Team Project Report - Phase 3

Phase 3 Summary

Team name: **ThreeCobblers**
Members (names and Andrew IDs):
Leo Guo (jiongtig)
Benny Jiang (xinhaoji)
Yukun Jiang (yukunj)

Performance data and configurations

Please note down the following information for your service in the live test.

Describe your web-tier configuration (ECS Fargate/EKS Fargate/EKS managed node group, what configuration):
We used EKS managed node group with 10 c6g.large instances, and each of the 10 instances is attached to a GP3 volume with 8 Gigabytes with IOPS = 3000 and throughput = 125MB/s. One more NLB for routing.
Describe your storage-tier configuration (which service, what configuration):
We used RDS with MySQL. The instance we used was db.r6g.xlarge with 150 Gigabytes GP2 disk.
Cost per hour of the entire system:
0.1 (EKS cluster) + (10 * 0.068) (10 c6g.large instances) + (10 * 8 * 0.08 / 30 / 24) (10 GP3 volumes with 8 Gigabytes) + 0.0225 (Network Load Balancer) + 0.43 (DB instance) + (150 * 0.115 / 30 / 24) (DB storage) = 1.2653
Your team's overall rank on the live test scoreboard: We believe that we are in top 3.
Live test results for each microservice:

	M1	M2	M3
score	15.00	25.00	25.00
submission id	730661	730662	730663
throughput	185607.74	102492.41	31263.65
latency	2.26	4.00	13.01
correctness	100.00	100.00	99.74
error	0.00	0.00	0.00
rank	4	4	2

Rubric

- Each unanswered bullet point = -4%
- Each unsatisfactory answer = -2%

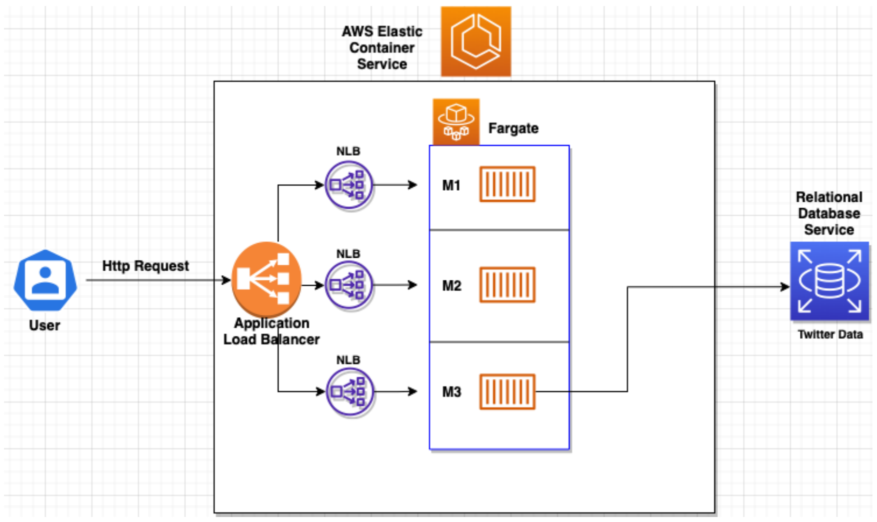
- Use the report as a record of your progress, and then condense it before submitting it.
  .
- If you write down an observation, we also expect you to try and explain it.
- Questions ending with "Why?" need evidence (not just logic).
- Always use your own words (paraphrase); we will check for plagiarism.

