Team Project Phase 2 Report

Summary

Team name: BalsamicBaguettes
Members:

Name	Andrew ID
Madison Teague	mteague
Francisco Besoian	fbesoian
Eric Kumara	ekumara

Please color your responses as red and upload the report in PDF format.

Live Test Performance & Configurations

Number and types of instances:
Cost per hour of the entire system:
(Cost includes on-demand EC2, EBS, and ELB costs. Ignore S3, Network, and Disk I/O costs)

Your team's overall rank on the live test scoreboard:
Live test submission details:

	Blockchain	QR Code	Twitter	Mixed Test (Blockchain)	Mixed Test (QR Code)	Mixed Test (Twitter)
score	12.3	4.2	.1	5.699	5.699	5.699
submission id	2327002	2327038	2327074	2327110	2327110	2327110
throughput	21542	20728	53	3342	6880	6
latency	26.1	31.4	941.1	62.8	47.5	872.8
correctness	100	100	62.1	100	100	72.5
error	0	0	1.2	0	0	1.1

Rubric

- Each unanswered bullet point = -4%
- Each unsatisfactory answer = -2%
- Optional Questions = +4%

Guidelines

- Use the report as a record of your progress, and then condense it before submitting it.
- When documenting an observation, we expect you to also provide an explanation for it.
- Questions ending with "Why?" require evidence, not just reasoning.
- It is essential to express ideas in your own words, through paraphrasing. Please note that we will be checking for instances of plagiarism.

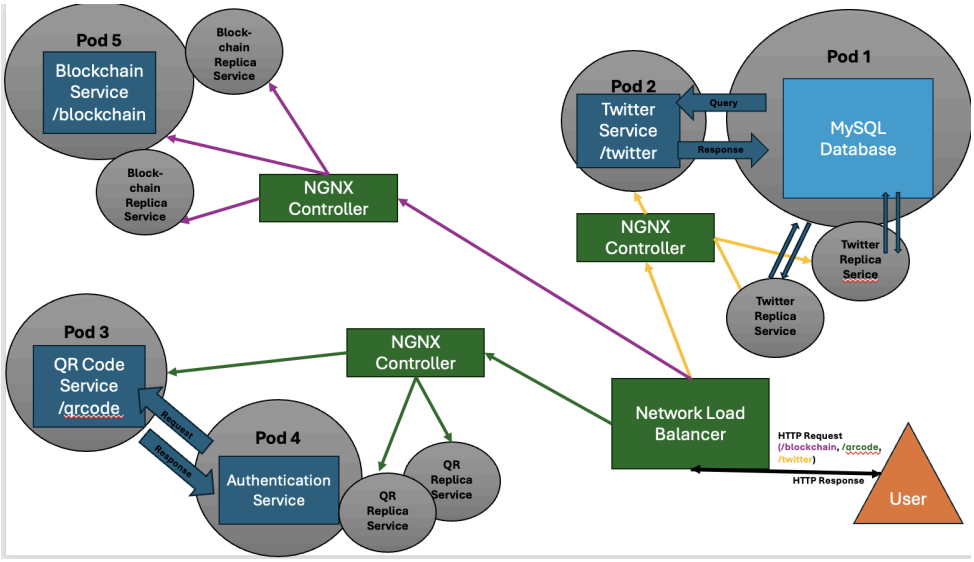
Task 1: Improvements of Microservices

Question 1: Cluster architecture

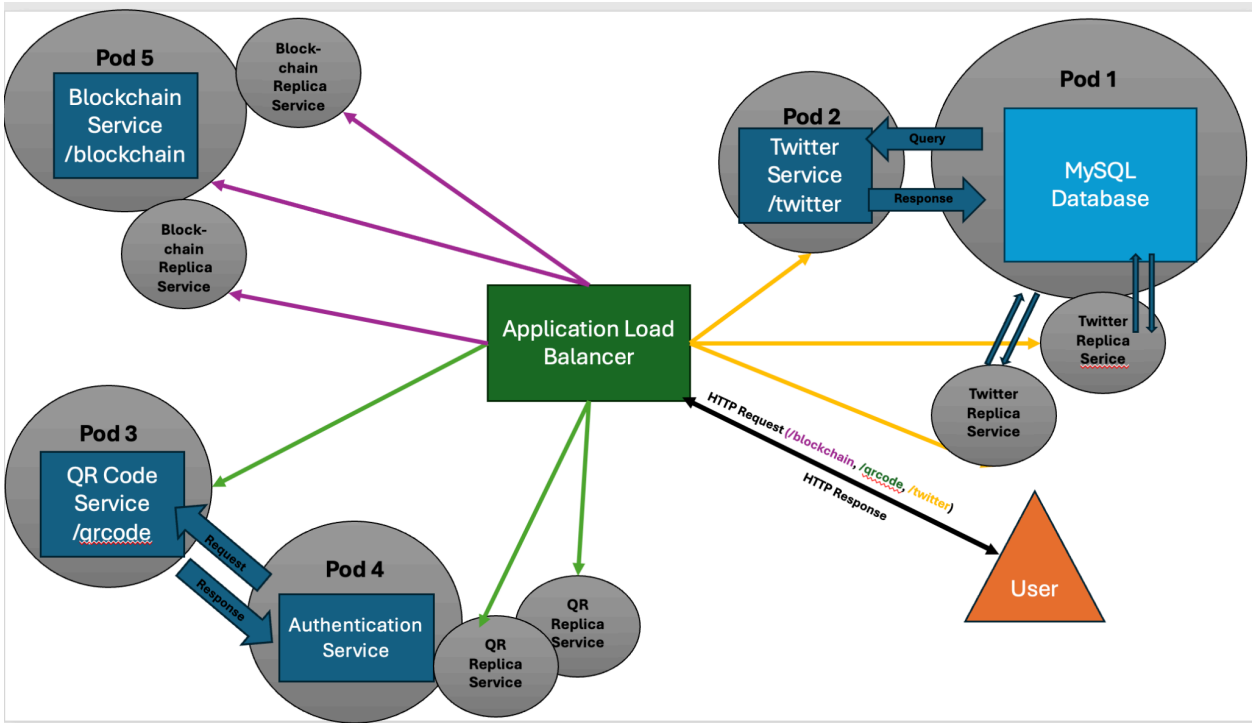
You are allowed to use several types of EC2 instances, and you are free to distribute the workload across your cluster in any way you'd like (for example, hosting MySQL only on some worker nodes, or having worker nodes of different sizes). When you try to improve the performance of your microservices, it might be helpful to think of how you make use of the available resources within the provided budget.

- List at least two reasonable varieties of cluster architectures. (e.g., draw schematics to show which pieces are involved in serving a request).

1) The first variety is what we used for our live test deployment. We have one network loadbalancer outside the cluster which is connected to a ngnx controller inside the cluster with seven replicas (one for each worker node). The purpose of this first loadbalancer is to redirect the traffic to the nginx controller. Then the nginx controller redirects to each service.



2) The second variety is to use an Application Loadbalancer. This would be the only waypoint for the requests to hit before they reach a service. The application Load balancer will not only redirect the requests to the right service, but will also manage the equal distribution of load to the separate service replicas.



- Discuss how your design choices mentioned above affect performance, and why.

1) We used the first variety of cluster architecture in our implementation for our live test. Though it was simple, it caused heavy latency in our system because all our requests had to jump through two redirection hoops, first the network load balancer, and second the controller (which is very slow) . In future deployments we would choose to use the Application Loadbalancer (which we have to implement) because it does the job of both the network loadbalancer and the ngnx controller which will reduce the overall latency.

- Describe your final choice of instances and architecture for your microservices. If it is different from what you had in Phase 1, describe whether the performance improved and why.

In phase 2, we did not see significant improvements in performance because we used the same instance choice and replica configurations on each service. Each service had seven pods requesting the minimum CPU possible, as that matched the number of nodes we deployed in our cluster. They request the minimum CPU so that they are not idly taking up resources. However the pods do not have a maximum on the CPU usage they can request, so that when they are not idle, they can request adequate resources to serve requests. The main difference between our two implementation was to add one ingress for each all services. This caused us to remove the individual load balancers on each service and create the NLB-NGNX Controller system described above, which significantly reduced our performance.

Question 2: Database and schema

If you want to make Twitter Service faster, the first thing to look at is the database because the amount of computation at the web tier is not as much as in Blockchain Service and QR Code Service. As you optimize your Twitter Service, you will find out that various schemas can result in a very large difference in performance! Also, you might want to look at different metrics to understand the bottleneck and think of what to optimize.

- What schema did you use for Twitter in Phase 1? Did you change it in Phase 2? If so, please discuss how and why you changed it, and how did the new schema affect performance?

