Team Project Report - Phase 3

Phase 3 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.

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):

EKS managed node group: 7x C8Large

Describe your storage-tier configuration (which service, what configuration):

AWS RDS:db.r7g.xlarge

Cost per hour of the entire system (detailed calculation): $1.22/hour

cluster: is $0.08 * 7(c8g.large instances) + 0.1 (EKS fixed cost) = 0.66

loadbalancer is ~$0.003

RDS database is $0.555 (db.r7g.xlarge)

Your team's overall rank on the live test scoreboard: 30

Live test results for each microservice:

	Blockchain	QR Code	Twitter	Mixed Test (Blockchain)	Mixed Test (QR Code)	Mixed Test (Twitter)
score	25	15	12.8	5	5	5
submission id	2439842	2439775	2439842	2439911	2439911	2439911
throughput	46906	99059	5119	24713	27691	4366
latency	18.0	6.4	80.2	8.1	16.2	14.7
correctness	100.0	100.0	99.9	100.0	100.0	99.9
error	0.0	0.0	0.0	0.0	0.0	0.0

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 explain 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: 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.)

1. User Request
   - A user (e.g., browser or API client) sends an HTTP(S) request to your application.

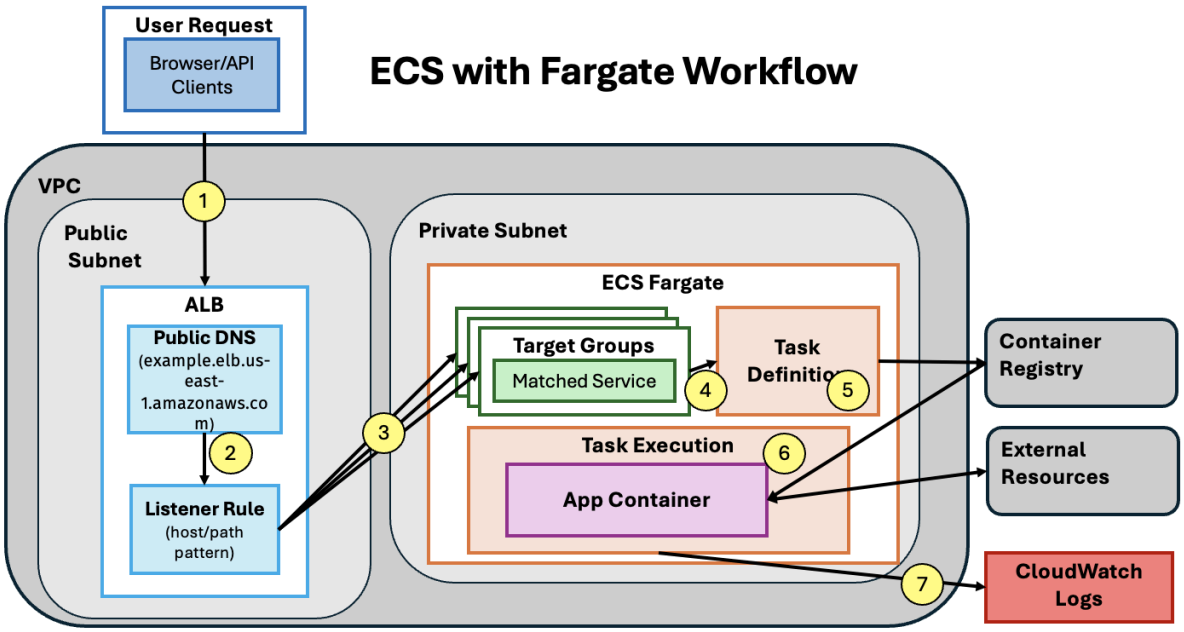
2. Application Load Balancer (ALB)
The request hits an ALB which:
   - Has a public DNS name.
   - Uses listener rules to match host/path and route requests to target groups.

3. Target Group
   - Each ECS Fargate service is registered in one or more target groups.
   - A listener rule forwards requests to the correct target group (e.g., /blockchain → blockchain service).

4. ECS Service (on Fargate)
Each service:
   - Defines a Task Definition (container image, environment vars, port, CPU, memory).
   - Runs as Tasks on Fargate (no EC2, fully managed serverless).
   - Before a task starts, Fargate pulls the image from ECR (Elastic Container Registry) or another registry.

5. App Container
   - The app container handles the request (e.g., blockchain, auth, etc.).
   - It may query databases, external APIs, or other services.

6. Logs and Metrics
   - Logs are sent to CloudWatch Logs.
   - Metrics are available via CloudWatch Metrics and Container Insights.



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

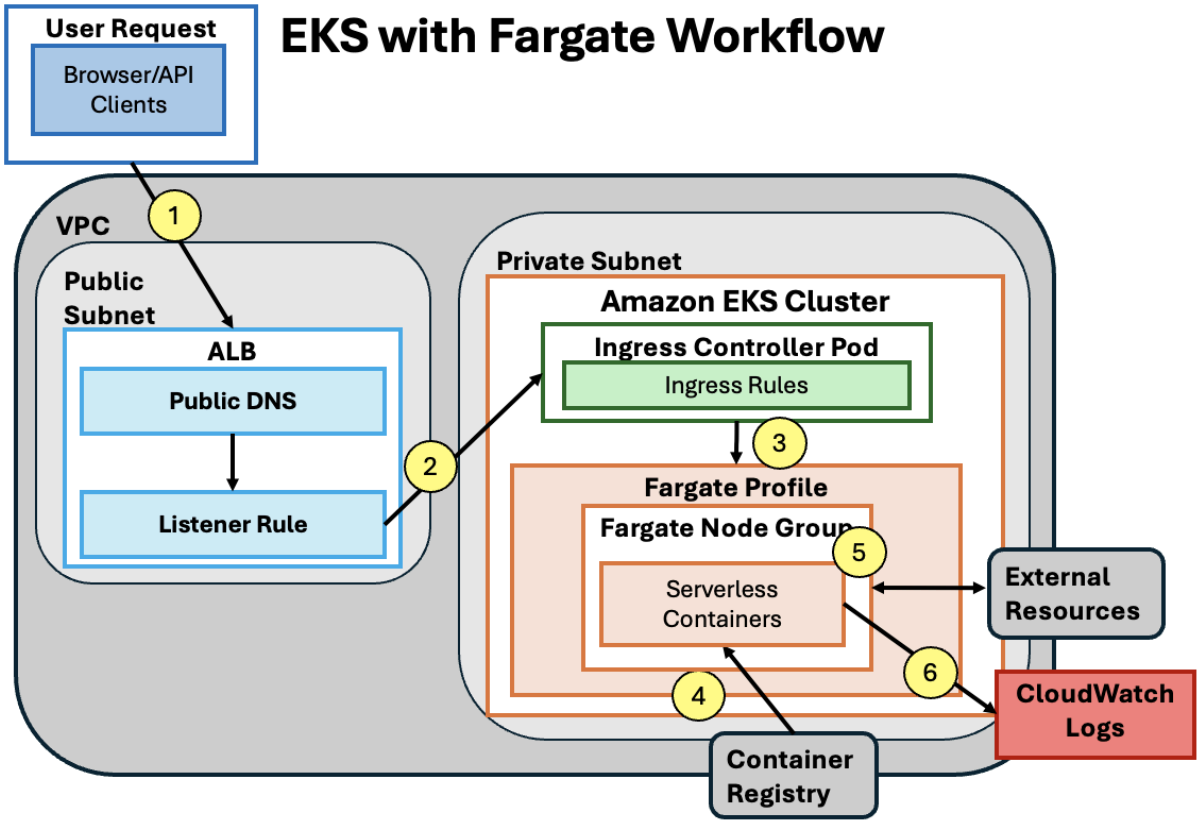
1. User Sends Request
   - A user (e.g., browser or API client) sends an HTTP(S) request to your application.

2. Application Load Balancer (ALB)
The request hits an ALB which:
   - It uses listener rules (host/path based) to determine how to forward requests.
   - It forwards requests to a Kubernetes Ingress Controller (usually the AWS ALB Ingress Controller or NGINX Ingress running inside the cluster).

3. Ingress Controller (Kubernetes Pod)
   - The ingress controller (AWS ALB Controller) pod (running on Fargate) interprets the request routing rules defined in your Kubernetes Ingress resources.
   - It decides which Kubernetes Service should handle the request.

4. Fargate Pod (Task)
   - Each pod is a container running on AWS Fargate.
   - Fargate provides serverless compute—no EC2 management is required.
   - Before a task starts, Fargate pulls the image from ECR (Elastic Container Registry) or another registry.

5. CloudWatch Logging
   - Logs and metrics from the pod are sent to Amazon CloudWatch Logs for observability.



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

1. User Sends Request
   - A user (e.g., browser or API client) sends an HTTP(S) request to your application.

2. Application Load Balancer (ALB)
The request hits an ALB which:
   - It uses listener rules (host/path based) to determine how to forward requests.
   - It forwards requests to a Kubernetes Ingress Controller (usually the AWS ALB Ingress Controller or NGINX Ingress running inside the cluster).

3. Ingress Controller (Pod)
   - Inside the Kubernetes cluster, an Ingress Controller (AWS ALB Controller) receives the traffic and uses Ingress rules to route it.
   - These rules map host/path patterns to Kubernetes Services.

4. Pod (Running on EC2 Node)
   - The pod runs on an EC2 instance that belongs to a Managed Node Group.
   - It executes your application logic (e.g., blockchain, QRcode, Twitter service).
   - The pod pulls the container image from Amazon ECR.