Task 1: Improvements of Microservice 1 & 2 & 3

Question 1: Web-tier architecture

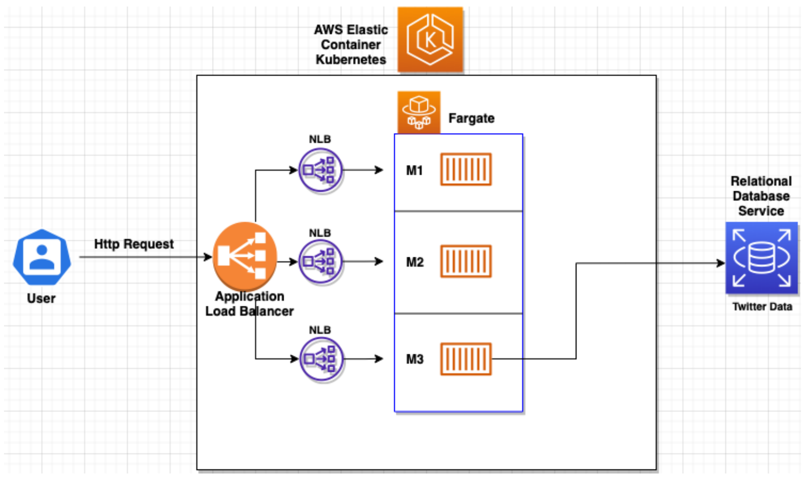
In Phase 3 you can choose from ECS with Fargate, EKS with Fargate, and EKS with managed node groups to host your web-tier. Modernizing your application with a fully-managed cloud service infrastructure is one of the biggest learning objectives of Phase 3. You have learned the concept of containers from Project 2 and built applications on self-managed Kubernetes clusters in Team Project. Now you will have the chance to explore fully-managed Kubernetes clusters.

- Describe and draw graphs about the architecture of ECS with Fargate. (e.g. which pieces are involved in serving a request.)


AWS Elastic Container Service
Fargate
NLB
NLB
NLB
M1
M2
M3
User
Http Request
Application Load Balancer
Relational Database Service
Twitter Data

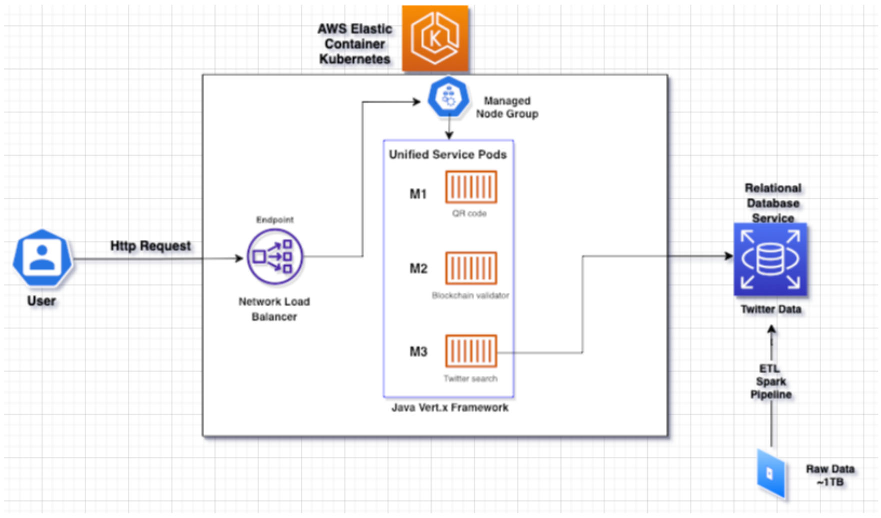
Within the ECS cluster, we use Fargate for deployment and use ALB + 3 NLB routing strategy to maintain the microservice. Fargate has the power to directly manage a set of containers without worrying about the underlying node machine types. The microservice-3 container also links to backend AWS RDS MySQL data

- Describe and draw graphs about the architecture of EKS with Fargate. (e.g. which pieces are involved in serving a request.)


AWS Elastic Container Kubernetes
Fargate
NLB
M1
NLB
M2
NLB
M3
Http Request
User
Application Load Balancer
Relational Database Service
Twitter Data

Within the EKS cluster, much like the ECS + Fargate, it's more or less the same except for the Kubernetes cluster switched to EKS. We use Fargate for deployment and use ALB + 3 NLB routing strategy to maintain the microservice. Fargate has the power to directly manage a set of containers without worrying about the underlying node machine types. The microservice-3 container also links to backend AWS RDS MySQL data

- Describe and draw graphs about the architecture of EKS with managed node groups. (e.g. which pieces are involved in serving a request.)



Using EKS + managed node group, we have the ability to first specify the underlying node machine's hardware architecture and the number of nodes we want to maintain. This gives us a better understanding of the budget incurred. For the rest, it's as usual. But in the final live test, for the purpose of competitiveness, we use monolithic architecture with one single network load balancer. We've already practiced using microservice architecture and it's stated on the piazza that it's OK to use monolithic architecture, we just adopt the most component one we have at hand during the live test.