For phase 1, our group was unable to create the twitter service for the checkpoint, but, for phase 2, we planned different iterations for the table design. For phase 2, we decided to initially start off with 3 different tables: score, user and tweet table. The score table would keep track of the product of the interaction and hashtag score of any 2 users, the user table would keep track of the screen_name and description of each user, and the tweet table would keep track of each tweet with the attached schema below for reference.

Score Table							
min_user_id	max_user_id	score					
user A	user B	1					
User Table							
User_id	screen_name	description					
User A	hello	there					
User B	cloud	cc					
Tweet Table							
min_user_id	max_user_id	type	text	hashtags	timestamp	tweet_id	
User A	User B	retweet	cloud cc	#cooked	today	1	

However, we decided to include redundant data because the schema above would require additional computation to determine which is the min_id.max_id between User A and User B in the endpoint, so we decided to add redundant entries to reduce endpoint calculations/unnecessary joins. The schema user for our current ETL for phase is attached below for your reference. The new database is designed to reduce additional computation overhead in the endpoint by introducing redundant data. However, with proper indexing, query time should not be adversely affected with the additional storage.

Score Table						
user_id	contacted_id	score				
user A	user B	1				
user B	user A	1				
User Table						
User_id	screen_name	description				
user A	hello	there				
user B	cloud	cc				
Tweet Table						
user_id	contacted_id	type	text	hashtags	timestamp	tweet_id
user A	user B	retweet	cloud cc	#cooked	today	1
user B	user A	retweet	cloud cc	#cooked	today	1

- Compare any two schemas for Twitter. Try to explain why one schema is more performant than another.

Before using just 3 tables, our group had a database design involving 5 different tables included below for reference. However, this table schema would introduce additional joins and computational overhead.

Score Table						
user_id	contacted_id	score				
User Table						
User_id	screen_name	description				
Tweet Table						
user_id	contacted_id	type	text	hashtags	timestamp	tweet_id
Contact Table						
user_id	contacted_id	tweet_id				
Interaction Table						
user_id	contacted_id					

The dilemma with database schema is the tradeoff between storage size and computation speed. Since we prioritized computation speed, we decided to move forward with a database scheme with only 3 tables as referenced below.

Score Table						
user_id	contacted_id	score				
User Table						
User_id	screen_name	description				
Tweet Table						
user_id	contacted_id	type	text	hashtags	timestamp	tweet_id

- Explain briefly the theory behind at least 3 performance optimization techniques for databases in general. How are each of these optimizations implemented in your storage tier?

1) We used composite keys for the tweet_table and score_table to use (user_id, contacted_id) as the index; and we used a primary index key for the user_table (user_id).
2) Denormalization of the score table to record the product of the hashtag score and interaction score between any 2 users reduced the computation time overhead
3) Setting the column tables to its appropriate data type such as any ID columns being stored as integers

Question 3: Your ETL process

It's highly likely that you need to re-run your ETL process in Phase 2 each time you implement a new database schema. You might even find it necessary to re-design your ETL pipeline if your previous one is insufficient for your needs. For the following bullet points, please briefly explain:

- The programming model/framework used for the ETL job and justification

Initially, the ETL was done with a Scala file using HDInsight Cluster. However, after trying to change the Yarn yaml files in HDInsight Cluster, the containers were each capped to 1 vCPU core. We also tried changing the Zeppelin interpreter default settings to assign more executors and vCPU cores to no avail. So instead, we decided to use Databricks and PySpark to rerun the same ETL logic. Databricks was successful in overcoming this 1 vCPU bottleneck (utilizing ~95% CPU), and the notebook format allowed quicker debugging since the program can be modularized. We decided to allow Azure Databricks to optimize the vCPU allocation since we were facing issues allocating the vCPU ourselves.

- The number and type of instances used during ETL and justification

When testing with the mini dataset, we used 5 workers and 1 master using Standard_D12_v2 to have 20 Cores and 140 GB. For this, we had more than enough memory to accommodate the 7.5GB file, but we wanted to approximate how long each operation would take. For the full dataset, we initially used 4 workers and 1 master before increasing it to 5 workers using Standard_E8_v3 to obtain 40 cores and 320 GB worth of resources. We approximated that since the data size is

increasing by 20x and the number of cores by 2x, the entire process should only be 10x longer. The memory size was just to accommodate the event we forgot to unpersist the previous run.

- The execution time for the ETL process, excluding database load time

For the full data set, it took 17 minutes to load, 14 minutes to write the first database (user table) and 3 minutes each to write the score and twitter table into the azure blob storage. For the mini set, it took about 70 seconds to load the database, 30 seconds to write the first table and ~5s each for the score and twitter table.

- The time spent loading your data into the database

It took about 17 minutes for the full data set and 70 seconds for the mini dataset. The full set (part 0000-1000) was about 150GB where as the mini (part 0000-0050) was about 7.5GB)

- The overall cost of the ETL process

We started off with a Azure account with ~$200 in student credit, but there was only $22.59 left after the ETL process. Most of the testing was performed with just one part (part-0000.gz) in a local zeppelin, but the mini+large dataset was performed in Databricks.

- The number of incomplete ETL runs before your final run

We did not keep track of the total number of incomplete ETL runs. However, the development process took extensive time in part because the write up in Sail was hard to follow with all the edge cases and requirements. In comparison with blockchain and QR code with reference input/output, this part had no reference which was hard to debug/develop logic for the ETL.

- The size of the resulting database and reasoning

The 3 databases have ~100 million rows between all of them, accounting to about 17 GB worth of memory. This is almost a 90% reduction from the original size of the full database of 150 GB, and it is partly due to removing additional columns not used such as retweeted_status.text.

- The size and format of the backup

The backup happened to be the filtered input data after taking into account duplicate entries and proper columns (such as user_id not null). The backup is stored in azure blob and stored whenever ETL is required to save time loading the full dataset and filtering it.

- Discuss the difficulties encountered

We spent a lot of time trying to optimize the ETL with HDInsight Cluster, but we could not get past the 1vCPU issue even after altering Yarn and interpreter configurations. Additionally, we encountered issues when translating the same logic between scala and PySpark, but this was resolved quite fast in Databricks.

- If you had multiple database schema iterations, did you have to rerun your whole ETL pipeline between schema iterations?

No, because the filtered data was cached. So, after changing the schema, we would only have to rerun some parts since the loading and filtered process do not need to be repeated with the caching process.

- How did you load data into your database? Are there other ways to do this, and what are the advantages of each method? Did you try any of these alternate methods of loading?

The initial loading into the database was with the full data set to be cleaned. Each iteration after date was loading the filtered data from a Azure blob storage which cut down the input memory by 50% (From ~150GB to ~60GB). An alternative method would be to store the filtered data locally and upload it to Databricks, but storing it in the same CSP is likely to be faster for data reading/writing from the internal networking system.

- Did you store any intermediate ETL results and why? If so, what data did you store and where did you store the data?

Yes, we stored the filtered data to avoid recomputing and reloading the full dataset. The filtered data was stored in an azure blob and we stored the tweet table after removing duplicates and filtering out invalid tweets according to the Sail.

- We would like you to produce a histogram of hashtag frequency on a **log** scale among all the valid tweets **without hashtag filtering**. Given this histogram, describe why we asked you to filter out the top hashtags when calculating the hashtag score.

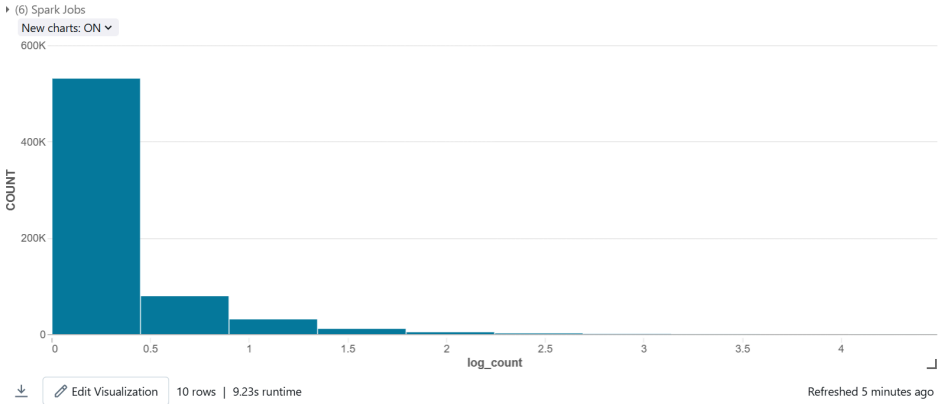
The table below shows the results of the hashtag frequency in log scale. The left table is without the filtering, and the right table is with filtering.

