Cloud Services Comparison

Oishi Poddar
MS CS, UC San Diego
opoddar@ucsd.edu

Nidhi Dhamnani
MS CS, UC San Diego
ndhamnani@ucsd.edu

1 GITHUB LINK

Code: https://github.com/oishi-poddar/291-virtualisation

2 INTRODUCTION

AWS provides a plethora of services for each category including compute. It is hard to choose the right service and options within a service. For instance, for compute someone might ask whether is it better for them to run it on virtual machines, containerize, or go fully serverless. AWS provides a wide breadth of services and features for customers to run virtually any kind of workload.

The goal of this project is to compare different AWS cloud services for an application. The motivation for this project is to solve the confusion which people face when trying to choose an AWS service for their application. AWS provides a host of various services ranging for use cases like compute, storage, database, networking, analytics, machine learning, and security. Our objective is to provide a detailed description and analysis of the selected services of a category and evaluate them for an application based on different metrics. We aim to understand the various tradeoffs that each service we are analyzing provides to understand the pros and cons of each service and option depending on the use case.

3 HIGH-LEVEL DESIGN

3.1 Application

We chose handwritten digit classification using machine learning as our application which predicts the digit a handwritten image represents. The dataset we use is the inbuilt Python library sklearn MNIST dataset which contains 70,000 images. We train our ML model using Support Vector Machines and perform prediction/inference on new images using the pre-trained model. We used Sagemaker to train and deploy our model. We created a web application that exposes the APIs to perform inference on a new image. We deployed the web app on different compute services (EC2, Auto-scaling groups, and ECS) and compare the performance across different metrics.

3.2 Architecture

The architecture of our project is depicted in Fig. 1 and Fig. 2. We train our model on Sagemaker notebooks and create a pickle of the trained model and store it on S3. We then create a Sagemaker Model Endpoint that uses the pre-trained model from S3 and Docker image stored in ECR and use it for inference. We also spin up Docker container that runs a web server providing APIs to perform inference using the trained model.

We deploy the inference code on AWS compute services (details on the implementation of deployment found in Section 4). As shown in Fig. 2, we send requests as the client application to our application API's hosted on web servers on EC2 and ECS, which performs inference on the data and provides a response back. The inference application for each request used the deployed model endpoint on Sagemaker to get a prediction response and make an inference.

4 IMPLEMENTATION

4.1 Sagemaker

Amazon SageMaker is a fully managed machine learning service.