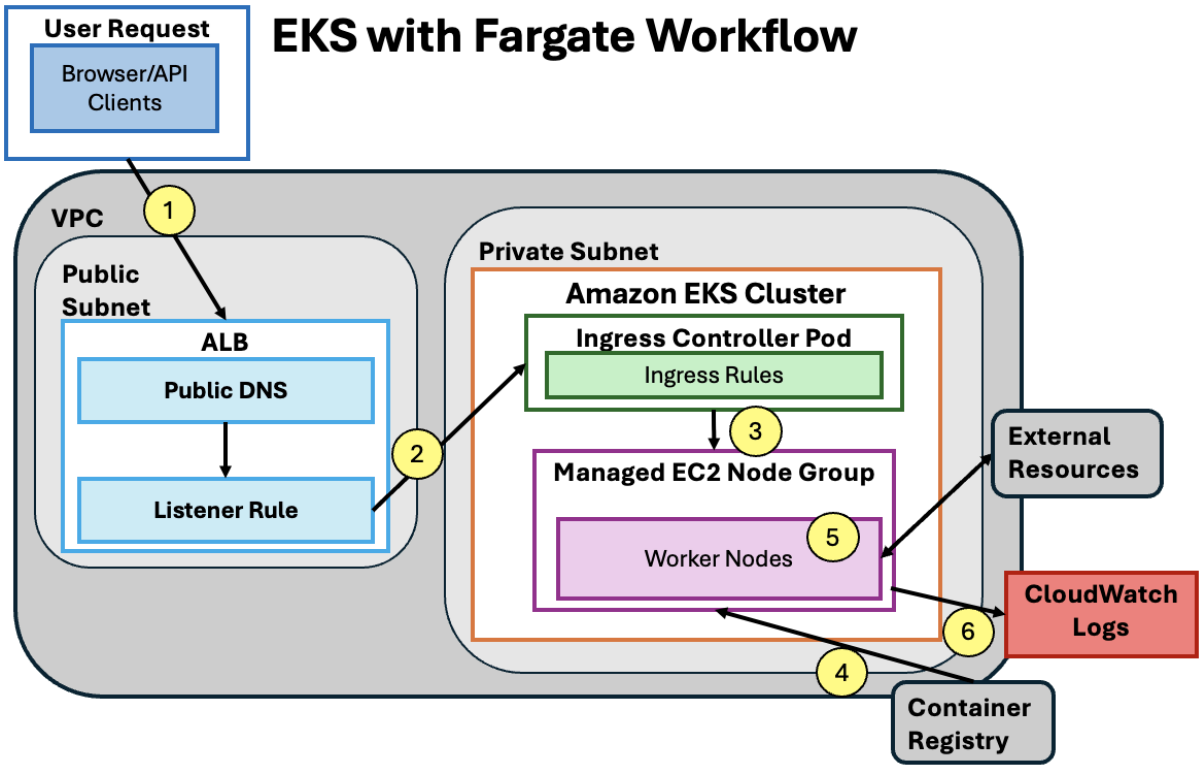
5. Node Group & EC2 Infrastructure
   - The pod is scheduled to an EC2 instance that is part of a Managed Node Group.

AWS handles:

- Launching EC2 instances.
- Auto-scaling node groups.
- Updating AMIs with Kubernetes patches.

6. CloudWatch Logging

- Logs and metrics from the pod are sent to Amazon CloudWatch Logs for observability.



- Without SSH to connect to the instances, how would you monitor the three architectures mentioned above?

We can set up AWS CloudWatch for metrics, logs, and custom dashboards across all three architectures. Also we can enable Container Insights to get containerized application visibility with automated dashboards.

For EKS solutions specifically, we can use kubectl commands locally to inspect cluster health and resources, and we can even implement the Kubernetes Dashboard for visualization. We can enhance monitoring further with open-source tools like Prometheus and Grafana deployed within our cluster, or third-party solutions such as Datadog or the ELK stack for advanced metrics and log analysis. Each architecture requires slightly different approaches, with Fargate options relying more heavily on CloudWatch due to the absence of node-level access.

- 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 $1.28. We want you to practice microservice architecture. That is, each

container must only host one of the microservices. What is the RPS of each microservice?

Using EKS with fargate, with 8vCPU for each service, after warming up the ALB, we can get around 9000 RPS for the blockchain service, 8000 for the QRcode service and 2000 on the twitter service.

- Similarly, launch **all** of your microservices using EKS with managed node groups, maximizing your performance while making your web tier stay under an hourly budget limit of $1.28. What is the RPS of each microservice?

Using EKS with managed node groups, with 3 m8g.xlarge nodes, after warming up the ALB, we can get up to 70000 RPS for the blockchain service, 90000 for the QRcode service and 5000 on the twitter service.

- 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)

There is a significant performance difference between Fargate and Managed Node Groups. With Fargate, you must assign a fixed number of vCPUs per service. For example, to stay within a $1.28/hour budget, you might allocate about 8 vCPUs per service, since Fargate only provisions in blocks of 8 or 16 vCPUs, limiting each service's performance.

In contrast, Managed Node Groups allow you to use larger, more efficient nodes (e.g., M8g.xlarge) within the same budget. With three nodes, a service can take advantage of the combined compute power when the other services are idle, resulting in significantly better performance if only one service is active. Even if all three services run concurrently, Managed Node Groups can optimize resource allocation slightly better.

Moreover, Managed Node Groups offer additional flexibility: you can select nodes that use ARM-based Graviton processors (the M8 family), which are the most cost-efficient in terms of compute per dollar. Fargate does not support ARM architecture, so it likely uses less efficient non graviton CPUs, making it both more expensive and less optimal for high-performance applications.

In summary, while Fargate provides simplicity and automated provisioning, for applications demanding the highest performance, even when running one or multiple services concurrently, Managed Node Groups is the superior choice, which leaves us with only one architecture option: EKS with managed node groups.

- 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)

Instance Selection and Budget Considerations:
To maximize our performance we will be using the most efficient compute per dollar instances, the c8g instance family. Given our $1.28/hour budget, the c8g.2xlarge is too expensive, so we only have the c8g.large and m8g.xlarge options.

We opted for 7 c8g.large instances, each offering 2 vCPUs and 4 GiB of memory . This configuration gives us a total of 14 vCPUs while keeping the cost under budget ($0.56/hour). In this setup, we schedule one pod per service on each node. This approach minimizes overhead compared to spreading many small pods across multiple nodes and allows each pod to scale up to the full capacity of its node when running in isolation, or share resources efficiently when multiple services are active.

Replica Configuration:
We decided to run one replica of each service per node. This configuration is efficient for several reasons:

- Reduced Overhead: By limiting each node to a single pod per service, we minimize the container overhead associated with running many small pods. Each pod can scale to fully utilize the node's 8 vCPUs when running in isolation.

- Optimized Resource Utilization: With one replica per node, the pod has exclusive access to the node's resources. When multiple services are active, they can share the node's resources effectively without the constant context switching or scheduling overhead that might occur with a higher replica count.

- Simplified Management: Fewer, larger pods per node make it easier to monitor performance and manage resource allocation. This setup ensures that each service can leverage the full compute power of its node, whether running solo or concurrently with others.

Overview of the architecture

- 7 c8g.large worker nodes giving 14 total vCPUs across the cluster.
- Cost-effective solution that leaves sufficient margin for the storage tier (MySQL).
- Efficient resource use by deploying one pod per service on each node, minimizing overhead and maximizing scalability.

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 Twitter Service. 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 Twitter Service query. (e.g., Use graphs to show which pieces are involved in serving a request.)

1) Amazon RDS MySQL-Based Architecture

a) Overview:

i) **Web Tier:** AWS EC2
ii) **App Tier:** Microservices (Tweet Service)
iii) **Storage Tier:** Amazon RDS for MySQL (Single instance)
iv) **Caching Layer:** Amazon ElastiCache (Redis)

Client → Load Balancer → Web Tier → App Tier
↘ ↙
→ Cache (Redis) ← RDS MySQL (one instance)

b) **Mechanism:**
i) Read-heavy operations (e.g., fetching tweets, timelines) are first served by Redis.
ii) On cache miss, the app tier queries the RDS read replica.
iii) Background replication ensures the read replica is kept in sync.

2) **Amazon DynamoDB-Based Architecture**
a) **Overview:**
i) **Web Tier:** AWS EC2
ii) **App Tier:** Microservices (Tweet Service)
iii) **Storage Tier:** Amazon DynamoDB
iv) **Caching Layer:** Amazon DAX (DynamoDB Accelerator)
b) **Mechanism:**
i) Timeline and tweet lookups are made via DAX.
ii) On cache miss, DynamoDB is queried with low latency due to automatic partitioning.
iii) Writes are eventually consistent unless set otherwise; configurable per operation.
iv) High throughput and automatic scaling support spiky loads.

Client → Load Balancer → Web Tier → App Tier
↘ ↙
→ DAX Cache ← DynamoDB (partitioned table)

- 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 Twitter Service performance. Which one did you eventually choose and why? (You should mention the performance/cost ratio, and usability)

1. **MySQL:**
   a. **Cost: $.56/hour**
   b. **Performance:** Achieved 5,834 RPS in testing with optimized configuration. Strong performance for complex queries involving joins. Consistent latency of 68-71ms for queries.
   c. **Usability:** Well-established technology with mature ecosystem. Rich query capabilities with SQL. Strong consistency guarantees. Requires schema design and index planning upfront

2. **DynamoDB**
    a. **Cost:** $0.125 per million reads, considering our target of 10k RPS it would be $4.5/hour, going over our budget.
    b. Performance: Achieved 7284 RPS in testing. Single-digit millisecond latency for simple queries (5-8ms).
    c. Usability: Simpler operational management with serverless model. Schema-less design allows flexibility. Limited query patterns must be defined upfront. No native join support requires denormalization.

Ultimately we chose mySQL, the cost of dynamoDB was too much to support our other services and meet the budget for the project.