	hashtag_value	log_count
1	android	4.296379953771385
2	sougofollow	3.9072500828813284
3	iphone	3.732634967539196
4	رتويت	3.727378569451489
5	soompiawards	3.5443161417474274
6	nowfifthharmony	3.5281450782531065
7	mufc	3.4583356259919475
8	selfie	3.4202858849419178
9	iphonegames	3.4102709642521845
10	hiring	3.4092566520389096
11	bigolive	3.3740147402919116
12	travel	3.3681008517093516
13	نشر_سيرته	3.3510228525841237
14	モンスト	3.3236645356081
15	art	3.292256071356476

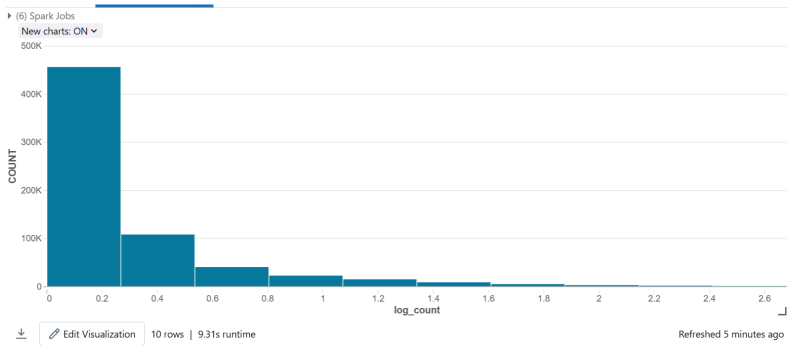
	hashtag_value	log_count
1	anime	2.622214022966295
2	kuwait	2.6074550232146687
3	rm4a	2.6053050461411096
4	videos	2.577491799837225
5	rdmas	2.574031267727719
6	aldubdelightfulduo	2.5728716022004803
7	isac2017	2.567026636615906
8	retweeet	2.5646660642520893
9	欅坂46	2.5634810853944106
10	virgos	2.5622928644564746
11	healthy	2.5538830266438746
12	nikeplus	2.546542663478131
13	justinbieber	2.5428254269591797
14	gop	2.5390760987927767
15	finance	2.5352941200427703

10,000+ rows | Truncated data | 9.98s runtime

Histogram without Filtering



Histogram with Filtering



As shown in the 2 histogram based on just the mini dataset, it removed hashtags that appeared too frequently as indicated by the smaller x-axis max value in the *Histogram with Filtering.* This is likely because these popular hashtags are likely to inflate the hashtag scores which would over represent the overall score between 2 users.

- Before you run your Spark tasks in a cluster, it would be a good idea to apply some tests. You may choose to test the filter rules, or even test the entire process against a small dataset. You may choose to test against the mini dataset or design your own dataset that contains interesting edge cases. **Please include a screenshot of the ETL test coverage report here.** (We don't have a strict coverage percentage you need to reach, but we do want to see your efforts in ETL testing.) Note: Don't forget to put your code for ETL testing under the **ETL** folder in your team's GitHub repo **master** branch.

- Check if the number of rows in the filter is the same as reference

```
import org.apache.spark.sql.functions._

// Filter and select columns
println("Filtering the data")

val dfFilter = df.filter(
    (col("id").isNotNull || col("id_str").isNotNull) &&
    (col("user.id").isNotNull || col("user.id_str").isNotNull) &&
    col("created_at").isNotNull &&
    (col("text").isNotNull && trim(col("text")) =!= "") &&
    (col("entities.hashtags").isNotNull && size(col("entities.hashtags")) > 0) &&
    col("lang").isin("ar", "en", "fr", "in", "pt", "es", "tr", "ja")
).select(
    coalesce(col("user.id"), col("user.id_str").cast("long")).alias("user_id"),
    col("user.screen_name").alias("screen_name"),
    col("user.description").alias("description"),
    col("retweeted_status"),
    col("in_reply_to_user_id"),
    coalesce(col("id"), col("id_str").cast("long")).alias("tweet_id"),
    col("created_at"),
    col("text"),
    col("entities.hashtags")
).dropDuplicates("tweet_id").cache()

dfFilter.show(5)
```

```
Filtering the data
+----------+--------------+------------------------------------+--------------------+-------------------+------------------
--------+
|   user_id|   screen_name|                         description|    retweeted_status|in_reply_to_user_id|          tweet_id|
hashtags|
+----------+--------------+------------------------------------+--------------------+-------------------+------------------
--------+
|1210587492|    SaijMejico|                                null|                null|               null|447295900365238272|
3], Vot...|
| 608830315|       pcb_im| 立教ハモソのベーしすと
とりあえず...|                null|                null|               null|447624301785542656|Sun Mar 23 06:42:...|       【赤ずきんと健康】YADOKARI
| 421994188|Albertoruano95|                'Sentirse Dios cu...|[,, Sun Mar 23 10...|               null|447689787441696768|
7], mor...|
|2398572482|     iwataaimi|相互フォロー支援アカウントです！  ...|                null|               null|449133961805963264|Thu
ロ...|
| 133611203|    NanyDancer|               El amor de mi vid...|[,, Thu Mar 27 04...|               null|449183035200909312|
Horos...|
```

```
%spark

// Sort both DataFrames by tweet_id
val dfFilterSorted = dfFilter.select("tweet_id").orderBy("tweet_id")
val dfReferenceSorted = dfReference.select(col("id").alias("tweet_id")).orderBy("tweet_id")

// Compare if both DataFrames are identical
val areIdentical = dfFilterSorted.except(dfReferenceSorted).isEmpty &&
                   dfReferenceSorted.except(dfFilterSorted).isEmpty

// Print results
println("Number in Filtered: " + dfFilter.count())
println("Number in Reference: " + dfReference.count())

if (areIdentical) {
    println("Both DataFrames have identical tweet_id values.")
} else {
    println("DataFrames have different tweet_id values.")
}
```

```
Number in Filtered: 39092
Number in Reference: 39092
```

- Check if it stores USER ID

```
%spark

import org.apache.spark.sql.functions._
import org.apache.spark.sql.expressions.Window

//Sample Data

val sampleJsonData = """[
  {"created_at": "Mon Mar 25 12:30:45 +0000 2024", "tweet_id": 1001, "user_id": 501, "screen_name": "alice", "description": "Tech enthusia
    "retweeted_status": {"user": {"id": 601, "screen_name": "bob", "description": "Blockchain expert"}}},
  {"created_at": "Mon Mar 25 12:35:10 +0000 2024", "tweet_id": 1002, "user_id": 502, "screen_name": "charlie", "description": "Data Scient
  {"created_at": "Mon Mar 25 12:40:20 +0000 2024", "tweet_id": 1003, "user_id": 503, "screen_name": "david", "description": "Cybersecurity
    "retweeted_status": {"user": {"id": 602, "screen_name": "eve", "description": "AI specialist"}}},
  {"created_at": "Mon Mar 25 12:45:30 +0000 2024", "tweet_id": 1004, "user_id": 504, "screen_name": "frank", "description": "Software deve
]"""

val dfTest = spark.read.json(Seq(sampleJsonData).toDS())

// User Table: Extract retweet users
val dfRetweetUsers = dfTest.filter(col("retweeted_status").isNotNull)
    .select(
        col("created_at"),
        col("tweet_id"),
        col("retweeted_status.user.id").alias("user_id"),
        col("retweeted_status.user.screen_name").alias("screen_name"),
        col("retweeted_status.user.description").alias("description")
    )

// Extract original users
val dfUsers = dfTest.select(
        col("created_at"),
        col("tweet_id"),
        col("user_id"),
        col("screen_name"),
        col("description")
    )

// Combine both user datasets
val dfUsersCombined = dfRetweetUsers.union(dfUsers)

// Process users: Convert `created_at` to timestamp
val dfUsersProcessed = dfUsersCombined.withColumn("created_at_ts", to_timestamp(col("created_at"), "EEE MMM dd HH:mm:ss Z yyyy"))

dfUsersProcessed.show(false)

// Define a window specification for ranking users by latest tweet
val windowSpec = Window.partitionBy("user_id").orderBy(col("created_at_ts").desc, col("tweet_id").desc)

// Rank and filter to keep only the latest record per user
val dfFinalUsers = dfUsersProcessed.withColumn("rank", row_number().over(windowSpec))
    .filter(col("rank") === 1)
    .drop("rank")
    .select(
        col("user_id"),
        coalesce(col("screen_name"), lit("")).alias("screen_name"),
        coalesce(col("description"), lit("")).alias("description")
    )

//expected to have alice, charlie, david, frank

dfUsers.show()

//expected to have bob, eve

dfRetweetUsers.show()

//expected to have alice, charlie, david, frank AND bob, eve

dfFinalUsers.show()
```

```
+-----------------------------+--------+-------+-----------+-----------------------+-------------------+
|created_at                   |tweet_id|user_id|screen_name|description            |created_at_ts      |
+-----------------------------+--------+-------+-----------+-----------------------+-------------------+
|Mon Mar 25 12:30:45 +0000 2024|1001    |601    |bob        |Blockchain expert      |2024-03-25 12:30:45|
|Mon Mar 25 12:40:20 +0000 2024|1003    |602    |eve        |AI specialist          |2024-03-25 12:40:20|
|Mon Mar 25 12:30:45 +0000 2024|1001    |501    |alice      |Tech enthusiast        |2024-03-25 12:30:45|
|Mon Mar 25 12:35:10 +0000 2024|1002    |502    |charlie    |Data Scientist         |2024-03-25 12:35:10|
|Mon Mar 25 12:40:20 +0000 2024|1003    |503    |david      |Cybersecurity researcher|2024-03-25 12:40:20|
|Mon Mar 25 12:45:30 +0000 2024|1004    |504    |frank      |Software developer     |2024-03-25 12:45:30|
+-----------------------------+--------+-------+-----------+-----------------------+-------------------+


+--------------------+--------+-------+-----------+--------------------+
|          created_at|tweet_id|user_id|screen_name|         description|
+--------------------+--------+-------+-----------+--------------------+
|Mon Mar 25 12:30:...|    1001|    501|      alice|     Tech enthusiast|
|Mon Mar 25 12:35:...|    1002|    502|    charlie|      Data Scientist|
|Mon Mar 25 12:40:...|    1003|    503|      david|Cybersecurity res...|
```