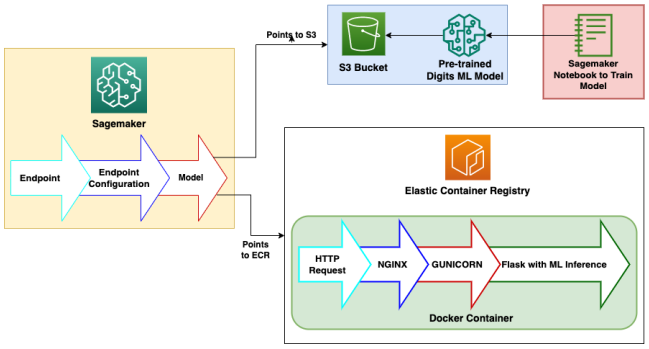

Figure 1. Architecture Diagram of ML Training and Deployment on Sagemaker

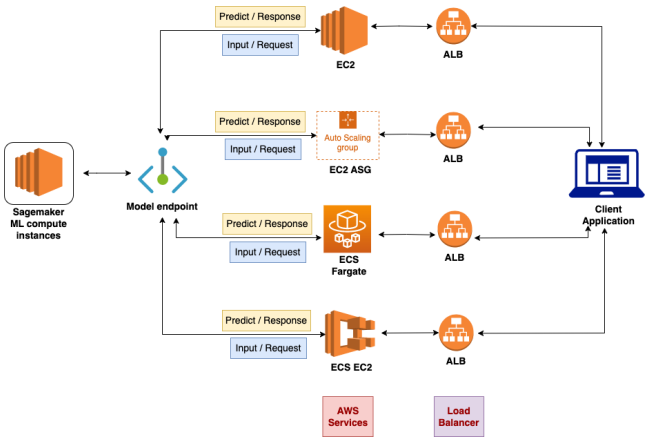

Figure 2. Architecture Diagram for Inference

With SageMaker, we can quickly and easily build and train machine learning models, and then directly deploy them into a production-ready hosted environment. It provides an integrated Jupyter authoring notebook instance for easy access to our data sources for exploration and analysis, so we don't have to manage servers.

4.1.1 Implementation Details: SageMaker offers a functionality known as Bring Your Own Container (BYOC) that provided us full control as a developer. The steps we performed to set up train and deploy our model on Sagemaker are as follows:

- Used Sagemaker Notebooks to train the digits classifier model on the sklearn's inbuilt MNIST dataset.
- Stored the trained model pickle file in S3 bucket.
- Used docker for running the inference code. Created a docker container containing configurations for NGINX and Gunicorn. Along, with it implemented the flask web server containing two APIs - 1) Health check API 2) Inference API.
- Built and pushed the docker image to ECR (Elastic Container Registry).
- Created Sagemaker model that uses the docker image from ECR and trained model from S3.
- Created an endpoint configuration and endpoint for the model that can be used to invoke the inference API.

Fig. 1 depicts the architecture diagram and a summary of the above steps.

4.2 EC2

Amazon EC2 is a web-based service that provides scalable computing power on Amazon Web Services.

4.2.1 Implementation Details: We spun up 3 EC2 instances: t2.small, t2.medium, and t2.xlarge (results for those shown in Section 5). For our application which only performs inference, T2 is suitable as it has less than 2 cores per instance and is thus a lower-cost option. T2 is designed for general-purpose workloads such as web servers, development environments, and small databases and is suitable for our use.

As shown in Fig. 3, we have spun up our Python application on a web server using Gunicorn, NGINX, and Flask. An overview of the steps are:

- Created an Ubuntu EC2 on AWS and deployed the Flask App via SSH.
- Ran Gunicorn WSGI server to serve the Flask Application.
- Used *systemd* to manage Gunicorn.
- Ran NGINX web server to accept and route requests to Gunicorn and expose a public endpoint.
- Sent multiple requests via a script to the public web server URL.

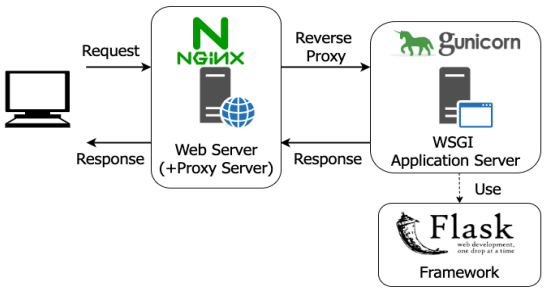

Figure 3. Inside the EC2 instance with application deployed using NGINX, Gunicorn, and Flask.

4.3 EC2 with AutoScaling

An Autoscaling group (ASG) offers the unique feature of increasing or decreasing the computing capacity as needed by our application. This is done by scaling EC2 instances.

4.3.1 Implementation Details: The steps we performed to set up ASG are as follows:

- Set up a launch template to create an AWS auto-scaling group.
- Created an Amazon Machine Image (AMI) image of the EC2 instance containing our application.
- Used the AMI image in the launch template of ASG.
- We set up 2 target-tracking dynamic scaling policies for the ASG to allow instances to scale. One policy was to increase and decrease the current capacity of the group based on a target value of the average number of bytes received by a single instance on all network interfaces. Another scaling policy for was based on CPU utilization capped at 97%. Thus, whenever we reach 97% of CPU utilization in any container, a new instance is spun up.
- Attached an application load balancer to the ASG to distribute requests evenly across instances according to load.
- Sent multiple requests to the load-balanced URL and saw upward scaling in the number of instances.