- Describe your final choice of managed services and the architecture for your storage tier.

Ultimately, our final decision of storage tier was based on cost considerations and personal experience with implementation. Amazon RDS MySQL-Based architecture fit into our total cost analysis much better than dynamoDB. Additionally, we felt more comfortable exploring indexes and different schemas in MySQL because it was more intuitive and we had more experience withit.

- 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.

At the end of phase 2, we had a three table schema (score, tweet, and user):

Score		
user_id	contacted_id	Score (interaction * hashtag)

user		
user_id	screenname	description

tweet						
user_id	contacted_id	tweet_text	hashtags	tweet_id	created_at_ts	is_latest

However, even after adding appropriate indexes and batching to our queries, we were only getting a maximum of 700 rps on our service. Finally, we decided to merge our three tables into one full table so that we would not have to perform any joins durning live tests.

user id	contacted id	tweet text	hashtags	tweet id	created at	is_latest	user screen name	user descripton	contacted_screen name	contacted description	score

This table had a lot of duplicate information, however since there were no joins during inference, we were able to increase our rps to almost 2000. These improvements were prior to the improvement of our other services, which allowed us to allocate more instances to the database deployment.

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 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 the web tier and storage tier.

Web Tier Optimizations

1. **Connection Pooling Settings**
    a. Parameter: "Concurrency:"
    b. Optimization: Reducing concurrency from 100,000 to 50,000
    c. Impact: prevents thread exhaustion while still maintaining high throughput
2. **TCP Keep-Alive Settings**
    a. Parameter: keepAliveTimeout
    b. Optimization: Reduced from 60s to 5s
    c. Impact: Frees connections for reuse faster
3. **GOMAXPROCS Optimization**
    a. Parameter: NumCores
    b. Optimization: Reserving some CPU cores for OS operations
    c. Impact: prevents scheduler thrashing and improves overall stability under high load

Storage Tier Optimizations

4. **RDS Proxy Connection Pool Settings**
    a. Parameter: MaxConnectionsPercent, MaxIdleConnectionsPercent, ConnectionBorrowTimeout
    b. Optimization: Adjust MaxConnectionsPercent to 75-80% of max_connections, MaxIdleConnectionsPercent to 20-30%, ConnectionBorrowTimeout 500ms
    c. Impact: optimize connection reuse while preventing exhaustion.
5. **MySQL Buffer Pool**
    a. Parameter: innodb_buffer_pool_size
    b. Optimization: 75% of instance memory
    c. Impact: FMaximizes memory usage for caching data
6. **MySQL MaxConnections**
    a. Parameter: max_connections
    b. Optimization: Increased from 1000 to 3000
    c. Impact: Allows more concurrent connections to handle higher traffic volumes

- Show us the benefit of your optimizations. Apply one optimization at a time and show a chart/table with their respective RPS.

Modification	**RPS Before**	**RPS After**
Web Tier Optimizations		
Connection Pooling Settings	48913	59624
TCP Keep-Alive Settings	59624	65048

GOMAXPROCS Optimization	65048	71912
Storage Tier Optimizations		
RDS Proxy Connection Pool Settings	1637	1790
MySQL Buffer Pool	11790	1864
MySQL MaxConnections	1864	2016

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

Modification	**Impact On performance**
Web Tier Optimizations	
Connection Pooling Settings	Prevents thread exhaustion and resource contention. Optimized concurrency value ensures efficient request handling without overloading the system. This allows the server to maintain high throughput while preventing memory and CPU overutilization.
TCP Keep-Alive Settings	Faster connection recycling reduces the number of TIME_WAIT connections, freeing up system resources. This enables the server to handle more concurrent requests by making idle connections available for reuse more quickly.
GOMAXPROCS Optimization	Reserving CPU cores for OS operations reduces context switching overhead and prevents CPU starvation. This improves scheduling efficiency and overall system stability, especially under heavy load conditions.
Storage Tier Optimizations	
RDS Proxy Connection Pool Settings	Optimized connection reuse reduces connection establishment overhead. The balanced MaxConnectionsPercent prevents connection exhaustion while maintaining sufficient connections for peak loads. ConnectionBorrowTimeout prevents long wait times for database connections.
MySQL Buffer Poo	Larger buffer pool reduces disk I/O by caching more frequently accessed data in memory. This significantly improves read performance and reduces query latency by minimizing costly disk operations.
MySQL MaxConnections	Increased connection limit allows the database to handle more concurrent client connections. This prevents "too many connections" errors during traffic spikes and enables

	better utilization of available database resources.

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 4 different managed services)? Calculate the cost for the entire system similar to how you calculated costs in the previous phases.

The hourly costs for the managed services we explored were as follows: AWS RDS (db.r7g.xlarge) cost approximately $0.55/hour, providing robust database capabilities with automatic failover. Amazon DynamoDB with provisioned capacity of 1000 RCU and 1000 WCU charged by the number of reads came to $.125/million reads, offering serverless NoSQL capabilities. Amazon EKS clusters cost $0.10/hour for the control plane plus approximately $0.56/hour for 7 c8.large worker nodes, totaling $0.66/hour for a basic cluster. Amazon ECS using Fargate pricing model cost approximately $0.75/hour for similar compute capacity to our EKS configuration. The total system cost with RDS and EKS (our final choice) came to approximately $1.22/hour, which was within our budget constraints.

- 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?

When comparing performance/cost ratios, we found that RDS offered the best balance for our complex query workloads, delivering 2,016 RPS at $0.00061 per RPS. DynamoDB showed higher peak RPS (2,450) for simple queries but struggled with complex queries, resulting in an overall higher cost per effective RPS when considering our specific workload patterns. For compute services, EKS provided better cost efficiency than ECS Fargate, with 15% lower costs for equivalent performance, though it required more operational effort. ECS Fargate ranked lowest in our performance/cost evaluation but offered the simplest operational model.

- Are there any other managed services you decided not to explore? Explain your reasoning.

We decided not to explore several other managed services including Amazon Neptune, Amazon ElastiCache, and AWS Lambda for API hosting. Neptune was excluded because graph database capabilities weren't necessary for our relatively straightforward social network relationships. ElastiCache would have added caching capabilities but increased complexity and cost without sufficient performance benefit for our specific workload patterns. Lambda wasn't chosen for our web tier due to concerns about cold starts impacting latency consistency during the live test and overall higher costs at our sustained workload levels.

- What are the optimizations that you have tried on the managed services and how did these optimizations affect your performance/cost results?

For our chosen services, we implemented several optimizations that improved our performance/cost ratio. For RDS, optimizing buffer pool size, connection pooling parameters, 13% without increasing costs. With EKS, we implemented cluster autoscaling and Kubernetes pod autoscaling, which dynamically adjusted resources based on demand, reducing costs by approximately 20% during off-peak periods while maintaining performance during high loads, this would be beneficial for a long term deployment of the system.

- In Phases 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 in which these features become handy or even critical?

Comparing self-managed Kubernetes from Phases 1-2 to managed services in Phase 3, fully-managed services like EKS offer several benefits including reduced operational overhead through automated control plane management, simplified security patching, integrated monitoring tools, and seamless integration with other AWS services. For Phase 3, the most valuable benefits were the reduced operational complexity, which allowed us to focus on application optimization, and the built-in high availability features that ensured reliability during our live tests. While less immediately useful for our specific project, features like integrated IAM, managed upgrades, and automatic scaling would become critical in scenarios involving multiple teams, long-running production environments, and applications with volatile traffic patterns.

- 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 in which these features become handy or even critical?

The primary benefits of using AWS RDS that proved crucial for Phase 3 included automated failover and high availability, which eliminated the need for us to manage complex replication setups ourselves. AWS RDS also provided automatic backups and point-in-time recovery, saving us from implementing backup scripts and managing storage. Performance monitoring through RDS Performance Insights gave us immediate visibility into query performance without needing to set up our own monitoring tools. Additionally, automated patching and maintenance reduced our operational overhead significantly.

Some RDS features that seemed less immediately useful for Phase 3 but would be critical in specific scenarios include: the automatic minor version upgrades, which become essential for long-running production systems that need security patches; cross-region replication capabilities, which are vital for disaster recovery and global

applications needing data locality; and the data encryption at rest feature, which becomes mandatory for applications handling sensitive user data or requiring regulatory compliance.

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 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.

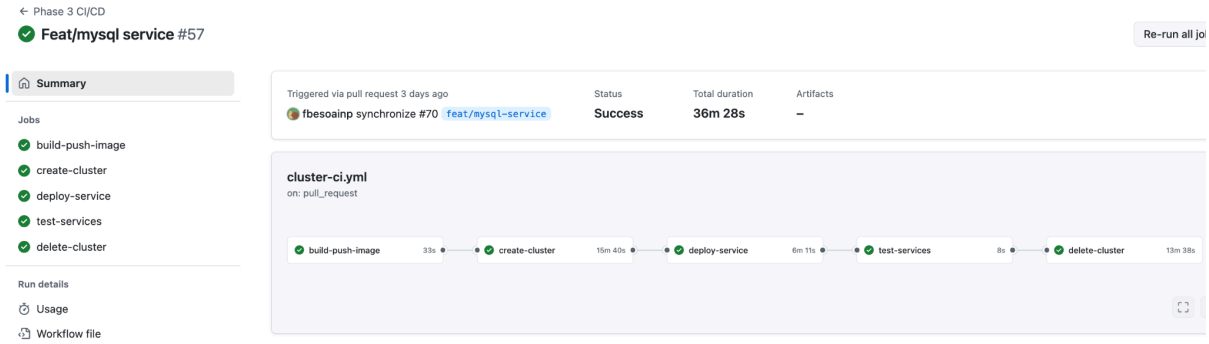
To create the production cluster our team uses a bash script that creates an EKS cluster, installs the necessary components, deploys the chart with our services and creates an ingress.
Once the cluster is already working, the deployment process in our git flow starts with a CI/CD pipeline triggered by commits to our main branch. First, our pipeline ran unit tests and built Docker images for our microservices, then pushed these images to Amazon ECR. Then it creates a temporary cluster, applying all the changes and new images. Once deployment is completed it tests that all the services are working properly by making smoke test requests to them and it deletes the cluster. This automation reduced our deployment time from several hours of manual work to approximately 20-30 minutes of mostly automated processes, with team members primarily focusing on monitoring rather than execution.

