instance of *ml.c5.2xlarge* having same number of cores. However, we see a large variation in the response time of the requests particularly at higher concurrency when using multiple instances instead of one of equivalent configuration as shown in figures 8 and 9. This is due to the load imbalance on the SageMaker model servers while distributing the requests to multiple instances.

Figure 10 shows the cost per inference comparison between *ml.c5.xlarge* (4 core) and *ml.c5.2xlarge* (8cores). While the performance gain as observed in figure 6 is small, we notice that cost per inference is approximately double in *ml.c5.2xlarge* as compared to the *ml.c5.xlarge.* Similar behavior is observed in GPUs (Figure 11).

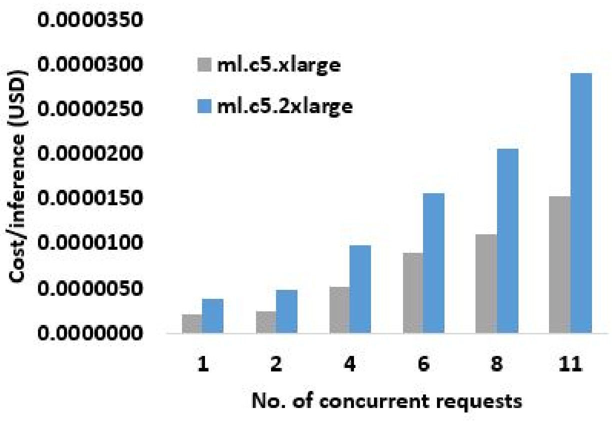


Figure 10: Cost per inference comparison of ml.c5.xlarge and ml.c5.2xlarge type CPU based SageMaker instances

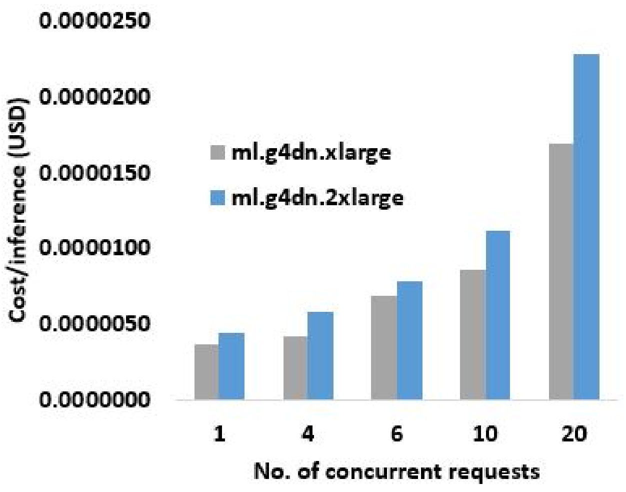


Figure 11: Cost per inference comparison of ml.g4dn.xlarge and ml.g4dn.2xlarge type GPU based SageMaker instances

However, the cost per inference difference between *ml.g4dn.xlarge* and *ml.g4dn.2xlage* is not as wide as observed in CPU instances.

These experiments suggest that there is a scope for trade-off in performance and cost of the deployment. An increase in the resources at higher cost does not translate to proportional gain in the performance of the application. The user has to choose deployment infrastructure judiciously that satisfies SLA and budget constraints.

5.3 Comparison of on-premise, AWS SageMaker and Lambda

Figure 12, shows the comparison of response time observed op-premise CPU, SageMaker CPU, GPU endpoint server, and deployment of the model on AWS Lambda. At very low concurrency, we observe that response time from Lambda is higher than SageMaker CPU and GPU. This is due to the fact that the compute resources of

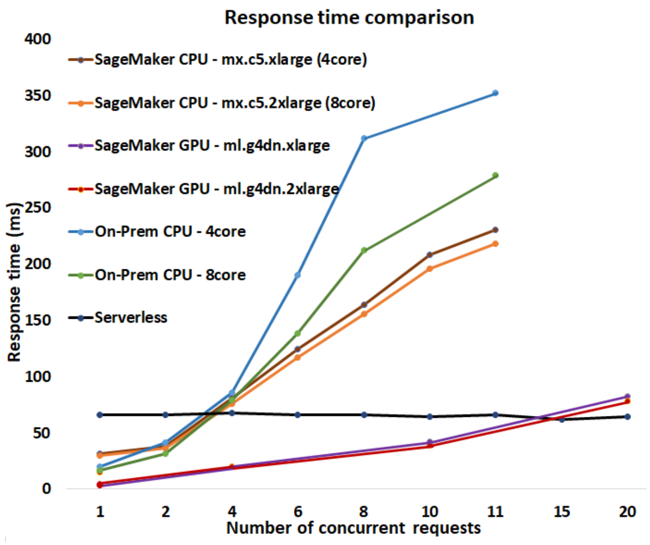


Figure 12: Response time comparison of on-premise, AWS SageMaker CPU, GPU endpoint and AWS Lambda with increasing workload