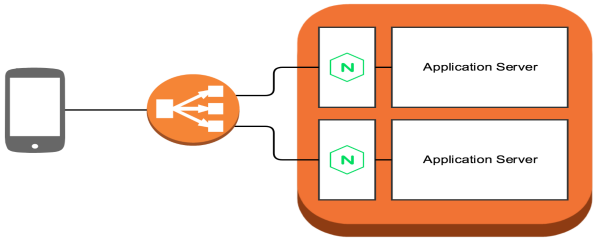

Figure 4. EC2 Auto Scaling Group Architecture

The above steps are depicted in the Fig. 4 where requests are being sent to a load-balanced URL of the ASG group which routes them to the specific EC2 instances.

4.4 Elastic Container Service (ECS)

AWS ECS (Elastic Container Service) is a managed container orchestration platform that enables fast deployment and scaling of containerized workload.

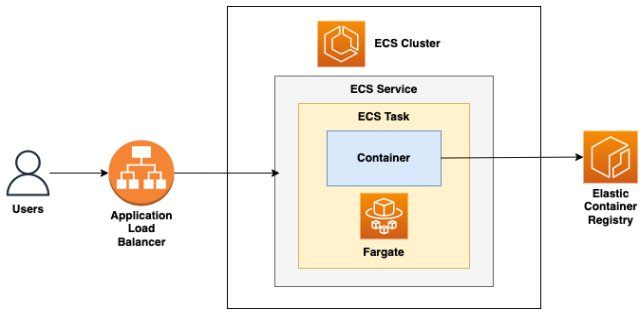

Figure 5. Architecture diagram of ECS

4.4.1 Implementation Details: The steps we performed to set up ECS cluster as shown in Fig. 5 are as follows:

- ECS runs the application via containers, therefore, we created a docker image of our application and stored it in ECR (Elastic Container Registry). AWS ECR is a fully managed container registry offering high-performance hosting that provides reliable deployment of application images and artifacts.
- Created a cluster in ECS with a 'Network Only' or 'EC2 Linux + Networking' template based on Fargate or EC2 respectively. Both these types of clusters provide an option to set up VPC, subnets, and security groups.
- Created the task definition with launch type as Fargate or EC2 that contains details of the desired container, memory allocation, container ports, etc.
- Linked the task definition with the service to ensure the desired number of tasks are always up and running.
- To autoscale our application, we created a load balancer with the desired, maximum, and minimum number of running tasks.
- The scaling policy for ALB was based on CPU utilization capped at 97%. Thus, whenever we reach 97% of CPU utilization in any container, a new container is spun up.

We then evaluated the application performance on ECS with Fargate and ECS with EC2 and compared the results.

4.4.2 Implementation Details for ECS with Fargate: ECS with Fargate allowed us to run our application on AWS Fargate, which allows AWS to fully manage the provisioning and scaling of the virtual machines upon which our containers will run. This means the only thing we need to worry about is defining the CPU and RAM needs of our various tasks and services and not if we have enough underlying compute to run them all.

To run our application on Fargate, we specified the Fargate launch type during task and service creation.

To compare the performances across different services, we allocated the same CPU and memory to Fargate containers as different instance types in EC2.

4.4.3 Implementation Details for ECS with EC2: ECS with EC2 provides more control over the underlying infrastructure but it was harder to create than Fargate. During cluster creation, we selected the instance size of EC2 instances. Because we had to manage the compute for EC2, we only ran one container per EC2 instance to avoid falling into memory and CPU utilization issues.