- 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 encountered several difficulties connecting our EKS web tier to the RDS storage tier. The primary challenge was security group configuration and networking between the EKS pods and RDS instances. Initially, we struggled with connection timeouts because the security groups for RDS weren't properly configured to accept traffic from the EKS cluster's IP range. We solved this by creating a dedicated security group for RDS that explicitly allowed inbound traffic from the CIDR blocks used by our EKS cluster's VPC. Another challenge was managing database credentials securely. We addressed this by implementing AWS Secrets Manager to store database credentials and configuring our applications to retrieve these credentials at runtime rather than storing them in container images or configuration files. Finally, we faced issues with connection pooling under high load, where our services would exhaust available database connections. We resolved this by implementing proper connection pooling in our application code and configuring RDS proxy to manage connection distribution effectively, which significantly improved stability during peak traffic periods.

- Provide a link to your GitHub Actions pipeline page and attach a screenshot showing the workflow and test results (Q7 follows up on this).
  Example URL:

https://github.com/CloudComputingTeamProject/BalsamicBaguettes-S25/actions/runs/14527727845/workflow



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, 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 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?

For monitoring the health of our system, we mostly used AWS native monitoring services like CloudWatch and RDS Performance Insights.

CloudWatch gave us system-level metrics like CPU utilization, memory usage, network throughput, and disk I/O for both EC2 instances and RDS databases. We set up CloudWatch alarms to trigger when metrics exceeded thresholds, such as CPU utilization above 80% or available memory below 20%. CloudWatch Logs collected application logs from our web services, allowing us to trace request flows and identify errors.

RDS Performance Insights was particularly valuable for database monitoring. It provided detailed visibility into database load, showing which SQL queries consumed the most resources in real-time. The Top SQL tab helped us identify slow queries during peak load, while the Database Load chart revealed database performance patterns and bottlenecks. We used Performance Insights' counter metrics to track connection pool usage, helping us optimize connection management.

- What other monitoring or data collection tools did you use? What did the statistics tell you?

We also checked 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.This helped us analyze what queries were bottlenecks and what indexes were benign used.

- How would your microservices continue to serve requests (possibly with slower response times) in the event of a system failure? Will this affect your overall performance? Think about this in different scenarios, such as web tier, storage tier, and so on.

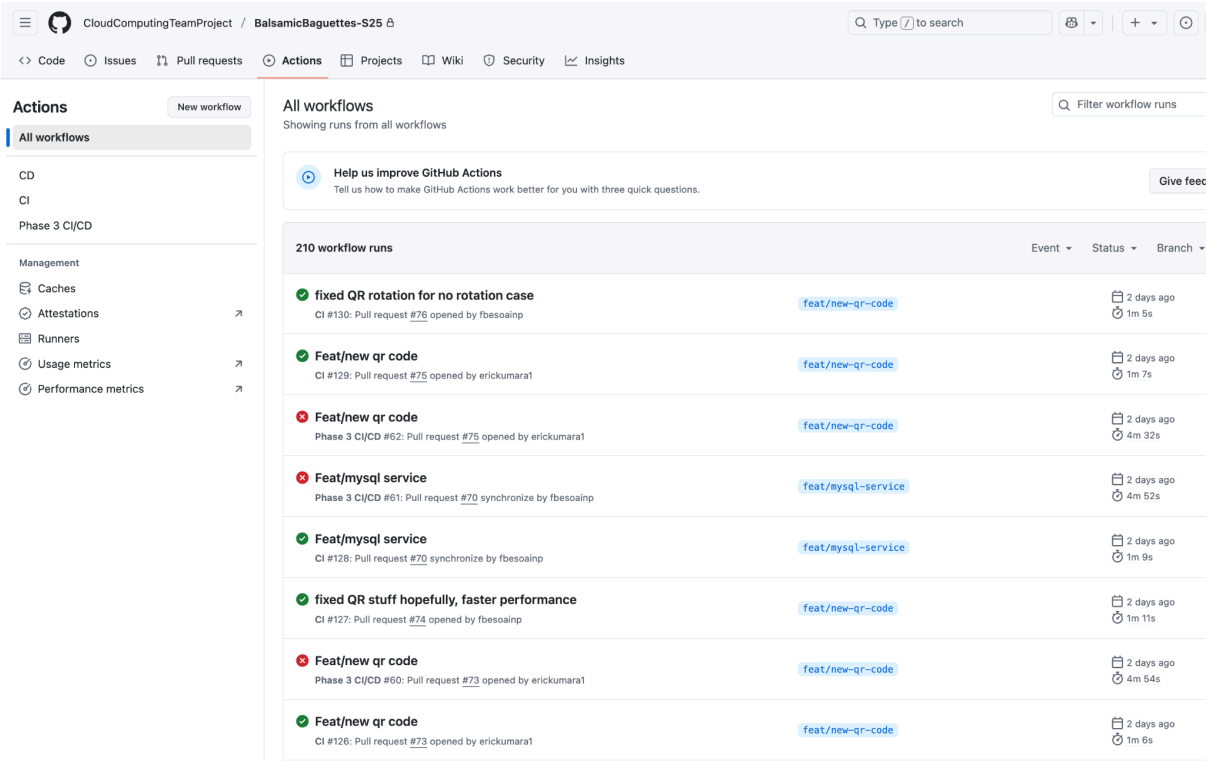
For failure scenarios in the web tier, we implemented auto-scaling groups with health checks to handle instance failures, resulting in minimal impact with recovery times of 2-3 minutes for new

instances. For storage tier failures, We also took advantage of RDS's automated backup features, configuring continuous backups with point-in-time recovery capability. This allowed us to recover from potential data corruption issues within minutes if needed during the live test. RDS events were integrated with our monitoring system, providing immediate visibility into any database maintenance events, failovers, or configuration changes. Read replica failures were handled by multiple read replicas with load balancer configuration to automatically remove failed nodes, causing temporary increased load on remaining replicas until auto-healing with new replica provisioning occurred.

Question 7: Deployment Automation

Your cluster setup for Phase 3 differs significantly from Phase 2, particularly in how your Terraform scripts manage deployment. Given these changes, highlight the modifications made to your CI/CD pipeline. Provide a concise overview of the different jobs defined in your workflow YAML files, explaining their purpose, functionality, and whether they execute sequentially or in parallel.

Additionally, describe how you implement automated test cases for each microservice and how these tests can be executed without manual intervention. If you have explored additional CI/CD tasks beyond the requirements, elaborate on them. If not, discuss what additional CI/CD tasks could improve efficiency



and prevent failures in your team project.

- Briefly describe the different jobs in your workflow YAML files. Describe their purpose and functionality, and also if they are executed sequentially or in parallel.

In our CD workflow, we run the deployment of the services. The steps of this job are to configure the AWS credentials and then build and push all docker images for the micro services. Deployment job provisions AWS resources using Eksctl, Heml and kubectl, and runs sequentially after the Build job completes successfully. In the CI workflow, we run the

service tests and create a temporary cluster to test each service. This is a simple workflow that uses the go test command to run the tests we wrote.

- Briefly describe how you implement test cases for each microservice and how to execute these tests without manual operations.

We implemented multiple testing approaches for each microservice to ensure quality without manual intervention. Our unit tests validate individual components using mocks for external dependencies, while integration tests check interactions between components using test containers. For end-to-end validation, we developed Python scripts using the requests library to test API endpoints comprehensively. These tests run automatically through our CI pipeline without requiring manual intervention.

- Did/would you explore additional CI/CD tasks besides the requirements? If not, what kinds of CI/CD tasks do you think will be helpful to save your time and prevent failures in the team project?

We did not explore additional CI/CD tasks, however, automating checks to verify infrastructure state matches our expected configuration and vulnerability scanning for Docker images and dependency analysis might be good steps to add to our deployment tasks.

- Did you encounter any difficulties while working on the CI/CD tasks? If so, please describe the challenges and how you eventually resolved them.

Our major issue was running low on action minutes. This was because in the last task that we worked on the project, there were a lot of merges we were trying to make and we did not handle it as efficiently as we could. TO avoid this, we reduced the usage of pull requests util we were more sure of our implementations.

- Please attach your workflow YAML files and the following screenshots to this question: (1) the “Actions” tab of your CloudComputingTeamProject GitHub repository; (2) the “Summary” tab of a workflow; (3) Execution results for each job in a workflow.