- Without using SSH to connect to the instances, how would you monitor the three architectures mentioned above?
  We can check the status of each pod by using things like kubectl describe pods. In that way, we can know if our pods are healthy. We can also use things like "kubectl describe" nodes to see if the underlying nodes are healthy.

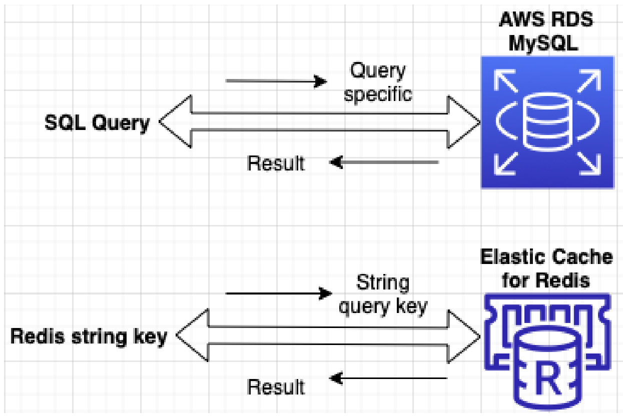
Launch **all** of your microservices using either ECS or EKS with Fargate. Try to maximize your performance while making your web-tier stay under an hourly budget limit of $0.70. We want you to practice microservice architecture. That is, each container must only host one of the microservice. What is the RPS of each microservice?
EKS with Fargate: Micro1: 40194.48; Micro2: 18509.07; Micro3: 13686.80

- Similarly, launch **all** of your microservices using EKS with managed node groups, and maximize your performance while making your web-tier stay under an hourly budget limit of $0.70. What is the RPS of each microservice?
  Micro1: 93801.56; Micro2: 62280.15 Micro3: 32243.47
- Which AWS managed service did you eventually choose for the web tier and why? (You should mention performance/cost ratio, flexibility, and suitability for our use case)
  We used EKS with managed node groups for the web tier. With the same budget, the RPS would be higher if we use EKS with managed node groups. Effectively, EKS with Fargate behaves like adding requests to CPU and memory. For Fargate, even if we do not use some microservices, the resources provisioned to them will not be utilized by the microservice that needs the resources. Thus, we would waste some resources if we choose to use Fargate.
- Describe your configuration of the web tier during the live test and why you choose such a configuration. (e.g. number of instances, instance type, vCPU, memory, replications, etc)
  We used EKS with managed node groups for the web tier. Specifically, we deployed 10 c6g.large instances, and each of the 10 instances is attached to a GP3 volume with 8 Gigabytes with IOPS = 3000 and throughput = 125MB/s. In total, we would have 20 vCPUs, and memory of 40 Gigabytes. Additionally, we used a Network Load Balancer to route the request. A monolithic architecture allowed us to do this. Such configuration can maximize throughput with given budgets.

Question 2: Storage-tier architecture

In Phase 3 you can choose any database engine as long as it's supported by the designated managed services. As you've experienced in Project 3, different database engines are designed under different preferences and you need to choose one that suits Microservice 3. Various schemas can also result in a very large difference in performance! You might want to look at different metrics to understand the possible bottlenecks and think of what to optimize.

- List at least two reasonable storage-tier architecture designs using different DB engines. Explain their mechanisms of processing a Microservice 3 query. (e.g., Use graphs to show which pieces are involved in serving a request.)

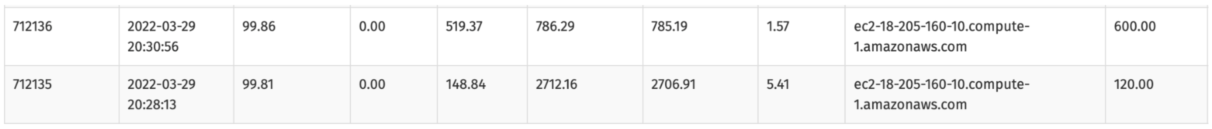


We have thought of two possible ways to implement our storage tier architectures. Firstly, we could use AWS RDS with MySQL and send a query to the database with the user id and related hashtag and keywords to retrieve the required information from it. Secondly, we could use AWS Elasticache Redis. Redis is like an in-memory hashtable with unique string key-value pairs. We could join this architecture with a MySQL database and store the result in Redis. So that if next time this user is queried again, we could directly use the result stored in Redis instead of sending a request to the database again.