CPU and GPU instances in SageMaker are underutilized. As the concurrency or load increases, resources are saturated in SageMaker instances. However, Lambda provides high scalability and spawns instances proportional to the number of requests. This results in a constant response time in Lambda but it increases linearly with increasing concurrency in SageMaker CPU and GPU endpoints. As shown, the response time observed with CPU endpoint is highest as compared to the GPU endpoint and Lambda at higher concurrency values. Although activating scaling feature in SageMaker can mitigate the increasing response time problem, but that results in higher implementation cost as compared to the pay-per-use model in Lambda. We observe that response time for on-premise deployment is higher as compared to SageMaker CPU deployment with comparable configuration. This is due to optimized libraries and environment used in SageMaker.

Figure 13 shows cost comparison of AWS SageMaker and Lambda. SageMaker and Lambda have unique cost models and hence are challenging to compare. The cost of SageMaker instances is proportional to the up-time of the instances and is independent of CPU usage. Whereas, Lambda follows pay-per-use model that derives cost based on the execution time, memory used, and the number of requests served.

Figure 13 shows the maximum achievable request rate for each instance and its fixed cost for one hour. However, Lambda cost as shown in the figure, is dependent on requests served and the time for serving those requests. The intersection point of Lambda trend-line with each of the instances represents the break-even point. If the request rate extracted from the instance is below the intersection point, serverless is a cheaper option. This is due to the fact that SageMaker instances are underutilized in the stipulated one hour, but billed for the whole duration. However, If a request rate extracted from an instance is between the intersection point and

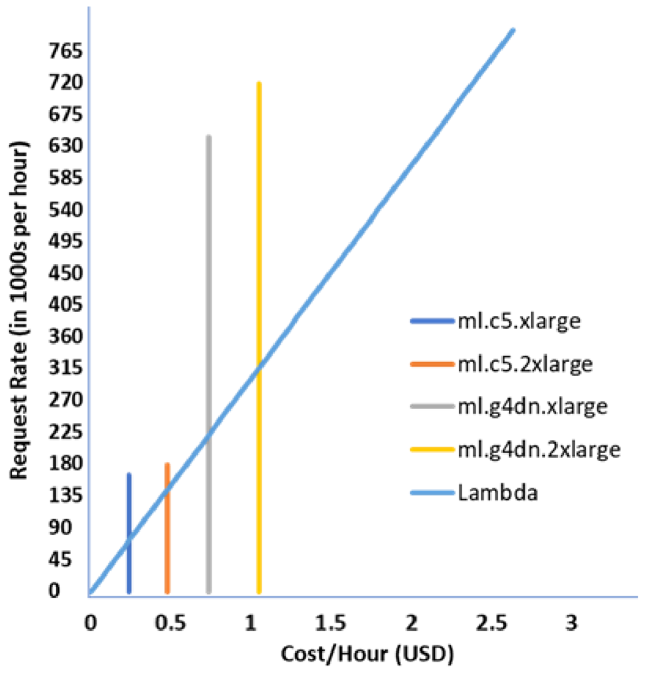


Figure 13: Cost comparison of AWS SageMaker instances (CPU and GPU) and AWS Lambda

maximum achievable, then SageMaker is cheaper because instances are optimally utilized.

Based on our experimental data, we can conclude that there are some scenarios where serverless architecture delivers better performance due to fast and high scalability for bursty workloads as compared to the ML platform. On the other hand, ML Platform is a better option for long running workloads when resources are optimally used.

However, there are some inherent constraints and challenges in ML Platform and serverless platform. For example, the cold start in serverless results in very high latency for the first request. We observed a cold start time of 30 seconds. Use of ML platform in conjunction with serverless architecture can mitigate the problem of cold start. Workload can be balanced between ML platform and serverless architecture such that the bursty traffic can be redirected to the serverless without provisioning for short lived workload in the ML platform. Although auto-scaling is possible in most of the ML platforms including SageMaker, but scaling up and down takes a few minutes as compared to few seconds in scaling up and down serverless instances. Hence, serverless can be a good choice for bursty workloads.