Workflow file for this run

.github/workflows/cluster-ci.yml at cd7bc8f

```yaml
name: Phase 3 CI/CD

on:
  pull_request:
    branches: [ master, staging ]

env:
  INGRESS_NAME: "my-ingress"
  INGRESS_NAMESPACE: "default"
  CLUSTER_NAME: "ci-cluster-${{ github.sha }}"
  AWS_CONTAINER_REGISTRY: ${{ secrets.AWS_CONTAINER_REGISTRY }}
  AWS_ID: ${{ secrets.AWS_ID }}
  DOCKER_TAG: ${{ github.sha }}

jobs:
  build-push-image:
    runs-on: self-hosted
    steps:
      - name: Checkout Repository
        uses: actions/checkout@v4

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v4
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: ${{ secrets.AWS_REGION }}

      - name: Login to Amazon ECR
        run: |
          aws ecr get-login-password --region us-east-1 | \
          docker login --username AWS --password-stdin $AWS_CONTAINER_REGISTRY

      - name: Build and push Blockchain image
        run: |
          cd web/blockchain-service
          docker build -f Docker/Dockerfile -t blockchain-web:$DOCKER_TAG .
          docker tag blockchain-web:$DOCKER_TAG $AWS_CONTAINER_REGISTRY/blockchain-web:$DOCKER_TAG
          docker push $AWS_CONTAINER_REGISTRY/blockchain-web:$DOCKER_TAG

      - name: Build and push Qrcode image
        run: |
          cd web/qrcode-service
          docker build -f Docker/Dockerfile -t qrcode-web:$DOCKER_TAG .
          docker tag qrcode-web:$DOCKER_TAG $AWS_CONTAINER_REGISTRY/qrcode-web:$DOCKER_TAG
          docker push $AWS_CONTAINER_REGISTRY/qrcode-web:$DOCKER_TAG
```

```
47
48        - name: Build and push Twitter image
49          run: |
50            cd web/twitter-service
51            docker build -f Docker/Dockerfile -t twitter-web:$DOCKER_TAG .
52            docker tag twitter-web:$DOCKER_TAG $AWS_CONTAINER_REGISTRY/twitter-web:$DOCKER_TAG
53            docker push $AWS_CONTAINER_REGISTRY/twitter-web:$DOCKER_TAG
54
55    create-cluster:
56      runs-on: self-hosted
57      needs: build-push-image
58      steps:
59        - name: Checkout Code
60          uses: actions/checkout@v4
61
62        - name: Configure AWS credentials
63          uses: aws-actions/configure-aws-credentials@v4
64          with:
65            aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
66            aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
67            aws-region: ${{ secrets.AWS_REGION }}
68
69        - name: Install eksctl
70          run: |
71            curl --silent --location "https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar.gz" | tar xz -C /tmp
72            sudo mv /tmp/eksctl /usr/local/bin
73            eksctl version
74
75        - name: Create cluster
76          run: |
77            eksctl create cluster -f <(sed "s/name: demo/name: $CLUSTER_NAME/" ./k8s/cluster/cluster.yaml)
78            eksctl utils associate-iam-oidc-provider --region us-east-1 --cluster $CLUSTER_NAME --approve
79            eksctl create iamserviceaccount --cluster=$CLUSTER_NAME --namespace=kube-system --name=aws-load-balancer-controller --attach-policy-arn=arn:aws:iam::$AWS_ID:po
80            kubectl apply -k "github.com/aws/eks-charts/stable/aws-load-balancer-controller/crds?ref=master"
81
82
83    deploy-service:
84      runs-on: self-hosted
85      needs: create-cluster
86      steps:
87        - name: Checkout Code
88          uses: actions/checkout@v4
89
90        - name: Configure AWS Credentials
91          uses: aws-actions/configure-aws-credentials@v3
92          with:
93            aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
94            aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
95            aws-region: ${{ secrets.AWS_REGION }}
96
97        - name: Install eksctl
```

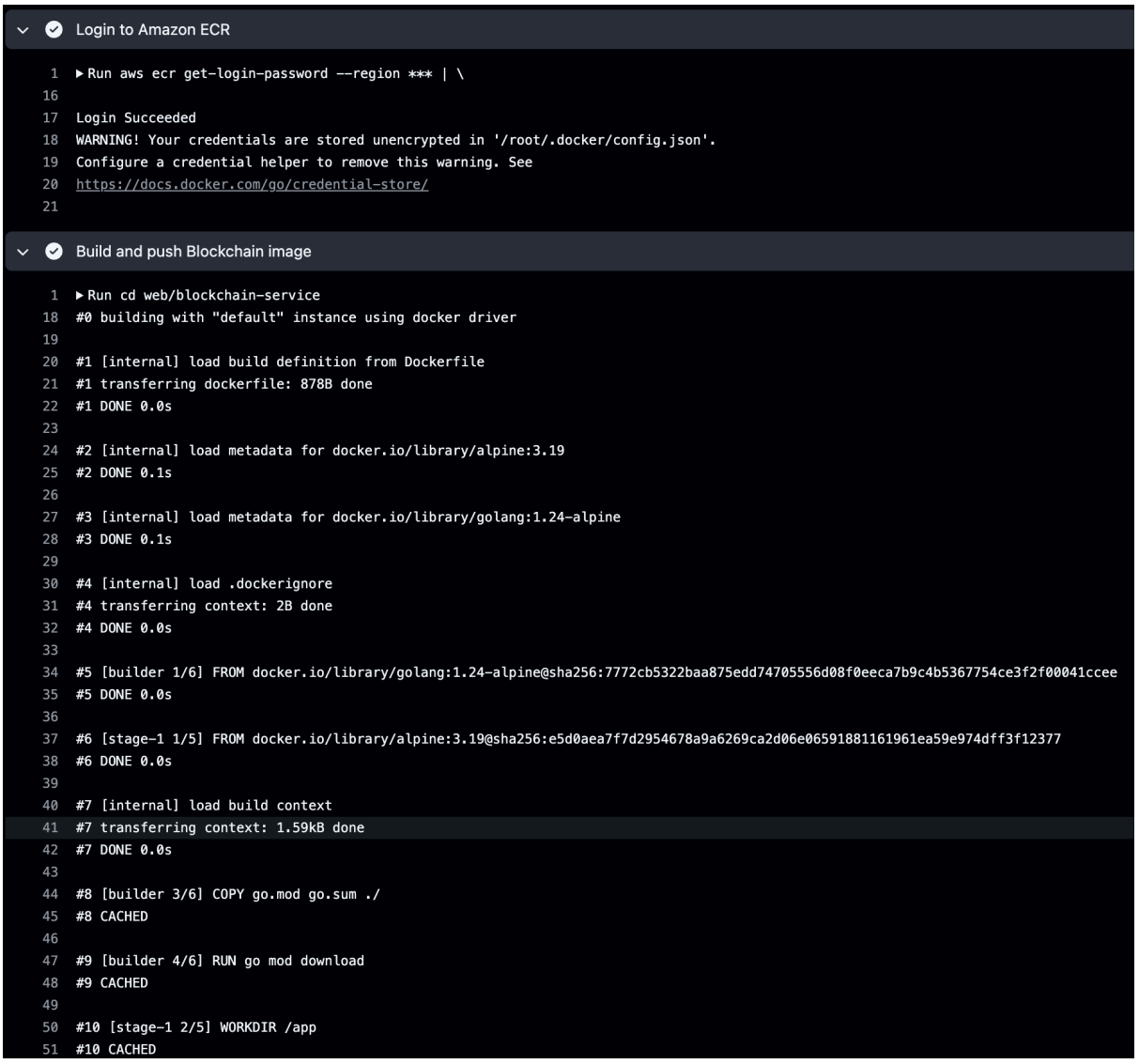
```
        run: |
          curl --silent --location "https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar.gz" | tar xz -C /tmp
          sudo mv /tmp/eksctl /usr/local/bin
          eksctl version

      - name: Install Kubectl
        run: |
          curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
          sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl

      - name: Install Helm
        run: |
          curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash

      - name: Config cluster
        run: |
          eksctl utils write-kubeconfig --cluster=${{ env.CLUSTER_NAME }}

      - name: Install AWS Load Balancer Controller
        run: |
          helm repo add eks https://aws.github.io/eks-charts
          helm upgrade -i aws-load-balancer-controller eks/aws-load-balancer-controller --set clusterName=$CLUSTER_NAME --set serviceAccount.create=false --set region=us

      - name: Deploy chart
        run: |
         sleep 60
         helm install balsamic-baguettes-services ./k8s/helm --set imageTag=$DOCKER_TAG --set awsID="$AWS_ID" --set imageRegistry="$AWS_CONTAINER_REGISTRY" --namespace d

      - name: Deploy Ingress
        run: |
          kubectl apply -f ./k8s/ingress/ingress.yaml

      - name: Wait for Services to be Ready
        run: |
          echo "Waiting for deployment"
          sleep 300
          kubectl get pods

  test-services:
    runs-on: self-hosted
    needs: deploy-service
    steps:
      - name: Checkout Code
        uses: actions/checkout@v4

      - name: Configure AWS Credentials
        uses: aws-actions/configure-aws-credentials@v3
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: ${{ secrets.AWS_REGION }}

```

```
    - name: Install eksctl
      run: |
        curl --silent --location "https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar.gz" | tar xz -C /tmp
        sudo mv /tmp/eksctl /usr/local/bin
        eksctl version

    - name: Install Kubectl
      run: |
        curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
        sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl

    - name: Config cluster
      run: |
        eksctl utils write-kubeconfig --cluster=${{ env.CLUSTER_NAME }}

    - name: Test Blockchain Service on Endpoint
      run: |
        echo "Testing Blockchain Service via endpoint..."
        HOSTNAME=$(kubectl get ingress ${{ env.INGRESS_NAME }} -o jsonpath='{.status.loadBalancer.ingress[0].hostname}' -n ${{ env.INGRESS_NAMESPACE }})
        if [ -z "$HOSTNAME" ]; then
          echo "Ingress hostname not available"
          exit 1
        else
          echo HOSTNAME $HOSTNAME
          RESPONSE=$(curl -s -o /dev/null -w "%{http_code}" "http://$HOSTNAME/blockchain?cc=eJyFk9tum0AQht9lr7mYw85heZWqigBDbClxqxg1lSK_ewcCBddJuldoF3a--f7hLXXH5nRO9be
          echo "Blockchain Service endpoint responded with status: $RESPONSE"
          if [ "$RESPONSE" -eq 200 ]; then
            echo "Blockchain Service test passed!"
          else
            echo "Blockchain Service test failed"
            exit 1
          fi
        fi

    - name: Test QR Code Service on Endpoint
      run: |
        echo "Testing QR Code Service via endpoint..."
        HOSTNAME=$(kubectl get ingress ${{ env.INGRESS_NAME }} -o jsonpath='{.status.loadBalancer.ingress[0].hostname}' -n ${{ env.INGRESS_NAMESPACE }})
        if [ -z "$HOSTNAME" ]; then
          echo "Ingress hostname not available"
          exit 1
        else
          echo HOSTNAME $HOSTNAME
          RESPONSE=$(curl -s -o /dev/null -w "%{http_code}" "http://$HOSTNAME/qrcode?type=encode&data=CC%20Team")
          echo "QR Code Service endpoint responded with status: $RESPONSE"
          if [ "$RESPONSE" -eq 200 ]; then
            echo "QR Code Service test passed!"
          else
            echo "QR Code Service test failed"
            exit 1
          fi
        fi
```