5 EVALUATION METHODS AND RESULTS

We evaluated three instance types for each of the following four services - 1) EC2, 2) EC2 Auto scaling group (ASG), 3) ECS with Fargate, 4) ECS with EC2. Fargate does not provide an option to specify instance type and only provides an option to allocate CPU and memory for containers. Therefore, we allocated same CPU and memory to Fargate container as an instance type in EC2. For ECS with EC2, we only ran one container per EC2 instance in order to fairly compare it with Fargate.

The graphs created in Section 5.1 show the distribution of the latency metric averaged 5 times for a number of requests for different instance types for each service. We took the average results to make up for any noise. For ASG and ECS (with EC2 and Fargate), we created load balancers with the following configurations:

- Minimum number of running instances = 1
- Desired number of running instances = 2
- Maximum number of running instances = 4
- Scaling policy = 97% CPU Utilization

We used a python script to bombard the load balancers of the respective services wherever applicable. For EC2 instances, we directly used the public DNS to test the load.

In Figures 6, 7, 8, 9 we compare the latency metric across different instances within a service. We can see that as we increase the number of parallel requests, the latency increases for each instance type. For, a smaller number of requests, the average latency remains the same across instances, but for a larger number of requests (100 and above), we see a performance improvement for larger instances (t2.medium and t2.large), i.e., the latency decreases significantly as we increase CPU and memory available. This pattern is seen across all four services.

We also analyzed the CPU and memory utilization across different instance types for each of the services. Figures 11 and 12 show the sample for ECS Fargate. We observed that with the 1 CPU, 2 GiB memory allocation, our application reached almost 95% of CPU utilization for 100 requests.

Whereas when we increased the configuration to 2 CPU, 4 GiB memory, the CPU utilization was much lower (82%). Since new containers are spun up when existing ones reach the CPU utilisation cap specified (97%) in our scaling policy, 2 CPU, 4 GiB instances take longer to scale as their CPU utilisation takes longer to reach the maximum. Creation and deployment of new containers accounts for additional start up time, therefore, we see a steep latency drop between smaller to larger compute instances for the same number of requests because the CPU utilization cap reaches early for smaller instances and hence, more containers are created.

Finally, in Figure 10, we compare performance across different services. As shown in the graph, latency for EC2 is the highest for each request type as there is no auto-scaling for instances. EC2 with ASG and ECS with EC2 show similar performance with ECS with EC2 being a bit faster because VMs take longer to scale up. ECS with Fargate is the fastest since it is based on lightweight containers and the resources are managed internally by AWS.