- Compare at least two different storage-tiers with the same web-tier configuration and similar cost, and submit your endpoint to the Sail() Platform to see the Microservice 3 performance. Which one did you eventually choose and why? (You should mention performance/cost ratio, usability)
  Amazon RDS + MySQL: 32166.55
  Amazon RDS + Aurora: 15279.09
  We chose Amazon RDS + MySQL. We can achieve higher throughput with it, and it is cheaper than Amazon RDS + Aurora. The cost of Amazon RDS + MySQL is 0.43 + (150 * 0.115/30/24) = 0.454, whereas the cost of Amazon RDS + Aurora is 0.51, if we are using db.r6g.xlarge.
- Describe your final choice of managed services and the architecture for your storage-tier.
  We used Amazon RDS + MySQL. The instance we used was db.r6g.xlarge with 150 Gigabytes GP2 disk.
- What different database schemas have you designed and explored? How did the different schemas impact performance? Which one did you choose? We expect you to provide evidence that supports your decision.
  We tested two different schemas. The first one contains two tables, one for dynamic score calculation (content and hashtag matching), and one for static score (interaction score). This was the first schema we tested, and the commands for building tables are included in the screenshot below.

```
/* Step 1 create the static table1
          for interaction+tashtag score and latest tweet */
DROP TABLE IF EXISTS `static_table`;

CREATE TABLE `static_table`
  (
     `user_a`               BIGINT        NOT NULL,
     `user_b`               BIGINT        NOT NULL,
     `product_score`        DECIMAL(5, 4) NOT NULL,
     `latest_tweet_reply`   VARCHAR(500)  DEFAULT NULL,
     `latest_tweet_retweet` VARCHAR(500)  DEFAULT NULL,
     `latest_tweet_both`    VARCHAR(500)  DEFAULT NULL,
     `latest_screen`        VARCHAR(50)   NOT NULL,
     `latest_description`   VARCHAR(300)  NOT NULL
  ) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

-- -------------------------------------------------------

/* Step 2 create the dynamic table2
        for the storage of hashtags and contents of each tweet */
DROP TABLE IF EXISTS `dynamic_table`;

CREATE TABLE `dynamic_table`
  (
     `is_reply` VARCHAR(10) DEFAULT NULL,
     `content`  VARCHAR(500) DEFAULT NULL,
     `tags`     VARCHAR(100) DEFAULT NULL,
     `sender`   BIGINT NOT NULL,
     `receiver` BIGINT NOT NULL
  ) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;
```

However, such a schema would create a large number of random accesses on disk, which results in a significant IO burst, which leads to degradation of performance. Specifically, We have tested the performance of the database using only one instance during phase 2, and the performance of the database would drop significantly if we increase the duration of testing. The degradation of performance is shown below in the screenshot.

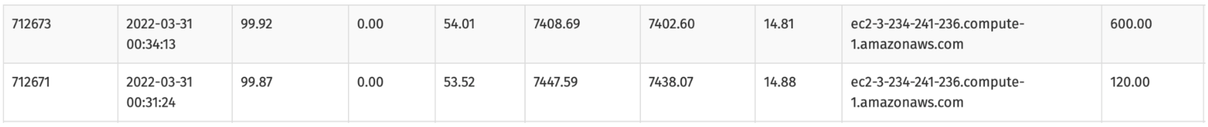


712136	2022-03-29 20:30:56	99.86	0.00	519.37	786.29	785.19	1.57	ec2-18-205-160-10.compute-1.amazonaws.com	600.00
712135	2022-03-29 20:28:13	99.81	0.00	148.84	2712.16	2706.91	5.41	ec2-18-205-160-10.compute-1.amazonaws.com	120.00

For the second schema, we only used one table. Specifically, we combined two tables in the first schema into one unified table. The command for building the table is included in the screenshot below.

```
DROP TABLE IF EXISTS `unified_table`;

CREATE TABLE `unified_table`
  (
     `content`               VARCHAR(500) DEFAULT NULL,
     `is_reply`              VARCHAR(10) DEFAULT NULL,
     `receiver`              BIGINT NOT NULL,
     `sender`                BIGINT NOT NULL,
     `tags`                  VARCHAR(250) DEFAULT NULL,
     `both_latest`           VARCHAR(500) DEFAULT NULL,
     `latest_description`    VARCHAR(250) NOT NULL,
     `latest_screen`         VARCHAR(50) NOT NULL,
     `product_score`         DECIMAL(14, 13) NOT NULL,
     `latest_retweet_reply`  VARCHAR(500) DEFAULT NULL
  )
engine=myisam
DEFAULT CHARACTER SET utf8mb4
COLLATE utf8mb4_bin;
```