- Check if it stores the latest user ID with duplicate user_id

```
%spark

import org.apache.spark.sql.functions._
import org.apache.spark.sql.expressions.Window

//Sample Data for duplicated User entries and timestamp checking

// 3 entries with the same timestamp at 2024, 2 entries with the same timestamp at 2023 for the information of alice and bob

// testing fetching the latest information which would be timestamp at 2023 and tweet_id 1001: alice1 and bob1

val sampleJsonData = """[
  {"created_at": "Mon Mar 25 12:30:45 +0000 2024", "tweet_id": 1000, "user_id": 501, "screen_name": "alice2", "description": "Tech enthusiast2",
    "retweeted_status": {"user": {"id": 601, "screen_name": "bob2", "description": "Blockchain expert2"}}},
  {"created_at": "Mon Mar 25 12:30:45 +0000 2024", "tweet_id": 999, "user_id": 501, "screen_name": "alice3", "description": "Tech enthusiast3",
    "retweeted_status": {"user": {"id": 601, "screen_name": "bob3", "description": "Blockchain expert3"}}},
  {"created_at": "Mon Mar 25 12:30:45 +0000 2023", "tweet_id": 998, "user_id": 501, "screen_name": "alice4", "description": "Tech enthusiast4",
    "retweeted_status": {"user": {"id": 601, "screen_name": "bob4", "description": "Blockchain expert4"}}},
{"created_at": "Mon Mar 25 12:30:45 +0000 2024", "tweet_id": 1001, "user_id": 501, "screen_name": "alice1", "description": "Tech enthusiast1",
    "retweeted_status": {"user": {"id": 601, "screen_name": "bob1", "description": "Blockchain expert1"}}},
  {"created_at": "Mon Mar 25 12:30:45 +0000 2023", "tweet_id": 997, "user_id": 501, "screen_name": "alice5", "description": "Tech enthusiast5",
    "retweeted_status": {"user": {"id": 601, "screen_name": "bob5", "description": "Blockchain expert5"}}}
]"""

val dfTest = spark.read.json(Seq(sampleJsonData).toDS())

// User Table: Extract retweet users
val dfRetweetUsers = dfTest.filter(col("retweeted_status").isNotNull)
    .select(
        col("created_at"),
        col("tweet_id"),
        col("retweeted_status.user.id").alias("user_id"),
        col("retweeted_status.user.screen_name").alias("screen_name"),
        col("retweeted_status.user.description").alias("description")
    )

// Extract original users
val dfUsers = dfTest.select(
        col("created_at"),
        col("tweet_id"),
        col("user_id"),
        col("screen_name"),
        col("description")
    )

// Combine both user datasets
val dfUsersCombined = dfRetweetUsers.union(dfUsers)

// Process users: Convert `created_at` to timestamp
val dfUsersProcessed = dfUsersCombined.withColumn("created_at_ts", to_timestamp(col("created_at"), "EEE MMM dd HH:mm:ss Z yyyy"))

// Define a window specification for ranking users by latest tweet
val windowSpec = Window.partitionBy("user_id").orderBy(col("created_at_ts").desc, col("tweet_id").desc)

// Rank and filter to keep only the latest record per user
val dfFinalUsers = dfUsersProcessed.withColumn("rank", row_number().over(windowSpec))
    .filter(col("rank") === 1)
    .drop("rank")
    .select(
        col("user_id"),
        coalesce(col("screen_name"), lit("")).alias("screen_name"),
        coalesce(col("description"), lit("")).alias("description")
    )

//expected to have alice1 and bob1

dfFinalUsers.show()
```

```
+-------+-----------+-----------------+
|user_id|screen_name|      description|
+-------+-----------+-----------------+
|    501|     alice1|  Tech enthusiast1|
|    601|       bob1|Blockchain expert1|
+-------+-----------+-----------------+
```

-

- Test to check if the number of reply and retweets between 2 users is calculated right

```
%spark
//Sample Data interaction of alice and bob: alice retweed bob 3 times, replied twice. Bob retweed Alice 2 times, replied once.
val sampleJsonData = """[
  {"created_at": "Mon Mar 25 12:30:45 +0000 2024", "tweet_id": 1001, "user_id": 501, "screen_name": "alice", "description": "Tech enthusiast", "in_reply_to_user_id": null,
    "retweeted_status": {"user": {"id": 601, "screen_name": "bob", "description": "Blockchain expert"}}},
  {"created_at": "Mon Mar 25 12:30:45 +0000 2024", "tweet_id": 1002, "user_id": 501, "screen_name": "alice", "description": "Tech enthusiast", "in_reply_to_user_id": null,
    "retweeted_status": {"user": {"id": 601, "screen_name": "bob", "description": "Blockchain expert"}}},
  {"created_at": "Mon Mar 25 12:30:45 +0000 2024", "tweet_id": 1003, "user_id": 501, "screen_name": "alice", "description": "Tech enthusiast", "in_reply_to_user_id": null,
    "retweeted_status": {"user": {"id": 601, "screen_name": "bob", "description": "Blockchain expert"}}},
  {"created_at": "Mon Mar 25 12:35:10 +0000 2024", "tweet_id": 1004, "user_id": 501, "screen_name": "charlie", "description": "Data Scientist", "in_reply_to_user_id": 601, "retweeted_status": null},
  {"created_at": "Mon Mar 25 12:35:10 +0000 2024", "tweet_id": 1005, "user_id": 501, "screen_name": "charlie", "description": "Data Scientist", "in_reply_to_user_id": 601, "retweeted_status": null},
  {"created_at": "Mon Mar 25 12:30:45 +0000 2024", "tweet_id": 1006, "user_id": 601, "screen_name": "alice", "description": "Tech enthusiast", "in_reply_to_user_id": null,
    "retweeted_status": {"user": {"id": 501, "screen_name": "bob", "description": "Blockchain expert"}}},
  {"created_at": "Mon Mar 25 12:30:45 +0000 2024", "tweet_id": 1007, "user_id": 601, "screen_name": "alice", "description": "Tech enthusiast", "in_reply_to_user_id": null,
    "retweeted_status": {"user": {"id": 501, "screen_name": "bob", "description": "Blockchain expert"}}},
  {"created_at": "Mon Mar 25 12:35:10 +0000 2024", "tweet_id": 1008, "user_id": 601, "screen_name": "charlie", "description": "Data Scientist", "in_reply_to_user_id": 501, "retweeted_status": null},
  {"created_at": "Mon Mar 25 12:35:10 +0000 2024", "tweet_id": 1004, "user_id": 15319, "screen_name": "charlie", "description": "Data Scientist", "in_reply_to_user_id": 15619, "retweeted_status": null}

]"""

// Convert the JSON string to a DataFrame
import spark.implicits._
val dfTest = spark.read.json(Seq(sampleJsonData).toDS())

dfTest.show(false)

// Filter replies and compute min_id and max_id
val df_replies = dfTest.filter(col("in_reply_to_user_id").isNotNull)
  .withColumn("min_id", least(col("user_id"), col("in_reply_to_user_id")))
  .withColumn("max_id", greatest(col("user_id"), col("in_reply_to_user_id")))

//expect 3 rows of replies
df_replies.show(false)

// Group by min_id and max_id, and count occurrences
val df_reply_count = df_replies.groupBy("min_id", "max_id")
  .count()
  .withColumnRenamed("count", "count_reply")

// one row for the interaction of bob and alice counting for 3
df_reply_count.show(false)

val df_retweets = dfTest.filter(col("retweeted_status").isNotNull)
    .withColumn("min_id", least(col("user_id"), col("retweeted_status.user.id")))
    .withColumn("max_id", greatest(col("user_id"), col("retweeted_status.user.id")))


//expect 5 entries

df_retweets.show(false)

val df_retweet_count = df_retweets.groupBy("min_id", "max_id")
    .count()
    .withColumnRenamed("count", "count_retweet")

//one row for the interaction of bob and alice counting for 5

df_retweet_count.show(false)

// Join the reply count and retweet count data
val df_interactions = df_reply_count.join(df_retweet_count, Seq("min_id", "max_id"), "outer")
  .withColumn(
    "interaction_score",
    log(lit(1) + lit(2) * coalesce(col("count_reply"), lit(0)) + coalesce(col("count_retweet"), lit(0))) / log(lit(10))
  )

//expect the interaction score to be log(1 + 2*3 + 5) base 10 which is log(12) = 1.079
```

```
|Mon Mar 25 12:35:10 +0000 2024|Data Scientist|601               |null           |charlie   |1004   |501   |501   |601   |
|Mon Mar 25 12:35:10 +0000 2024|Data Scientist|601               |null           |charlie   |1005   |501   |501   |601   |
|Mon Mar 25 12:35:10 +0000 2024|Data Scientist|501               |null           |charlie   |1008   |601   |501   |601   |
|Mon Mar 25 12:35:10 +0000 2024|Data Scientist|15619             |null           |charlie   |1004   |15319 |15319 |15619 |
+------------------------------+--------------+------------------+---------------+----------+-------+------+------+------+


+------+------+-----------+
|min_id|max_id|count_reply|
+------+------+-----------+
|501   |601   |3          |
|15319 |15619 |1          |
+------+------+-----------+


+------------------------------+--------------+------------------+------------------------------+----------+--------+------+------+------+
|created_at                    |description   |in_reply_to_user_id|retweeted_status              |screen_name|tweet_id|user_id|min_id|max_id|
+------------------------------+--------------+------------------+------------------------------+----------+--------+------+------+------+
|Mon Mar 25 12:30:45 +0000 2024|Tech enthusiast|null              |[[Blockchain expert, 601, bob]]|alice     |1001    |501   |501   |601   |
|Mon Mar 25 12:30:45 +0000 2024|Tech enthusiast|null              |[[Blockchain expert, 601, bob]]|alice     |1002    |501   |501   |601   |
```