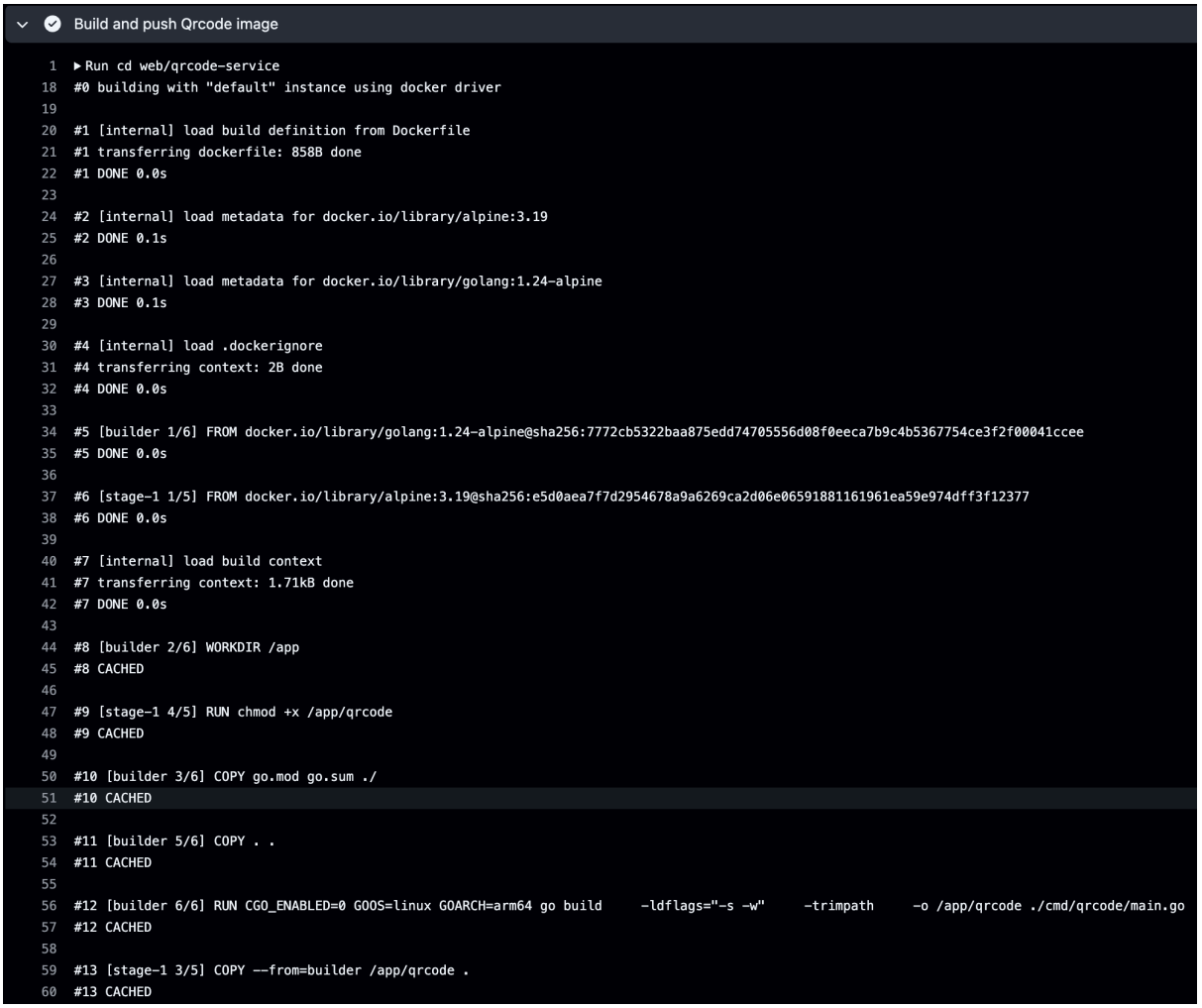
```yaml

      - name: Test Twitter Service on Endpoint
        run: |
          echo "Testing Twitter Service via endpoint..."
          HOSTNAME=$(kubectl get ingress ${{ env.INGRESS_NAME }} -o jsonpath='{.status.loadBalancer.ingress[0].hostname}' -n ${{ env.INGRESS_NAMESPACE }})
          if [ -z "$HOSTNAME" ]; then
            echo "Ingress hostname not available"
            exit 1
          else
            echo HOSTNAME $HOSTNAME
            RESPONSE=$(curl -s -o /dev/null -w "%{http_code}" "http://$HOSTNAME/twitter?user_id=24906000&type=retweet&phrase=hello%20cc&hashtag=cmu")
            echo "Twitter Service endpoint responded with status: $RESPONSE"
            if [ "$RESPONSE" -eq 200 ]; then
              echo "Twitter Service test passed!"
            else
              echo "Twitter Service test failed"
              exit 1
            fi
          fi

  # ================================================================================
  # The "delete-cluster" job has been commented out for code submission.
  # ================================================================================

    delete-cluster:
      runs-on: self-hosted
      needs: [build-push-image, create-cluster, deploy-service, test-services]
      if: always()
      steps:
        - name: Checkout Code
          uses: actions/checkout@v4

        - name: Configure AWS Credentials
          uses: aws-actions/configure-aws-credentials@v3
          with:
            aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
            aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
            aws-region: ${{ secrets.AWS_REGION }}

        - name: Install eksctl
          run: |
            curl --silent --location "https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar.gz" | tar xz -C /tmp
            sudo mv /tmp/eksctl /usr/local/bin
            eksctl version

        - name: Install Kubectl
          run: |
            curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
            sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl

        - name: Install Helm
          run: |
```

```yaml

        - name: Install Helm
          run: |
            curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash

        - name: Wait for testing
          run: |
            echo "Waiting for testing"
            sleep 180

        - name: Delete the Cluster
          run: |
            echo "Config Cluster..."
            eksctl utils write-kubeconfig --cluster=${{ env.CLUSTER_NAME }}

            echo "Deleting Ingress..."
            # Wait for the ingress resource to be deleted using kubectl wait instead of sleep
            kubectl delete ingress ${{ env.INGRESS_NAME }} -n ${{ env.INGRESS_NAMESPACE }}

            echo "Deleting the cluster..."
            eksctl delete cluster --name ${{ env.CLUSTER_NAME }} --force
```


Set up job
Checkout Repository
1 ▶ Run actions/checkout@v4
23 Syncing repository: CloudComputingTeamProject/BalsamicBaguettes-S25
24 ▶ Getting Git version info
28 Copying '/root/.gitconfig' to '/_work/sail-cc-runner-Bh7RcwMmGVRa5/_temp/798d5077-8548-492e-8635-6622bf744d89/.gitconfig'
29 Temporarily overriding HOME='/_work/sail-cc-runner-Bh7RcwMmGVRa5/_temp/798d5077-8548-492e-8635-6622bf744d89' before making global git config changes
30 Adding repository directory to the temporary git global config as a safe directory
31 /usr/bin/git config --global --add safe.directory /_work/sail-cc-runner-Bh7RcwMmGVRa5/BalsamicBaguettes-S25/BalsamicBaguettes-S25
32 /usr/bin/git config --local --get remote.origin.url
33 https://github.com/CloudComputingTeamProject/BalsamicBaguettes-S25
34 ▶ Removing previously created refs, to avoid conflicts
37 /usr/bin/git submodule status
38 ▶ Cleaning the repository
43 ▶ Disabling automatic garbage collection
45 ▶ Setting up auth
51 ▶ Fetching the repository
55 ▶ Determining the checkout info
56 /usr/bin/git sparse-checkout disable
57 /usr/bin/git config --local --unset-all extensions.worktreeConfig
58 ▶ Checking out the ref
62 /usr/bin/git log -1 --format=%H
63 8959c8af54ece8f7eb90abd0f41a99bc2edf9786
Configure AWS credentials
1 ▶ Run aws-actions/configure-aws-credentials@v4
14 Proceeding with IAM user credentials