After changing to this schema, the number of random accesses dropped significantly, and the throughput improved a lot. Moreover, the performance of the database became more stable. The duration of the test will not influence the throughput anymore. The evidence is shown below in the snapshot.



712673	2022-03-31 00:34:13	99.92	0.00	54.01	7408.69	7402.60	14.81	ec2-3-234-241-236.compute-1.amazonaws.com	600.00
712671	2022-03-31 00:31:24	99.87	0.00	53.52	7447.59	7438.07	14.88	ec2-3-234-241-236.compute-1.amazonaws.com	120.00

Question 3: Optimizations and Tweaks

Frameworks and databases generally require tuning to reach peak performance. Each of them has tens of parameters to tune. Once you are satisfied with the schema, you can start learning about the configuration parameters to see which ones might be most relevant and gather data and evidence with each individual optimization. You have probably tried a few in Phase 1 & Phase 2. To perform even better, you probably want to continue with your experimentation.

- List as many possible items to configure or tweak that might impact performance. You need to provide at least 3 performance optimization techniques for both web-tier and storage-tier.
  Web-tier: load balancing strategy, type of load balancer; the number and distribution of deployment pods; type of instance to deploy web-tier.
  Storage-tier: storage engine; DB schema; key buffer size; Cache size, connection pool size; DB engine; DB schema.
- Show us the benefit of your optimizations. Apply one optimization at a time and show a chart/table with their respective RPS.
  Notice that some of the experiments were conducted during Phase 2.

schema (# of tables)	Storage engine	Key buffer size	Cache size	Pool size/server instance	Best 600s RPS (Micro3)
3	Innodb	16M	0	5	1086(one instance)
2	innodb	16M	0	5	375(one instance)
2	MyISAM	16M	0	5	523(one instance)
2	MyISAM	16M	0	30	524(one instance)
2	MyISAM	256M	0	30	478(one instance)
2	MyISAM	512M	0	5	844(one instance)
2	MyISAM	16M	16M	30	723(one instance)
2	MyIsam	16M	8M	5	924(one instance)
1	MyIsam	16M	0	5	7402(one instance)

1	MyIsam	8M	0	5	12916(on cluster)
1	MyIsam	8M	0	2	10584(on cluster)
1	MyIsam	1G	0	2	20832(on cluster)
1	RDS MySQL; MyIsam	28G	0	2	30629.35(on cluster)
1	RDS Aurora; MyIsam	0	32G	2	15279.09(on cluster)
1	RDS MySQL; MyIsam	28G	0	1	32245.37(on cluster)

- For every configuration parameter or tweak, try to explain how it contributes to performance.
  Notice that we mostly listed experiments results on Micro3 because our Micro1 and Micro2 have pretty stable performance on the same Load Balancer(Network Loadbalancer is faster than Application Load Balancer+NodePort/NLB).
  For Storage Tier, with fewer tables in our schema, we will request fewer queries to MySQL and potentially fewer random accesses to the disk, improving performance and keeping from IO burst. Increasing key buffer size and cache size will also alleviate the load on disk IO and speed up query performance. Besides, MyISAM is generally faster than InnoDB when the task is read-only. Also, we found that RDS Aurora has worse performance than RDS MySQL.

Task 2: Managed Service

Question 4: Cost and Benefit

In Phase 3, you are required to focus on both performance and cost. You have both business constraints as well as non-functional requirements. Hence, you need to calculate the cost and compare the performance of different managed services to make your design decision on the architecture of the web service.