- Test to check if the tweet table can mark the latest tweet information ordered by timestamp and tweet_id

```
    "description": "El amor de mi vid...",
    "retweeted_status": {
      "user": {
        "id": 421994188,
        "screen_name": "Albertoruano95",
        "description": "'Sentirse Dios cu..."
      }
    },
    "in_reply_to_user_id": null,
    "tweet_id": 449183035200909400,
    "created_at": "Sat Mar 29 01:30:45 +0000 2025",
    "text": "HIGHEST TIMESTAMP AND HIGHEST ID",
    "hashtags": [[[19, 37], "Horos..."]]
  }
]"""

val dfTest = spark.read.json(Seq(sampleJsonData).toDS())

dfTest.show(false)

var df_tweet_left = dfTest
  .withColumn("type",
    when(col("retweeted_status").isNull,
      when(col("in_reply_to_user_id").isNull, "neither").otherwise("reply")
    ).otherwise("retweet")
  )
  .filter(col("type") =!= "neither")
  .withColumn("min_id",
    when(col("type") === "reply", least(col("user_id"), col("in_reply_to_user_id")))
      .when(col("type") === "retweet", least(col("user_id"), col("retweeted_status.user.id")))
  )
  .withColumn("max_id",
    when(col("type") === "reply", greatest(col("user_id"), col("in_reply_to_user_id")))
      .when(col("type") === "retweet", greatest(col("user_id"), col("retweeted_status.user.id")))
  )
  .withColumn("created_at_ts", to_timestamp(col("created_at"), "EEE MMM dd HH:mm:ss Z yyyy"))
  .select(
    col("min_id").alias("user_id"),
    col("max_id").alias("contacted_id"),
    col("type"),
    col("text"),
    col("hashtags").cast("string").alias("hashtags"),
    col("created_at_ts"),
    col("tweet_id")
  )

//filters for tweets that have a reply or retweet

df_tweet_left.show()

// find most recent tweet

// Define the window specification
val windowSpec = Window.partitionBy("user_id", "contacted_id")
  .orderBy(col("created_at_ts").desc, col("tweet_id").desc)

// Add the "rank" column
val df_tweet_left_with_rank = df_tweet_left
  .withColumn("rank", row_number().over(windowSpec))

// Add the "is_latest" column and select the necessary columns
val df_tweet_left_final = df_tweet_left_with_rank
  .withColumn("is_latest", when(col("rank") === 1, 1).otherwise(0))
  .select(
    col("user_id"),
    col("contacted_id"),
    col("type"),
    col("text"),
    col("hashtags").cast("string").alias("hashtags"),
```

```
|421994188|608830315   |retweet|RT @Xarsanmar: Pr...               |[[[125,137], mor...]]     |2024-03-27 12:30:45|447689787441696768|1       |
|608830315|1210587492  |reply  |【赤ずきんと健康】YADOKARI...       |[[[53,64], sm157...]]     |2024-03-26 12:30:45|447624301785542656|1       |
+---------+------------+-------+-----------------------------------+--------------------------+-------------------+------------------+--------+
```

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.expressions.Window
```

- Test to check if it counts the right number of common hashtags between 2 users

```
val sampleJsonData = """[
  {
    "user_id": 15619,
    "hashtags": [[[122, 133], "Aws"],[[122, 133], "azure"],[[122, 133], "STYLE"]]
  },
  {
    "user_id": 15619,
    "hashtags": [[[122, 133], "Cloud"],[[122, 133], "Azure"]]
  },
  {
    "user_id": 15619,
    "hashtags": [[[122, 133], "Cloud"],[[122, 133], "GCP"]]
  },
  {
    "user_id": 15619,
    "hashtags": [[[122, 133], "cloud"],[[122, 133], "aws"]]
  },
  {
    "user_id": 15319,
    "hashtags": [[[122, 133], "cmu"],[[122, 133], "indo"]]
  },
  {
    "user_id": 15319,
    "hashtags": [[[122, 133], "AZure"]]
  },
  {
    "user_id": 15319,
    "hashtags": [[[122, 133], "Cloud"],[[122, 133], "GCP"]]
  },
  {
    "user_id": 15319,
    "hashtags": [[[122, 133], "aWs"],[[122, 133], "style"],[[122, 133], "CLOUD"]]
  },
  {
    "user_id": 15513,
    "hashtags": [[[122, 133], "cmu"],[[122, 133], "Indo"]]
  },
  {
    "user_id": 15513,
    "hashtags": [[[122, 133], "haha"],[[122, 133], "STyle"]]
  },
  {
    "user_id": 15513,
    "hashtags": [[[122, 133], "style"]]
  }
]"""

val dfTest = spark.read.json(Seq(sampleJsonData).toDS())

dfTest.show(false)

// Read the hashtags file with UTF-8 encoding
val popular_hashtags_path = "/mnt/data/popular_hashtags.txt"

val df_hashtags_blacklist = spark.read.option("encoding", "UTF-8").text(popular_hashtags_pat

// Collect the hashtags into a blacklist list
val hashtag_blacklist = df_hashtags_blacklist.as[String].collect().toSet

// Process hashtags
var df_hashtags = dfTest
  .withColumn("hashtag", explode(col("hashtags")))
  .withColumn("hashtag_value", lower(col("hashtag")(1)))
  .filter(!col("hashtag_value").isin(hashtag_blacklist.toSeq: _*))