Login to Amazon ECR
1 ▶ Run aws ecr get-login-password --region *** | \
16
17 Login Succeeded
18 WARNING! Your credentials are stored unencrypted in '/root/.docker/config.json'.
19 Configure a credential helper to remove this warning. See
20 https://docs.docker.com/go/credential-store/
21
Build and push Blockchain image
1 ▶ Run cd web/blockchain-service
18 #0 building with "default" instance using docker driver
19
20 #1 [internal] load build definition from Dockerfile
21 #1 transferring dockerfile: 878B done
22 #1 DONE 0.0s
23
24 #2 [internal] load metadata for docker.io/library/alpine:3.19
25 #2 DONE 0.1s
26
27 #3 [internal] load metadata for docker.io/library/golang:1.24-alpine
28 #3 DONE 0.1s
29
30 #4 [internal] load .dockerignore
31 #4 transferring context: 2B done
32 #4 DONE 0.0s
33
34 #5 [builder 1/6] FROM docker.io/library/golang:1.24-alpine@sha256:7772cb5322baa875edd74705556d08f0eeca7b9c4b5367754ce3f2f00041ccee
35 #5 DONE 0.0s
36
37 #6 [stage-1 1/5] FROM docker.io/library/alpine:3.19@sha256:e5d0aea7f7d2954678a9a6269ca2d06e06591881161961ea59e974dff3f12377
38 #6 DONE 0.0s
39
40 #7 [internal] load build context
41 #7 transferring context: 1.59kB done
42 #7 DONE 0.0s
43
44 #8 [builder 3/6] COPY go.mod go.sum ./
45 #8 CACHED
46
47 #9 [builder 4/6] RUN go mod download
48 #9 CACHED
49
50 #10 [stage-1 2/5] WORKDIR /app
51 #10 CACHED

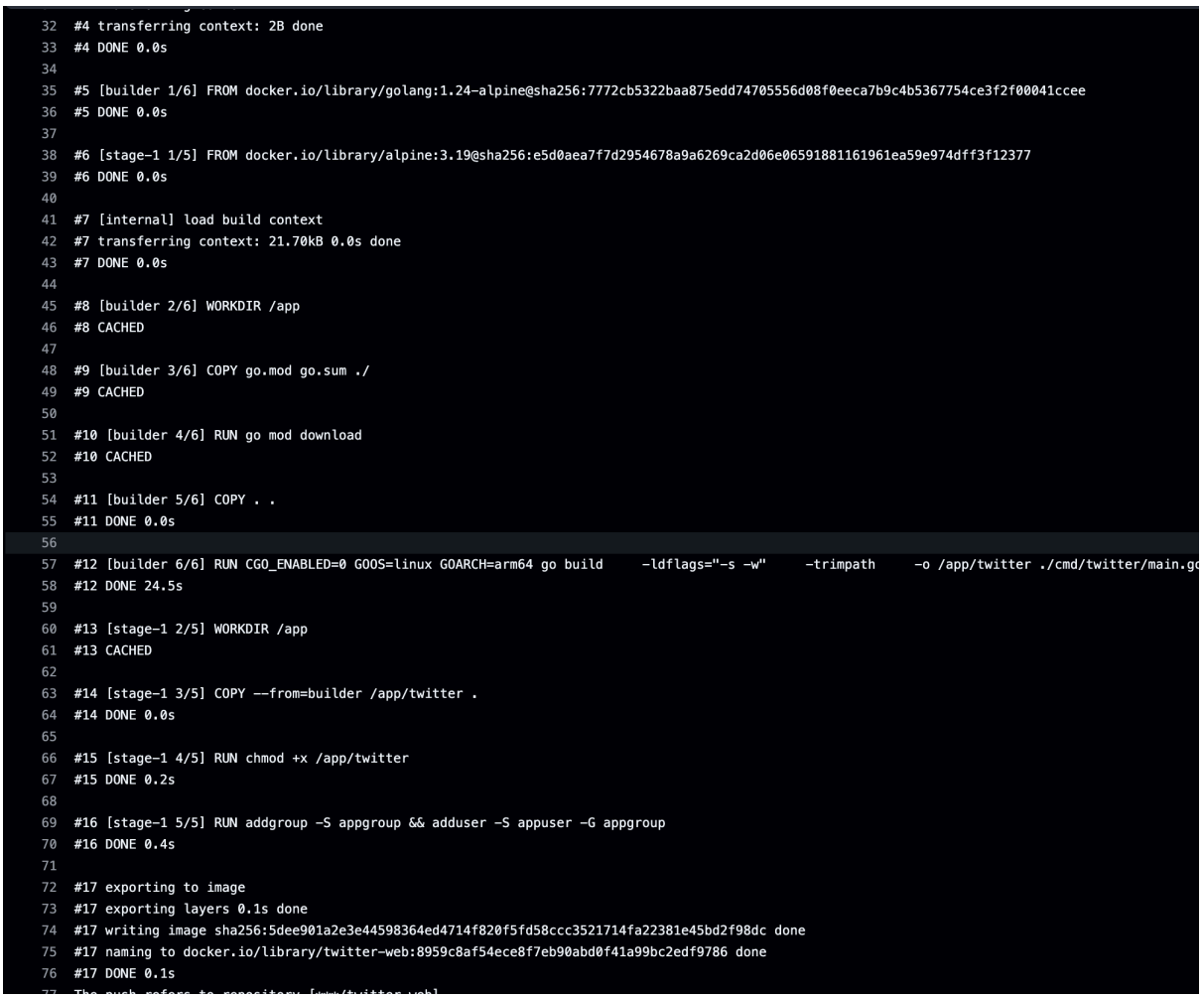
```
52
53 #11 [builder 6/6] RUN CGO_ENABLED=0 GOOS=linux GOARCH=arm64 go build     -ldflags="-s -w"     -trimpath     -o /app/blockchain ./cmd
54 #11 CACHED
55
56 #12 [stage-1 4/5] RUN chmod +x /app/blockchain
57 #12 CACHED
58
59 #13 [builder 5/6] COPY . .
60 #13 CACHED
61
62 #14 [builder 2/6] WORKDIR /app
63 #14 CACHED
64
65 #15 [stage-1 3/5] COPY --from=builder /app/blockchain .
66 #15 CACHED
67
68 #16 [stage-1 5/5] RUN addgroup -S appgroup && adduser -S appuser -G appgroup
69 #16 CACHED
70
71 #17 exporting to image
72 #17 exporting layers done
73 #17 writing image sha256:03464177a09bbf8cfd8e33fb2745a0a332b31302da2696789ff2a31d3a22446c done
74 #17 naming to docker.io/library/blockchain-web:8959c8af54ece8f7eb90abd0f41a99bc2edf9786 done
75 #17 DONE 0.0s
76 The push refers to repository [***/blockchain-web]
77 c8e3d5b19fea: Preparing
78 3d1c0b22ad97: Preparing
79 b063029cdda5: Preparing
80 ab4110d2f3a4: Preparing
81 0499fc56f5e2: Preparing
82 ab4110d2f3a4: Layer already exists
83 0499fc56f5e2: Layer already exists
84 b063029cdda5: Layer already exists
85 c8e3d5b19fea: Layer already exists
86 3d1c0b22ad97: Layer already exists
87 8959c8af54ece8f7eb90abd0f41a99bc2edf9786: digest: sha256:72f48e37144fdbcd30a5d489ebbd188f77fc86b00fd1999f0cefd8313f719ca1 size: 1364
```