- What is the hourly cost of the managed services you explored (list at least four different managed services)?
  1. AWS RDS MySQL: db.r6g.xlarge with 150 Gigabytes GP2 disk. Cost: 0.43 (DB instance) + (150 * 0.115 / 30 / 24) (DB storage) = 0.454
  2. AWS RDS Aurora: db.r6g.xlarge with 30 Gigabytes storage (since the size of our table is around 25 Gigabytes). Cost: 0.519 (DB instance) + (30 * 0.10 / 30 / 24) (DB storage) = 0.523
  3. EKS Node Group with 8 c6g.large instances and 8 GP3 volumes with 8 Gigabytes with ALB. Cost: 0.1 (EKS cluster) + (8 * 0.068) (8 c6g.large instances) + (8 * 8 * 0.08 / 30 / 24) (8 GP3 volumes with 8 Gigabytes) + 0.0225 (ALB) = 0.674

4. EKS Fargate with 11 vCPU 22 Gigabytes memory with ALB. Cost: 0.1 (EKS cluster) + (0.04048 * 11) (11 vCPU) + (0.004445 * 22) (22 Gigabytes memory) + 0.0225 (ALB) = 0.666

- How did you compare the performance/cost of the managed services you listed above, and what kind of performance/cost results and ranking of the managed services did you get?
  AWS RDS MySQL and AWS RDS Aurora: compare the RPS of micro service 3
  Amazon RDS + MySQL: RPS = 32166.55, performance / cost = 32166.55 / 0.454 = 70851.43
  Amazon RDS + Aurora: RPS = 15279.09, performance / cost = 15279.09 / 0.523 = 29214.32
  Obviously, Amazon RDS + MySQL performs much better, and is much cheaper.
  EKS Node Group and EKS Fargate: compare the RPS of micro service 1.
  EKS Node Group: RPS = 93801.56, performance / cost = 93801.56 / 0.674 = 139171.45
  EKS Fargate: RPS = 40194.48, performance / cost = 40194.48 / 0.666 = 60352.07
  EKS Node Group has a better performance/cost ratio.
- Are there any other managed services you decided not to explore? Explain your reasoning.
  We decided not to try other storage-tier managed services other than RDS because they are not compatible with MySQL databases in nature while our schema is totally for MySQL databases.
- What are the optimizations that you have tried on the managed services and how did these optimizations affect your performance/cost results?
  We tweaked the number of pods and connection pool size on web-tier managed services and found a balance between parallelization and database synchronization overhead. For storage managed services, we changed the default storage engine from InnoDB to MyISAM, and enlarge the key buffer size to largely speed up the read performance.
- In Phase 1 and 2, you deployed your web-tier on self-managed Kubernetes clusters. When using fully-managed services like ECS and EKS, what are the benefits of using them in general? Which of those benefits are helpful in Phase 3? For the less useful features, can you list at least one scenario that which these features become handy or even critical?
  The main benefit of these services is that they are easier to automate, and more cost-efficient since we don't need to pay for many redundant disks. This is especially helpful in Phase 3 where we are assessed by not only the throughput but also the budget. Automation feature of managed services becomes critical when we are updating and testing versions of software frequently.
- In Phases 1 and 2, you deployed the MySQL server by yourself on self-managed Kubernetes clusters. When using managed database services like AWS RDS, what are the benefits of using these in general? Which of those benefits are helpful for Phase 3? For the less useful features, can you list at least one scenario that these features become handy or even critical?
  1. With AWS RDS, the performance of queries becomes better.
  2. It is faster to load data into MySQL on RDS.
  3. It becomes easier to automate the deployment of RDS.

All of these benefits are quite helpful in Phase 3.
Feature 3 becomes especially critical when we are updating and testing versions of software frequently.

Task 3: Development and Operations

Question 5: Web service deployment automation

We know it takes dozens or maybe hundreds of iterations to build a satisfactory system, and the deployment process takes a long time to complete. Setting up services automatically will save developers from repetitive manual labor and will most likely reduce errors. In this phase, you need to use managed services, and you also want to make sure that you can start the services before the live test without any last-minute drama. Therefore, it is worthwhile to devote some time to the auto-deployment of your service. As in past phases, we require the use of Terraform as an orchestration and deployment tool.