// Group by user_id and hashtag_value, then count occurrences
df_hashtags = df_hashtags
  .groupBy("user_id", "hashtag_value")
  .count()

df_hashtags.show()

val df_grouped = df_hashtags.groupBy("hashtag_value").agg(
    collect_list(struct(col("user_id"), col("count"))).alias("user_list")
)

val df_pairs = df_grouped.withColumn("user1", explode(col("user_list")))
                         .withColumn("user2", explode(col("user_list")))
                         .filter(col("user1.user_id") <= col("user2.user_id"))

df_hashtags = df_pairs.select(
    col("user1.user_id").alias("uid_1"),
    col("user2.user_id").alias("uid_2"),
    col("hashtag_value"),
    col("user1.count").alias("count_1"),
    col("user2.count").alias("count_2")
)

df_hashtags.show()

df_hashtags = df_hashtags
    .withColumn("min_id", least(col("uid_1"), col("uid_2")))
    .withColumn("max_id", greatest(col("uid_1"), col("uid_2")))
    .withColumn("total_count", col("count_1") + col("count_2"))
    .select("min_id", "max_id", "hashtag_value", "total_count")

df_hashtags.show()

df_hashtags = df_hashtags.groupBy("min_id", "max_id")
    .agg(sum("total_count").alias("sum_total_count"))

df_hashtags.show()

df_hashtags = df_hashtags.withColumn(
    "hashtag_score",
    when(col("min_id") === col("max_id"), lit(1))
    .otherwise(
        when(col("sum_total_count") > 10,
             lit(1) + log(lit(1) + col("sum_total_count") - lit(10)) / log(lit(10))
        ).otherwise(lit(1))
    )
).select("min_id", "max_id", "hashtag_score")

df_hashtags.show()
```

- Test if all the join operations make sense in the end to score the interaction * hashtag score

```
%spark
df_interactions.show()

df_hashtags.show()

val df_hashtags_renamed = df_hashtags
    .withColumnRenamed("min_id", "hashtag_min_id")
    .withColumnRenamed("max_id", "hashtag_max_id")

val df_interactions_renamed = df_interactions
    .withColumnRenamed("min_id", "interaction_min_id")
    .withColumnRenamed("max_id", "interaction_max_id")

val df_score = df_hashtags_renamed.join(
    df_interactions_renamed,
    (col("hashtag_min_id") === col("interaction_min_id")) &&
    (col("hashtag_max_id") === col("interaction_max_id")),
    "right"
).withColumn(
    "score", coalesce(col("hashtag_score"), lit(1)) * col("interaction_
).select(
    col("interaction_min_id").alias("user_id"),
    col("interaction_max_id").alias("contacted_id"),
    col("score")
)
```

```
+------+------+------------------+
| 15513| 15513|               1.0|
| 15619| 15619|               1.0|
| 15319| 15319|               1.0|
| 15319| 15619|1.6020599913279623|
| 15319| 15513|               1.0|
+------+------+------------------+