5.1 Comparison Graphs

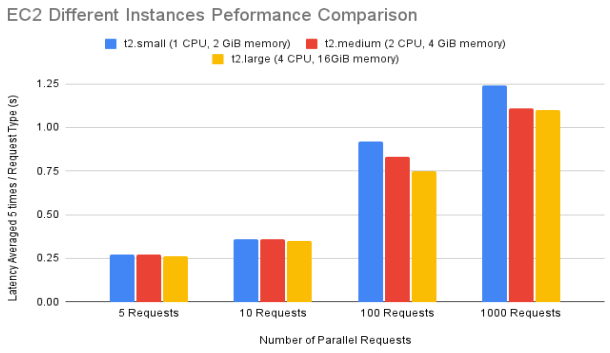

Figure 6. Latency metric comparison across instances types for EC2

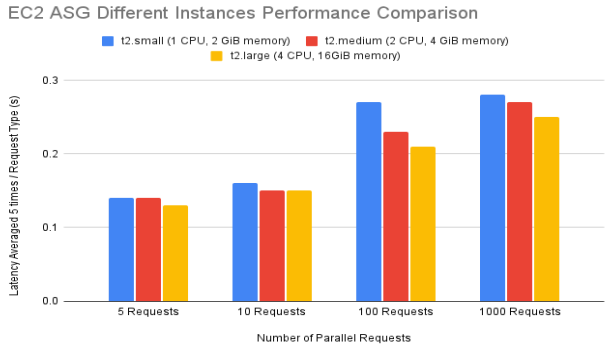

Figure 7. Latency metric comparison across instances types for EC2 ASG

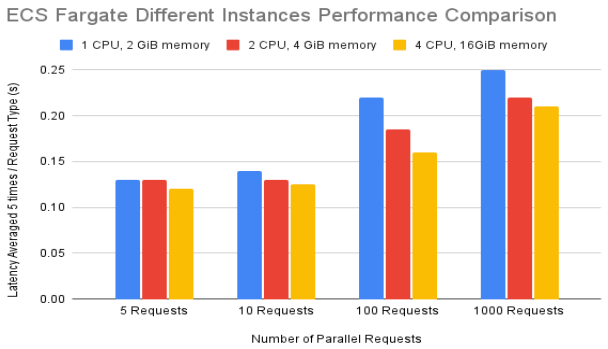

Figure 8. Latency metric comparison across instances types for ECS Fargate

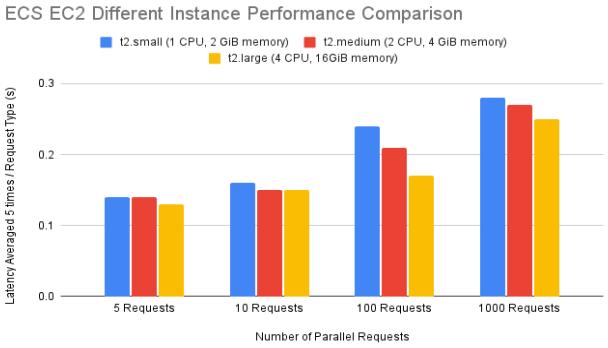

Figure 9. Latency metric comparison across instances types for ECS EC2

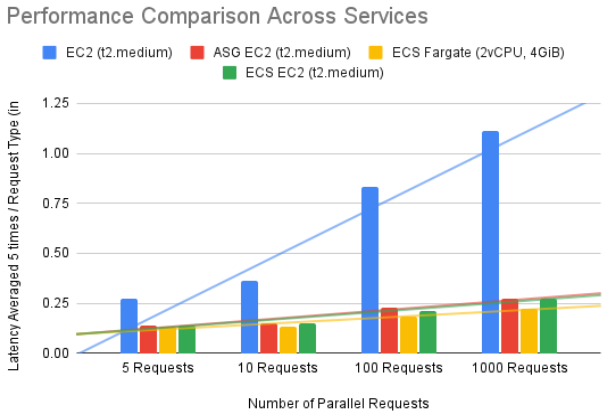

Figure 10. Performance comparison with same compute resources across different AWS services

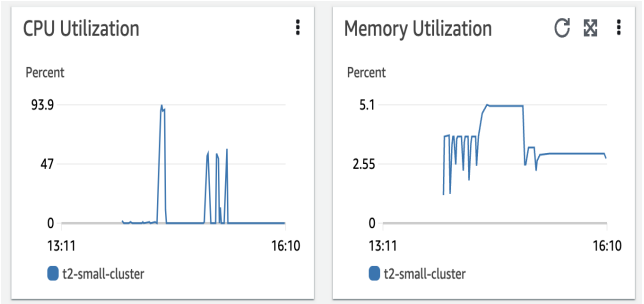

Figure 11. ECS Fargate container CPU and memory utilisation with t2.small config

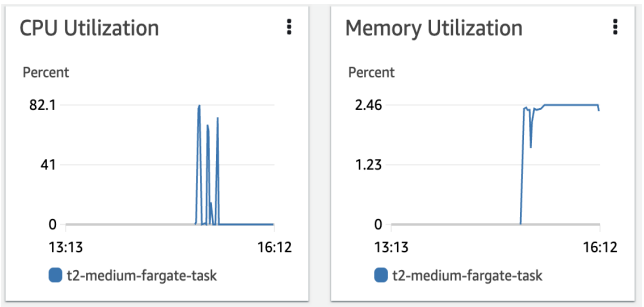

Figure 12. ECS Fargate container CPU and memory utilisation with t2.medium config

The results for each service across these instances for several metrics are summarised below in the Table 1 for 100 parallel requests.