- Describe how your team automated the deployment of your web service (both web-tier and storage-tier)? Describe the steps your team had to take every time you deployed your web service.
  We used eksctl to deploy the EKS cluster, Terraform to deploy the RDS MySQL database, and helm to install the microservices onto the EKS cluster. We need to first deploy the EKS cluster using eksctl and the YAML file. Then, after the cluster is available, we need to get the VPC id and the ids of the VPC security groups of the newly deployed EKS cluster, and feed those values into the terraform file used to deploy the RDS MySQL database. Then, we can deploy the database using terraform. After the database is ready, we need to find the endpoint of the newly deployed database, and feed the endpoint into the ConfigMap used to deploy the microservices. Finally, we would install the microservices using helm.
- What are the difficulties you encountered when you tried to connect to your storage-tier from your ECS/EKS web-tier. How did you solve it?
  We haven't met any difficulties when we tried to connect to the storage-tier from the EKS web-tier. We first tried to deploy everything manually using the web console, and it is very informative and easy to use. After deploying everything, we simply modified the endpoint of the database in the helm chart of microservice 3, and everything worked as expected. Then, we started to write the YAML file for the EKS cluster and the terraform script for the database.

Question 6: Live test and site reliability

Your web service in Phase 3 is evaluated similarly to how we evaluated your web service in Phase 2. In Phase 3, you can make as many attempts as you need by the deadline. However, all these attempts do not contribute to your score. Your team's Phase 3 score is determined solely by the live test. The live test is a one-time test of your web service during a specified period of time, so you will not want anything to suddenly fail; in case you encounter failure, you (or some program/script) probably want to notice it immediately and react!

- How would you monitor the health of your system (both web-tier and storage-tier)? What metrics helped you understand the performance of your web service during the live test?

1. The number of healthy instances in the autoscaling group.
2. We can also observe whether there are any drops in disk IO or CPU utilization.

- What other monitoring or data collection tools did you use? What did the statistics tell you?
  We use “describe hpa” to check the CPU utilization of each deployment pod. We also collect the number of pods through kubectl. We also use CloudWatch metrics of the RDS instance to monitor CPU Utilization/IO of the database.
  These statistics are telling us whether the servers are serving requests or are bottlenecked by the database or web-tier instances.
- Should something crash in your system, will your service still be able to serve requests? What have you done to resolve these issues? Will this affect your overall performance? Think about this in different scenarios, such as web-tier, storage-tier, and so on.
  If the disk is throttled, our server can still serve at a lower speed. For program crashes, our program can mostly tolerate it because we have multiple deployments of identical server pods on different machines.

Task 4: Reflection

Question 7: Consistency and Scalability

In this team project, we did not ask you to respond to any write requests. Think about whether your service could provide the same quality under a load of write requests or a growing amount of data.

- What problems might arise if your service served to write requests? What are some problems that you might encounter and how will you change your design to address them?
  We need to be careful about write-read consistency, i.e. ACID property in the database. Particularly when we have many multiple replicas of the same database, if we need to update, only 1 database will receive this request, yet for consistency purposes, it needs to propagate this information to everybody else to make sure at any given time, the data copy at all database storage is the same.
- For only read requests, would your design work as well if the quantity of data was 10 times larger? What about 100x? Why or why not? (Assume you get a proportional amount of machines.)
  Yes, we definitely can handle 10x times larger data. But since we are using a MySQL database backend with many replicas, we probably would not be able to handle 100x data volume. At that time we would probably have to migrate to a NoSQL database schema, for example, HBase.

Question 8: Postmortem

- How did you set up your local development environment? How was the development process using managed services different from what you did in Phase 2?
  Regarding the development process, we use branches for development and merge to the main branch only when a bug-free optimized version passes performance tests.

We have three two stages of testing. Firstly, we test on a single instance to test correctness and baseline performance. Secondly, we test on a cluster for scalability.

Different from phase2, we now automate most of the deployment process in our development.

- Did you attempt to generate a load to test your system on your own? If yes, how? And why?
  Although we generated a simulated load with a bash script to conveniently warm up our service at times, our test mostly relied on the official load generator.
- Describe an alternative design to your system that you wish you had time to try.
  If we had more time, we wish we had tried other storage-tier platforms such as ElasticCache.
- Which were the toughest roadblocks that your team faced in Phase 3, especially for managed services?
  As a matter of fact, we weren't faced with any major challenges while migrating our implementation from Phase 2 to managed services.
- Did you do something unique (any cool optimization/trick/hack) that you would like to share?
  After Phase 2, we didn't devise any trick that is particularly unique in Phase 3.