+-------+------------+------------------+
|user_id|contacted_id|             score|
+-------+------------+------------------+
|    501|         601|1.0791812460476247|
|  15319|       15619|0.7643768731985688|
+-------+------------+------------------+
```

Question 4: Optimizations and Tweaks

Frameworks and databases do not come in the best shape out-of-the-box. 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. To perform even better, you probably want to continue with your experimentation.

Hint: A good schema will easily double or even triple the target RPS of your Twitter Service without any parameter tuning. It is advised to first focus on improving your schema and get an okay performance with your best cluster configuration. If you are 10 times (an order of magnitude) away from the target, then it is very unlikely to make up the difference with parameter tunings.

- List as many possible items to configure or tweak that might impact performance, in terms of web framework, your program, DBs, DB connectors, OS, etc.

1) Reduction of Replicas to match number of nodes deployed in service
2) Adding proper Indexes to DB
3) Adjusting DB Schema to allow for more indexes, and create more intuitive relationships between tables
4) Add connection pooling for DB

- Apply them one at a time and show a table with the microservice you are optimizing, the optimization you have applied and the RPS before and after applying it.

Modification	RPS Before	RPS After
Reduction of Replicas (QR)	45820	56763
Proper DB Indexes (Twitter)	Could not get the page to load	389
Adjust DB Schema (Twitter)	389	2880
Add pooling for DB Connections (Twitter)	2880	131337

- For every configuration parameter or tweak, try to explain how it contributes to performance.

Modification	Contribution to performance
Reduction of Replicas (QR)	Reducing replicas in the QR service matches the deployment infrastructure to actual needs. This eliminates wasteful resource allocation and potential scheduling overhead. When you have more replicas than necessary, you incur extra network communication, synchronization overhead, and resource contention. By right-sizing to match the actual node count, you free up resources for other services and reduce unnecessary coordination.
Proper DB Indexes (Twitter)	Database indexes dramatically improve query performance for read operations by creating sorted access paths to data. Before this optimization, the service couldn't even load the page, indicating severe database bottlenecks. Proper indexes allow the database to quickly locate relevant rows without full table scans, reducing I/O operations and CPU usage. For a Twitter-like service, indexes on user IDs, tweet IDs, and timestamp columns are particularly crucial for timeline generation and user profile views.
Adjust DB Schema (Twitter)	chema adjustments improve data organization and relationship modeling. A well-designed schema reduces the need for complex joins, simplifies queries, and enables more efficient

	data access patterns. By structuring tables to match natural query patterns (like user timelines or tweet engagement metrics), you reduce the computational overhead of assembling results. This optimization likely involved denormalization for read-heavy operations and possibly vertical or horizontal partitioning strategies.
Add pooling for DB Connections (Twitter)	Connection pooling provides significant performance benefits by reusing database connections instead of creating new ones for each request. Database connection establishment is expensive, involving TCP handshakes, authentication, and resource allocation. By maintaining a pool of pre-established connections, you eliminate this overhead for most requests. Additionally, connection pooling helps manage concurrency by limiting the maximum number of simultaneous database connections, preventing database overload during traffic spikes.

Task 2: Development and Operations

Question 5: Web-tier orchestration development

We know it takes dozens or hundreds of iterations to build a satisfactory system, and the deployment process takes a long time to complete. We asked about the automated deployment process in the Phase 1 Final Report. Answer the following questions to let us know how your automation has changed in Phase 2.

- Did you improve your web-tier deployment process? What were the improvements or changes you made? Include your improved/changed Terraform scripts, Kubernetes manifest or Helm charts under the folders referring to Phase 1 folder structures.

The deployment process used in this phase is basically the same as phase 1, however we added an additional tier of load balancing to allow for redirection of the services. We maintained the idea from phase one about the number of replicas. Additionally, we followed the same pipeline in our deployment workflow as in phase 1.

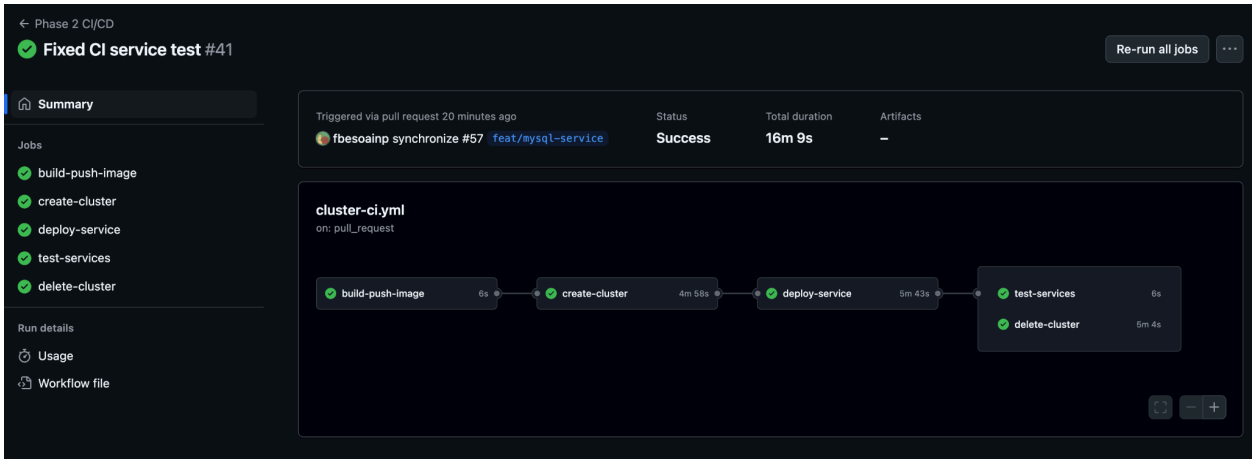
- How are the different steps in your pipeline triggered? Did you use tags or branches to trigger different actions? If so, describe how you structured this.

When we open a pull request to the master branch, the CI will be triggered. In the CI, the go tests for the webendpoint and logic will be run. Additionally the cluster will be tested for health by creating a new cluster with the new configuration, deploying the new images and testing each service health.. This helps us avoid merging code that will break the cluster. When that pull request is merged, the

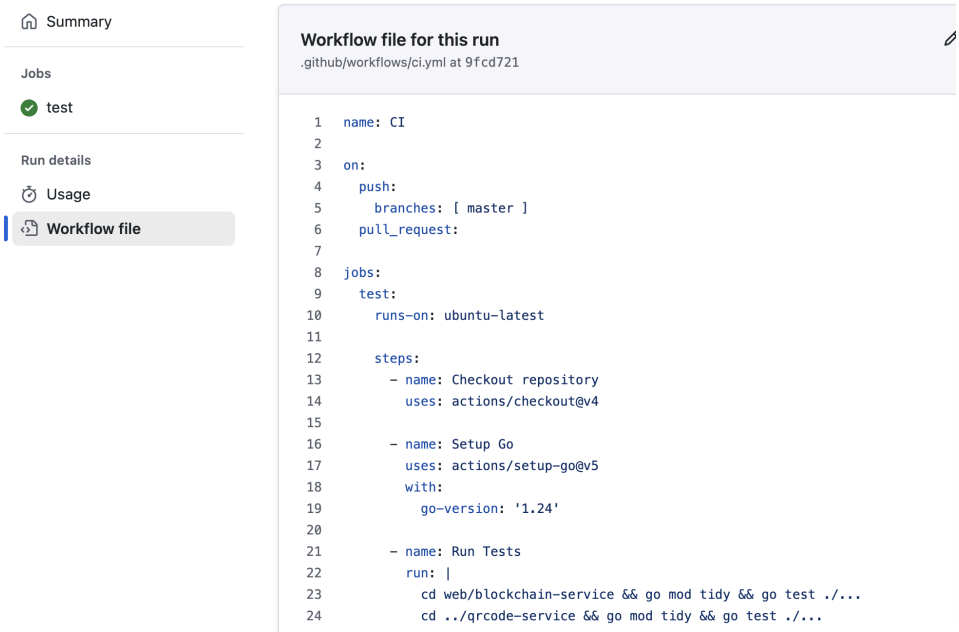
cluster structure will be updated automatically and any new docker images for individual services will be rebuilt and redeployed as necessary.

- Provide a link to your GitHub Actions pipeline page and attach a screenshot showing the workflow and test results.
  Example URL:

https://github.com/CloudComputingTeamProject/BalsamicBaguettes-S25/actions/runs/14205656032



https://github.com/CloudComputingTeamProject/BalsamicBaguettes-S25/actions/runs/14203439791



- [Optional] Do you think GitHub Actions was helpful in your development process? Why or why not? Did you make any modifications or improvements to the GitHub Actions template we provided? If so, what were they?

The actions provided did not install all the dependencies necessary to deploy our system. For example, to create a cluster we needed helm, kops, and kubectl installed, which we had to add ourselves. We also had to change the version of AWS credentials.

Unfortunately, we spent so much time working on the actual development of the recommendation service, we did not get a good chance to utilize the github Actions. However, we did find the testing to be a very useful aid in reviewing merge requests. Since so much was being added in the merge requests, the performance on the workflow checks was a good gauge on whether or not a new code base would successfully execute once integrated into the master branch.

Question 6: Storage-tier deployment orchestration

In addition to your web-tier deployment orchestration, it is equally important to have automation and orchestration for your storage-tier deployment in order to save labor time and avoid potential human errors during your deployment. Answer the following questions to let us know how your storage-tier deployment orchestration has changed in Phase 2 as compared to Phase 1.

- Did you improve your storage-tier orchestration? What were the improvements or changes you made? Include your improved/changed Terraform scripts, Kubernetes manifest or Helm charts under the folders referring to Phase 1 folder structures.

In this phase we deployed our initial version of our storage tier. Our implementation for mysql we have only one replica attached to a persistent volume claim using the snapshot from the ec2. If we were to improve this, we would have the one deployment of the database which allows for many connections connected to more replicas of the service. We also would like to experiment more with how to optimize the db connection settings.

Question 7: Live test and site reliability

Phase 2 is evaluated differently than how we evaluated Phase 1. In Phase 2, you can make as many attempts as you want before the deadline. However, all these attempts do not contribute to your score. Your team's Phase 2 score is determined by the live test. It is a one-off test of your web service during a specified period of time, and so you will not want anything to suddenly fail; if you encounter failure, you (or some program/script) probably want to notice it immediately and respond!

- What CloudWatch metrics are useful for monitoring the health of your system? What metrics help you understand performance?

EC2 Status Checks, Availability (healthy host count), and Newtork Status are all useful for monitoring health of the system. CPU Utilization, Memory Utilization, and Network In/Out, RequestCount, ResponseTime, and Request Latency are all useful for monitoring the performance of the system.

- Which statistics did you collect when logged into the EC2 VM via SSH? Which tools did you use? What do the statistics tell you?

When developing our microservices, we collected CPU Utilization and Memory utilization using top and vmstat. This helped us identify the resource bottlenecks in our system. We also collected Network throughput and query execution times using netstat and the mysql monitoring commands

SHOW STATUS and SHOW PROCESSLIST. These helped us monitor the workload patterns and our system's ability to respond to requests.

- Which statistics did you collect using kubectl? What do the statistics tell you?

We used kubectl to monitor the health and performance of our nodes and pods. Kubectl get pods showed all the pod statuses and we could get more detailed information like resource consumption and event performance using kubectl top and kubectl get events. These statistics told us about our system's health, effectiveness of our scaling on performance, and finally the efficiency of how we are allocating resources to each pod.

- How did you monitor the status of your storage tier? What interesting facts did you observe?