5.2 Comparison across metrics

Latency: There is an expected stark difference between the latency of running the application on EC2 vs the services which allow auto scaling (EC2 with ASG and ECS) as instances are scaled up to manage higher loads. The contrast is more evident in sending a higher number of requests in parallel to the servers as shown in Fig. 10. For 100 and above parallel requests, as instances are spun up, we can also see a latency difference for ECS vs EC2 with auto-scaling. This is because containers are more lightweight and would not require as much time to spin up as more heavy EC2 VM instances.

Scalability: ECS scales up slightly faster than ASG because containers are lightweight. Both EC2 ASG and ECS in terms of defining the automatic scaling policy, require manual adjustment of the scaling policy every time our application load reaches maximum utilization. The instances do not automatically scale lower than a pre-set minimum threshold either so we had to be careful while defining minimum, maximum, and desired capacity to avoid resource wastage.

Cost: Since we ran the same application on all the compute services and did a comparative analysis for similar application loads, the cost of the services is proportional to the cost set by AWS for the services. We saw that ECS with Fargate being a managed service has much higher costs of use compared to ECS with EC2 because we allocated the same number of CPUs and memory to both. Using ECS is free of charge, with users paying for compute costs by the hour. For EC2 with auto-scaling, the associated costs are proportional to the costs for each provisioned EC2 instance.

In Fargate we only pay for container runtime and not for unused VM capacity, therefore, the costs associated with Fargate are usually lower than ECS with EC2 for uneven workloads. Because in EC2, the compute and memory capacities are fixed and defined while setting up. EC2 launch type pricing is based on the CPU and memory of each instance type and is fixed regardless of whether or not it is fully utilized. Fargate pricing is based on task size, but task definitions are limited by the configurations supported by Fargate, and the per-hour cost for instances is higher than with EC2.

Ease of use: ECS with EC2 can be cumbersome to set up as we need to control and set up the underlying infrastructure. ECS with Fargate however offers a managed service and takes infrastructure management out of the equation allowing us to focus on just the tasks and services to be run. Comparing ECS with ASG, in ECS, we had to provision the docker containers beforehand and attach them to the cluster. Whereas, in EC2 with ASG we could reuse the AMI of the EC2 instances in the launch template. Therefore, EC2 with ASG was the easiest to set up.

Security: In general, we don't have to manage the operational issues in Fargate because AWS already does it. But in specific cases such as health care data, finance data, etc, the EC2 is more secure because we have better control over the low-level infrastructure.

When we have to manage our own EC2 clusters and EC2 instances, we need to manually set up security measures such as VM security, operating system patching, maintenance, and uptime which can lead to compromises in security due to mistakes made by developers. Since Fargate uses capacity managed by AWS, we did not need to worry about ensuring EC2 instances remain healthy and secure since AWS does this for us.

Debugging: Debugging can be made comparatively simple in EC2 when equipped with the right debugging tools. We set up logging files to capture logs to understand errors in deploying the application which made debugging much simpler. However, it is more difficult to set up Cloudwatch logging on EC2 as one needs to custom install a Cloudwatch agent, unlike ECS where it is automatically enabled to collect logs. In ECS with Fargate, we don't have the option to control the underlying infra or ssh into the server, however, it provides integration with cloudwatch logs by default. In ECS