6 CONCLUSION AND FUTURE WORK

We presented a methodology to migrate a recommender system to AWS cloud framework called SageMaker and serverless platform Lambda. In this work, we presented the performance and cost comparison with different types of instances used as SageMaker endpoints. Additionally, we presented a comparison of AWS SageMaker and serverless platform Lambda performance. Our experimental evaluation shows that cost incurred due to scaling of platform instances does not increase proportionally. Also, autoscaling in serverless architectures results in better performance as compared to the dedicated endpoints.

Our future work is focused on using the application profiling and characterization data to find the most appropriate cloud service. The application characteristics such as memory usage, compute requirement, processing time, portability, workload, environment requirement, etc. can be deciding factors in mapping the application to the optimal infrastructure, platform, or service. We are also building performance models using on-premise characterization data. These models can be used to configure the cloud platform resources for cost-effective and high performance deployment.

REFERENCES

[1] 2019. Uber Michelangelo. https://eng.uber.com/michelangelo/. [Online;Accessed 20 January 2019].
[2] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. 2020. Using Application Knowledge to Reduce Cold Starts in FaaS Services. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic) *(SAC '20)*. Association for Computing Machinery, New York, NY, USA, 134–143. https://doi.org/10.1145/3341105.3373909
[3] A. Bhattacharjee, A. D. Chhokra, Z. Kang, H. Sun, A. Gokhale, and G. Karsai. 2019. BARISTA: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*. 23–33. https://doi.org/10.1109/IC2E.2019.00-10
[4] Dheeraj Chahal, Ravi Ojha, Sharod Roy Choudhury, and Manoj Nambiar. 2020. Migrating a Recommendation System to Cloud Using ML Workflow. In *Companion of the ACM/SPEC International Conference on Performance Engineering* (Edmonton AB, Canada) *(ICPE '20)*. Association for Computing Machinery, New York, NY, USA, 1–4. https://doi.org/10.1145/3375555.3384423
[5] D. Chahal, R. Ojha, M. Ramesh, and R. Singhal. 2020. Migrating Large Deep Learning Models to Serverless Architecture. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 111–116. https://doi.org/10.1109/ISSREW51248.2020.00047
[6] Priyanka Gupta, Diksha Garg, Pankaj Malhotra, Lovekesh Vig, and Gautam Shroff. 2019. NISER: Normalized Item and Session Representations to Handle Popularity Bias. arXiv:1909.04276 [cs.IR]
[7] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 620–629.
[8] V. Ishakian, V. Muthusamy, and A. Slominski. 2018. Serving Deep Learning Models in a Serverless Platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 257–262. https://doi.org/10.1109/IC2E.2018.00052
[9] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, Can Balioglu, Saswata Chakravarty, Madhav Jha, Philip Gautier, David Arpin, Tim Januschowski, Valentin Flunkert, Yuyang Wang, Jan Gasthaus, Lorenzo Stella, Syama Rangapuram, David Salinas, Sebastian Schelter, and Alex Smola. 2020. Elastic Machine Learning Algorithms in Amazon SageMaker. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 731–737. https://doi.org/10.1145/3318464.3386126
[10] Ping-Min Lin and Alex Glikson. 2019. Mitigating cold starts in serverless platforms: A pool-based approach. *arXiv preprint arXiv:1903.12221* (2019).
[11] Akshay Naresh Modi, Chiu Yuen Koo, Chuan Yu Foo, Clemens Mewald, Denis M. Baylor, Eric Breck, Heng-Tze Cheng, Jarek Wilkiewicz, Levent Koc, Lukasz Lew, Martin A. Zinkevich, Martin Wicke, Mustafa Ispir, Neoklis Polyzotis, Noah Fiedel, Salem Elie Haykal, Steven Whang, Sudip Roy, Sukriti Ramesh, Vihan Jain, Xin Zhang, and Zakaria Haque. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *KDD 2017*.
[12] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-

an-imperative-style-high-performance-deep-learning-library.pdf

[13] Vikram Sreekanti, Harikaran Subbaraj, Chenggang Wu, Joseph E Gonzalez, and Joseph M Hellerstein. 2020. Optimizing Prediction Serving on Low-Latency Serverless Dataflow. *arXiv preprint arXiv:2007.05832* (2020).

[14] Matteo Tiezzi, Giuseppe Marra, Stefano Melacci, Marco Maggini, and Marco Gori. 2020. A Lagrangian Approach to Information Propagation in Graph Neural Networks. *arXiv preprint arXiv:2002.07684* (2020).

[15] Erwin van Eyk, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, and Simon Eismann. 2018. A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18)*. Association for Computing Machinery, New York, NY, USA, 21–24. https://doi.org/10.1145/3185768.3186308

[16] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018), 39–45.

[17] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*.