Task 5: Summary

Question 9: Summary

After Phase 3, you are ready to make important decisions and choices about SQL vs NoSQL, trade-offs between different Cloud Service Providers, sharding vs replication, MapReduce vs Spark, self-managed vs fully-managed services, etc. as you go out to industry. Please answer the following questions based on your experience in phases 1, 2, and 3.

- Which Cloud Service Provider did you use for ETL? Why?
  We used Azure for ETL since we have most credits on Azure, which supported us to finish the entire ETL process. Specifically, we used a large cluster during the ETL process to speed up the computation, which is very expensive. Azure provided enough budget for us to do this.
- Which programming framework did you use for ETL (e.g., MapReduce, Spark, other)? Why?
  We used Spark. As mentioned above, we deployed a large cluster during the ETL process, which can hold the entire dataset in the memory. In such a case, using Spark can speed up the ETL process a lot, so we used Spark.
- Which programming framework did you use for web-tier (e.g., Spring, Flask, other)? Why?
  We used Vert.x. It is non-blocking, so it can handle a lot of concurrencies, which provides significant speedup. Additionally, it is not as large as Spring, which saves space in memory.
- Which kind of database did you use for Microservice 3 e.g., SQL or NoSQL? Explain why and compare to other options by discussing their advantages and disadvantages.
  We used SQL database, because it is easy to implement, and the size of our final

database is small (around 25 Gigabytes). In this case, one single database can hold the entire database, so using the SQL database is a reasonable choice. NoSQL databases are not optimal if the database is not large enough. For example, Hbase is designed to be used for databases that are huge and sparse. However, when working with small databases using a single node, it is not as good as MySQL. Additionally, it does not support join operation, and we used such operation in the initial schema design.

- Compare the entire architecture of your web service in phase 2 and phase 3 in terms of performance/cost ratio, scalability, fault tolerance and the effort to deploy, maintain and monitor the service.

  Phase 2: cost = 0.6973, micro 1 RPS = 77046.71, micro 2 RPS = 56241.94, micro 3 RPS = 21178.38
  Phase 3: cost = 1.2653, micro 1 RPS = 185607.74, micro 2 RPS = 102492.41, micro 3 RPS = 31263.65
  Note that we modified the implementation of microservice 1, so we will not consider the performance/cost ratio in terms of microservice 1. We will use RPS of microservice 2 to compute the performance/cost ratio
  Performance/cost ratio in phase 2: 56241.94 / 0.6973 = 80656.73
  Performance/cost ratio in phase 3: 102492.41 / 1.2653 = 81002.45
  The architecture in phase 2 has a similar performance/cost ratio, compared with the architecture in phase 3.
  The web tier in phase 3 is more scalable. In phase 2, we had one MySQL server on each of the instances. Thus, the web tier is not that scalable, since it is hard to scale out the instance with data in volume, we have to write scripts so that the scaled out database can copy data from the peer. This is not the case in phase 3. We have separated the storage tier from the web tier. Thus, scaling out the web tier will be much easier. Due to the similar reason, the web tier in phase 3 is also more fault tolerant, since we do not need to worry about copying data.
  The architecture in phase 3 is also more easy to deploy. In phase 3, loading data into the database is much easier. We can simply set up a VM sharing VPC with the database, and we can then load the data into the database using the VM easily. However, loading data into the MySQL pods for the first time is very difficult, we need to attach an empty volume to the data directory of the MySQL pod, get into the MySQL using kubectl, and download the data into the pod. After these, we can load the data into the database and take a snapshot of the data directory.
  In terms of monitoring, the architecture in phase 3 is also better, especially for the storage tier. We can check the IO burst credit and CPU utilization using one interface. For architecture in phase 2, we need to check IO burst credit using cloudwatch for EBS, and CPU utilization using the cloudwatch for instance

- What is the coolest optimization/hack/learning you have achieved throughout the team project? (eg. Cost, Scalability, Performance, etc.)

  When using cloud platforms to deploy MySQL databases, it is better to have as few tables as possible. Usually, the block storage service provides volumes with IO credits. Having many tables can increase the number of random accesses on disk, which consumes the IO credits quickly. In such a case, the performance of the database would be throttled. Thus, it is better to have as few tables as possible when deploying databases onto cloud platforms.