AWS Service	Instance type	Latency (in sec)	Billing Time (in minutes)	Cost ($/hr)
EC2	t2.small (1 CPU, 2 GiB memory)	1.14	10.2	0.02
	t2.medium (2 CPU, 4 GiB memory)	0.63	10	0.04
	t2.xlarge (4 CPU, 16 GiB memory)	0.41	9.5	0.18
EC2 Auto Scaling Group	t2.small (1 CPU, 2 GiB memory)	0.27	2.5	0.09
	t2.medium (2 CPU, 4 GiB memory)	0.19	2.3	0.18
	t2.xlarge (4 CPU, 4 GiB memory)	0.14	2.2	0.74
ECS with Fargate	1 CPU, 2 GiB memory	0.22	2.2	0.16
	2 CPU, 4 GiB memory	0.13	2.2	0.32
	4 CPU, 16GiB memory	0.10	2.1	0.48
ECS with EC2	t2.small (1 CPU, 2 GiB memory)	0.20	2.3	0.10
	t2.medium (2 CPU, 4 GiB memory)	0.12	2.3	0.19
	t2.xlarge (4 CPU, 16GiB memory)	0.10	2.1	0.25

Table 1. Latency and Cost Summary Across Services and Instance Types

with EC2, we can log into the instances and check the status of the container. However, we have to explicitly enable the cloudwatch logs to see container logs.

6 CHALLENGES

Some of the challenges we faced are:

- **Creating docker container images**: Setting up the environments including setting up of the web server on docker for ECS and deploying the model endpoint on Sagemaker was one of the toughest challenges we faced. We checked Cloudwatch logs to check for errors in setting up of the model endpoint.
- **Setting up the network configurations**: We faced many issues in creating VPC, subnets, and inbound & outbound security rules to be able to deploy our application on a public URL and be able to access it over the internet.
- **Debugging errors**: We solved debugging challenges by setting up logging within our application. For example, creating a logging file in our Flask application to track our logs and errors.
- **Timeout issues**: We faced several timeout issues while requesting our webserver and fixed those by extending timeout for our Gunicorn service, NGINX server, and load balancer.

7 CONCLUSION

In this project, we did a comparative study of 4 AWS compute services - EC2, EC2 with ASG, ECS with Fargate, and ECS with EC2 on a handwritten digit classification application. We used Sagemaker to train and deploy our model and the compute services to perform inference. We ran various experiments to evaluate the best service for our application. Based on the experiments, we observed that EC2 had the highest latency, whereas, the other three services (EC2 with ASG, ECS with Fargate, and ECS with EC2) had similar and much lower latency than EC2. ECS with Fargate performed the best in terms of latency and ease of use, however, it incurred the highest cost among all the services. Scaling up ECS instances was faster compared to ASG because ECS uses containers whereas ASG spins up VMs. Each service required a different tool and way of debugging, therefore, each had its own learning curve. In terms of comparing different instance types within each of the services, we observed that t2.medium performed significantly better than t2.small for our workload. t2.large performed slightly better than t2.medium but it is also 2x the price of t2.medium.

8 ACKNOWLEDGEMENT

This project was a great opportunity to gain hands-on experience on different AWS services. We now have a better understanding of how to create and deploy VMs, containers, and images and their internal low-level details. The project goes beyond the usual classroom project and hopefully, we will be able to apply the learning from this project to real-life applications. We would like to thank our professor Dr. Yiying Zhang for providing us with guidance and feedback that helped us improve throughout the quarter.

9 REFERENCES

- https://aws.amazon.com/sagemaker/
- https://aws.amazon.com/ecs/
- https://aws.amazon.com/ec2/
- https://docs.aws.amazon.com/autoscaling/ec2/userguide/auto-scaling-groups.html
- https://aws.amazon.com/startups/start-building/how-to-choose-compute-option/
- https://towardsdatascience.com/deploying-your-ml-models-to-aws-sagemaker-6948439f48e1
- https://docs.aws.amazon.com/sagemaker/latest/dg/deploy-model.html
- https://sakyasumedh.medium.com/setup-application-load-balancer-and-point-to-ecs-deploy-to-aws-ecs-fargate-with-load-balancer-4b5f6785e8f