We monitored the storage tier using MYSQL commands like SHOW ENGINE INNODB STATUS, which showed the transaction processing and I/O operations, SHOW PROCESSLIST to observe the progress of active queries and identify inefficiencies. One of the interesting things we observed is that some indexes did not end up being used. This allowed us to remove the unused indexes and save on storage space.

- How would your microservices continue to serve requests (possibly with slower response times) in the event of a system failure? Consider various scenarios, including program crashes, network outages, and even the termination of your virtual machines.

Replication plays a large role in the resiliency of our services. In the event of failure of one, or a few, replicas of our services or VMs, our system can continue to serve clients from the healthy replicas. We also can take advantage of the Kubernetes self healing container orchestration which automatically restarts and reconnects containers to the micro service after a program crash. With network outages, we allow for retries which should help give time to reconnect after restart. If allowed we could use some kind of cached data in the future as well,, though we did not implement or attempt to implement this.

- During the live test, did anything unexpected happen? If so, how big was the impact on your performance, and what was the reason? What did you do to resolve these issues? Did they affect your overall performance?

After the warm up, we saw that our accuracy for QR code was less than 100% which was abnormal since all of our previous tests did not display that behavior. Upon further investigation we saw that it looked like a connection issue; the feedback showed that our authentication service was unavailable. We corrected this by adding more replicas to the auth services to match the node count. When we got to the qr test of qr it did correct our accuracy issue, however it is possible that our service was overloaded due to the increased service count.

Task 3: General questions

Question 8: Scalability

In this phase, we serve read-only requests from tens of GBs of processed data. Remember this is just an infinitesimal slice of all the user-generated content on Twitter. Can your servers provide the same quality of service when the amount of data grows? What if we realistically allow updates to tweets? Here are a few good questions to think about after successfully finishing this phase.

- 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.)

Our current architecture is well-suited to handle a 10x increase in data, as our containerized services can scale independently, and load balancers efficiently distribute traffic. However, a 100x increase in data would present significant challenges, as simple scaling alone wouldn't resolve access pattern issues. At that scale, implementing a caching layer for the Twitter service would be essential—something we currently lack.

- Would your design work if your web service also implemented insert/update (PUT) requests? Why or why not?

Our design is not well-optimized for PUT operations, as handling concurrent writes requires a significantly more complex implementation to ensure consistency. Since our services are built exclusively for read operations, we would be unable to maintain the correct semantics for users. Additionally, this would considerably slow down processing, reducing our capacity to serve customers efficiently..

- What kind of changes in your schema design or system architecture would help you serve insert/update requests? Are there any new optimizations that you can think of that you can leverage in this case?

In our database, the denormalization of some data to avoid expensive joins makes it difficult to add and update data. We could normalize the database reading of read performance in exchange, also use a write ahead logging approach which would allow us to derive the required state to share with clients with necessary, but save time on forward processing. Finally, we could introduce partitioning, likely by timestamp, which would help us keep our write operations limited to smaller tables.

Question 9: Postmortem

- How did you set up your local development environment? Did you install the databases locally on your machines to experiment? Did you test all your programs before running them on your Kubernetes cluster on the cloud?

When working with the mini dataset we tested the database creation and ETL process locally. Additionally, we used Zepplin to verify the accuracy of our data processing methods before moving to a cluster to execute the full ETL process. We did this with all our services before running them on the cloud. To test these locally, we used dependency scripts, which could then be used on individual instances as well, to install the required packages to run these tests.

- Did you attempt to generate a load to test your system on your own? If so, how? And why? (Optional)

We did try to generate a load test locally. Because of the local nature of the testing, there was a limitation on the number of maximum connections allowed in our machines, which meant that the test results were not very useful. In order to execute the test, we developed a script which created client GET requests at different rates (varied the load over time). Other than the local limitation, the fact that we did not try this with the combined service deployment really made the results unhelpful. In the future we could write a better test script and use it while the cluster is fully deployed to get more meaningful results.

- Describe an alternative design to your system that you wish you had time to try.i

The load balancing aspect of the system is one we really wish we had time, and the knowledge to try. We will definitely be implementing it in the next phase. Additionally, we wish we had more time ot play around with the deployment architecture of the twitter service. Potentially having one backup storage replica (though this would be expensive) or a better connection schema would have been idea. Additionally we would like to investigate how partitioning or other denormalization could increase the performance of our twitter service.

- What were the toughest roadblocks that your team faced in Phase 2?

We spent the most time working on the twitter service for phase 2. One of the most unique challenge we faced was escaping special characters in the tweet content and user description fields when we loaded them into our database. It was difficult to even identify this bug at first, all we knew was that the imported tables did not have the same number of rows as the spark processed tables. It took a long time to find the examples that showed this problem in both our tweet and user tables. Once we found it, it took even longer to figure out how to properly escape the characters (\ and “”) for use. Even after the fact, we could not get the order of the displayed tweets right on the web endpoint. After much experimentation, we suspect there is an issue in the calculation of our keyword score.

- Did you do something unique (any cool optimization/trick/hack) that you would like to share?

Once we got the rows of the table correct, we were still getting extra quotation mark. We simply added a step to the response generation that removed these extra quotes. Additionally, when we were initially writing the query to get the important contact user information from the database, we did not taking into account the redundant entries. Changing some of the conditions in the table to remove an OR statement drastically improved the response time from our database.

Question 10: Spark and ETL Process

- Did the concepts and tasks from Project 3 (Spark project) assist you in your ETL process for the Team Project? If so, in what ways did they contribute to your understanding or implementation? How did the Spark tools or techniques help, if at all, in optimizing or managing the ETL pipeline? Would you approach the ETL process differently without Spark?

The concepts and tasks from Project 3 helped in creating an ETL process that minimized expensive operations. As best as possible, we utilized lazy operations to avoid unnecessary shuffling.

For the implementation, we referred to the dataframe of the pagerank implementation for reference on how to use dataframe manipulation instead of RDD.

The spark tools such as the Yarn UI helped in determining the allocation of resources and Spark UI helped identify the bottlenecks of the operation. Some parts of the ETL had CPU Idle period, but it proceeded smoothly after a few minutes allowing for a lagged operation to finish.

If we were to approach the ETL process differently without Spark, we would start with Databricks if Spark UI and Yarn UI are not available because the optimization of vCPU is handled automatically.

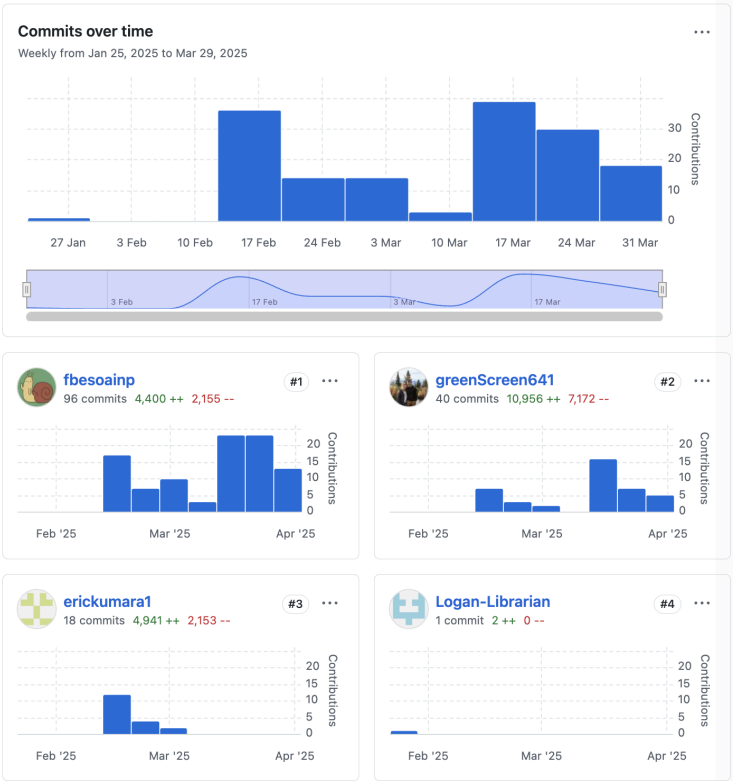
Question 11: Contribution

- In the Phase 1 Final reports, we asked about how you divided the exploration, evaluation, design, development, testing, deployment, etc. components of your system and how you partitioned the work. Were there any changes in responsibilities in Phase 2? Please show us the changes you have made and each member's responsibilities.

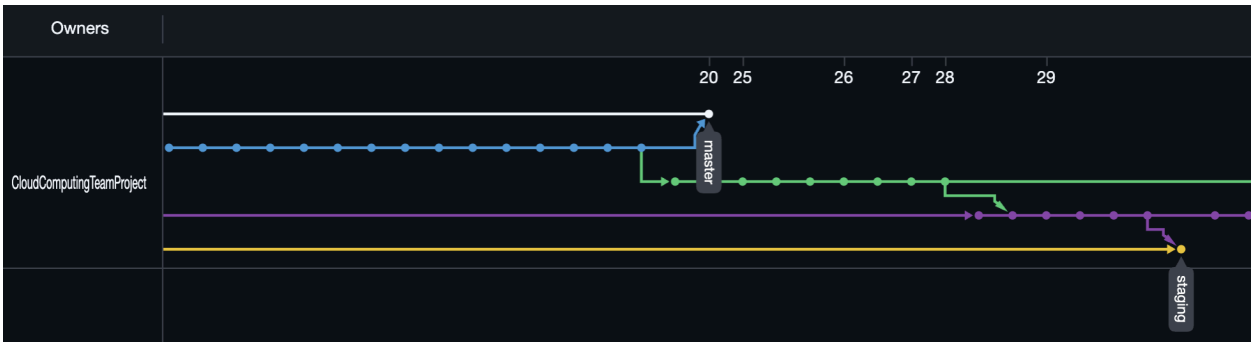
Since we were unsuccessful at creating a deployable we put a lot of resources and time into getting the twitter service up and working. In this phase, Madison and Eric worked pretty explicitly on ETL and the twitter service. Eric focused on the data processing and the creation of the final, pre-computed, database tables. Madison focused on the loading of that data into a mounted database, setting up the web endpoint, and getting the final response back to clients. Francisco created the deployment framework for the twitter service and improved the performance of our blockchain and qr code services.

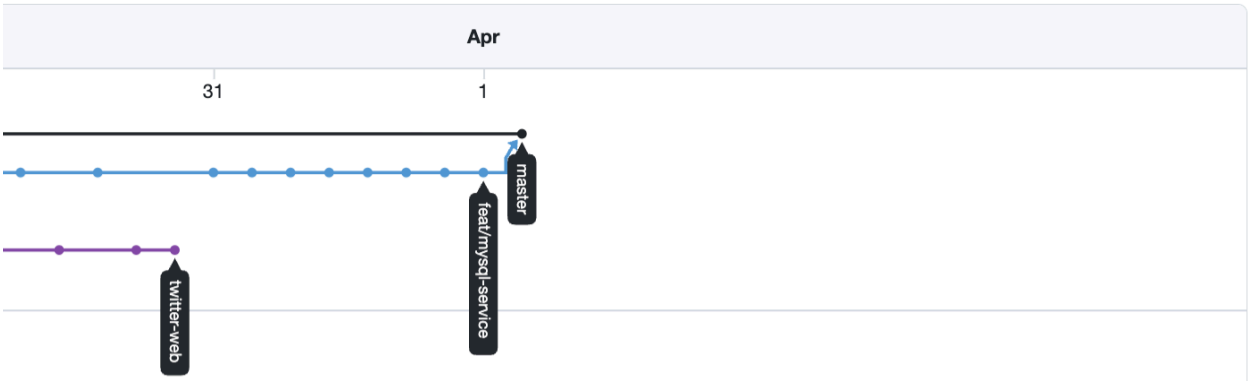
- Please show Github stats to reflect the contribution of each team member. Specifically, include screenshots for each of the links below.

  https://github.com/CloudComputingTeamProject/<repo>/graphs/contributors



  https://github.com/CloudComputingTeamProject/<repo>/network


Owners
20
25
26
27
28
29
CloudComputingTeamProject
master
staging


Apr
31
1
master
feat/mysql-service
twitter-web