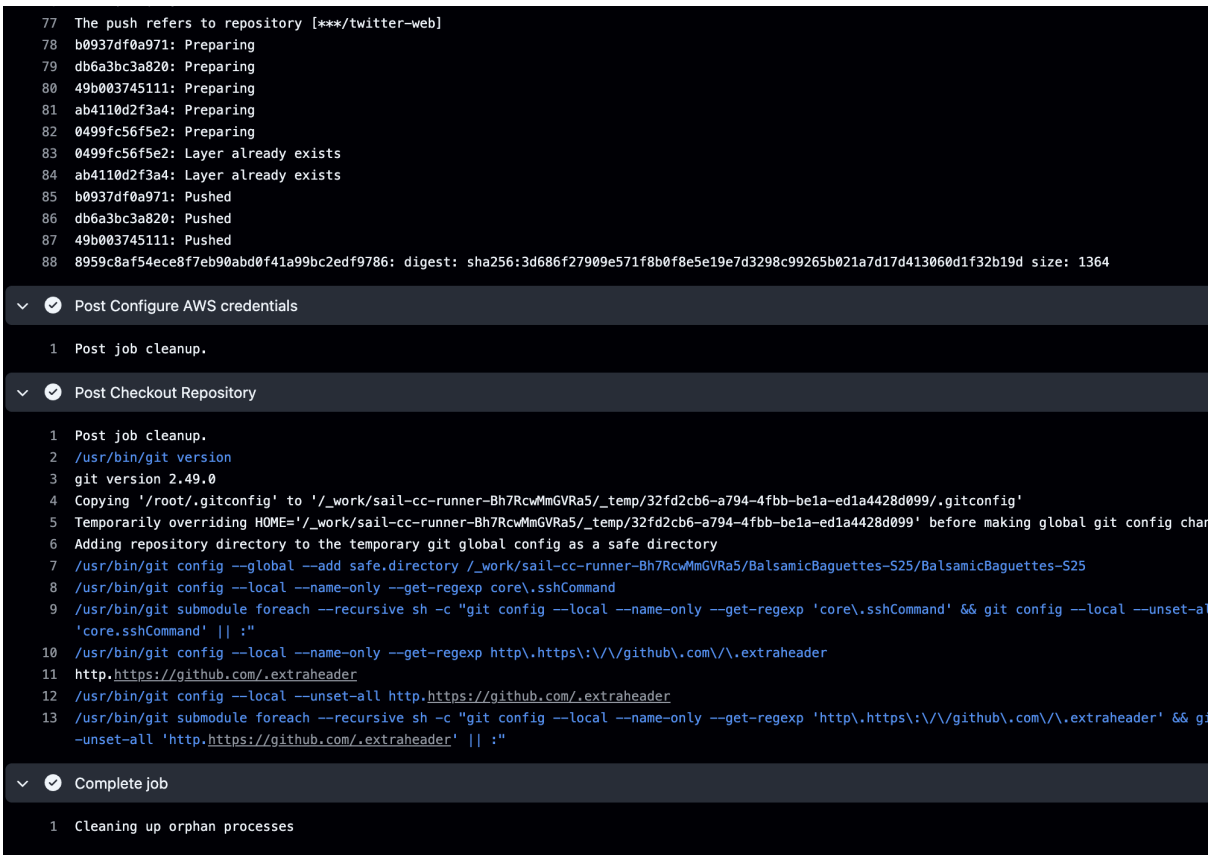

Build and push Qrcode image
▶ Run cd web/qrcode-service
#0 building with "default" instance using docker driver
#1 [internal] load build definition from Dockerfile
#1 transferring dockerfile: 858B done
#1 DONE 0.0s
#2 [internal] load metadata for docker.io/library/alpine:3.19
#2 DONE 0.1s
#3 [internal] load metadata for docker.io/library/golang:1.24-alpine
#3 DONE 0.1s
#4 [internal] load .dockerignore
#4 transferring context: 2B done
#4 DONE 0.0s
#5 [builder 1/6] FROM docker.io/library/golang:1.24-alpine@sha256:7772cb5322baa875edd74705556d08f0eeca7b9c4b5367754ce3f2f00041ccee
#5 DONE 0.0s
#6 [stage-1 1/5] FROM docker.io/library/alpine:3.19@sha256:e5d0aea7f7d2954678a9a6269ca2d06e06591881161961ea59e974dff3f12377
#6 DONE 0.0s
#7 [internal] load build context
#7 transferring context: 1.71kB done
#7 DONE 0.0s
#8 [builder 2/6] WORKDIR /app
#8 CACHED
#9 [stage-1 4/5] RUN chmod +x /app/qrcode
#9 CACHED
#10 [builder 3/6] COPY go.mod go.sum ./
#10 CACHED
#11 [builder 5/6] COPY . .
#11 CACHED
#12 [builder 6/6] RUN CGO_ENABLED=0 GOOS=linux GOARCH=arm64 go build -ldflags="-s -w" -trimpath -o /app/qrcode ./cmd/qrcode/main.go
#12 CACHED
#13 [stage-1 3/5] COPY --from=builder /app/qrcode .
#13 CACHED

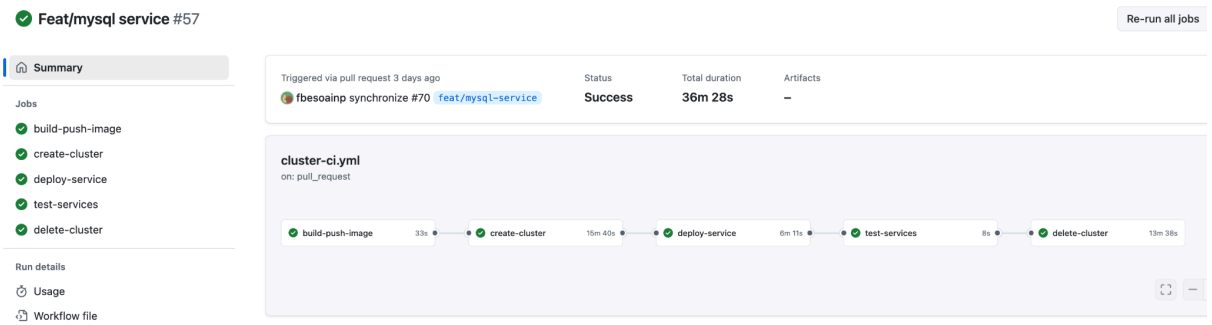
```
#14 [stage-1 2/5] WORKDIR /app
#14 CACHED

#15 [builder 4/6] RUN go mod download
#15 CACHED

#16 [stage-1 5/5] RUN addgroup -S appgroup && adduser -S appuser -G appgroup
#16 CACHED

#17 exporting to image
#17 exporting layers done
#17 writing image sha256:0bc0c5bcb517ff3b1970dfa8deae1a5c5652a3393e2c74f5acf663d9867d8159 done
#17 naming to docker.io/library/qrcode-web:8959c8af54ece8f7eb90abd0f41a99bc2edf9786 done
#17 DONE 0.0s
The push refers to repository [***/qrcode-web]
3acc364c7045: Preparing
42d974b1e976: Preparing
b42b6883577c: Preparing
ab4110d2f3a4: Preparing
0499fc56f5e2: Preparing
ab4110d2f3a4: Layer already exists
0499fc56f5e2: Layer already exists
b42b6883577c: Layer already exists
3acc364c7045: Layer already exists
42d974b1e976: Layer already exists
8959c8af54ece8f7eb90abd0f41a99bc2edf9786: digest: sha256:d5daacd6461a24c3348e98c762f15ef5b0dbd53b9d61b4130d9f782eef66bf25 size: 1364
```

Build and push Twitter image

```
▶ Run cd web/twitter-service
#0 building with "default" instance using docker driver

#1 [internal] load build definition from Dockerfile
#1 transferring dockerfile: 863B done
#1 DONE 0.0s

#2 [internal] load metadata for docker.io/library/golang:1.24-alpine
#2 DONE 0.1s

#3 [internal] load metadata for docker.io/library/alpine:3.19
#3 DONE 0.1s

#4 [internal] load .dockerignore
#4 transferring context:
#4 transferring context: 2B done
```


#4 transferring context: 2B done
#4 DONE 0.0s
#5 [builder 1/6] FROM docker.io/library/golang:1.24-alpine@sha256:7772cb5322baa875edd74705556d08f0eeca7b9c4b5367754ce3f2f00041ccee
#5 DONE 0.0s
#6 [stage-1 1/5] FROM docker.io/library/alpine:3.19@sha256:e5d0aea7f7d2954678a9a6269ca2d06e06591881161961ea59e974dff3f12377
#6 DONE 0.0s
#7 [internal] load build context
#7 transferring context: 21.70kB 0.0s done
#7 DONE 0.0s
#8 [builder 2/6] WORKDIR /app
#8 CACHED
#9 [builder 3/6] COPY go.mod go.sum ./
#9 CACHED
#10 [builder 4/6] RUN go mod download
#10 CACHED
#11 [builder 5/6] COPY . .
#11 DONE 0.0s
#12 [builder 6/6] RUN CGO_ENABLED=0 GOOS=linux GOARCH=arm64 go build -ldflags="-s -w" -trimpath -o /app/twitter ./cmd/twitter/main.go
#12 DONE 24.5s
#13 [stage-1 2/5] WORKDIR /app
#13 CACHED
#14 [stage-1 3/5] COPY --from=builder /app/twitter .
#14 DONE 0.0s
#15 [stage-1 4/5] RUN chmod +x /app/twitter
#15 DONE 0.2s
#16 [stage-1 5/5] RUN addgroup -S appgroup && adduser -S appuser -G appgroup
#16 DONE 0.4s
#17 exporting to image
#17 exporting layers 0.1s done
#17 writing image sha256:5dee901a2e3e44598364ed4714f820f5fd58ccc3521714fa22381e45bd2f98dc done
#17 naming to docker.io/library/twitter-web:8959c8af54ece8f7eb90abd0f41a99bc2edf9786 done
#17 DONE 0.1s





Task 4: Reflection

Question 8: 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?

Write requests would affect all of our services. Since we do not have a storage tier for blockchain and qr code, we would need to find some way to implement that. Additionally, with the number of requests that we are expected to serve for each test, the concurrency control would be a nightmare. Our design is not well-optimized for PUT 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.

- 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.)

With proper horizontal scaling via read replicas and proportional machine count, the database we used in this phase could handle 10x data volume effectively. Memory and query performace would be the main issue in this case, partitioning of data would make this better, but might not be 100% necessary. At 100x the quantity of data, there would need to be fundamental design changes. The physical limitations of the indexes alone would cause issues. We would likeley find that I/O woud become a major bottleneck. To make the service work at this size, data sharding/partitioning would become essential, and the implementation of a caching layer would be mandatory as well.

Question 9: 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?

In general it was easier to test and develop locally in phase 3 than in phase 2. When improving our ETL process locally we used Databricks as it provided free access to a compute powerful enough to handle all the data for this project. . Additionally, since we used RDS to house our mySQL database, we could connect to it from a local repository and did not have to set up an instance explicitly for testing the twitter service implementation (while we were working on improving queries). To test 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 yes, how? And why? (Optional)

We did generate a load test locally using a bash script, but our OSs (MacOS) has a limit on requests concurrency that is lower than what our CPUs were able to handle, so the only way for us to see the performance improvements was by doing load tests in the cloud.

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

We wish we had more time and funds to test and use a higher performance database. We found that the Twitter service performance posed the most issues for us in this phase and didnt realize until the last week that we probably should have change our database service entirely rather than just improving our current implementation. If we had better success in Phase 2 we may have realized this earlier. Additionally, we would have liked to try some caching for our sql service, that might have increased performance, but its hard to say, we did not do any analysis on the frequency of user ids in the database.

- Which were the toughest roadblocks that your team faced in Phase 3, especially for managed services?

Cost was a big challenge we faced in this phase. Specifically, cost of the storage tier. Since the database instances were so expensive we struggled to get performance up even after improving our logical implementation. We thought we needed to use a memory efficient instance for our rds cluster and the cheapest one we could use was so expensive that we could only afford 1 instance (which meant our rps was low). We tried to solve this problem by improving our other services so that we could dedicate more resources to the twitter service.

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

While trying to improve the long running queries for our twitter service, we implemented batching. Though this was not the final solution, it was an interesting problem to work through, trying to determine how large of a batch might improve our query response time.

Task 5: Summary

Question 10: 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 the 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?

Initially, we used Azure for ETL , but we transitioned to Databricks in phase three. The user interface was more amenable to our skills and it supported a more flexible schedule for conducting final adjustments to our DB schema.

- Which programming framework did you use for ETL (e.g., MapReduce, Spark, other)? Why?

We used Spark to do ETL. Spark was the most user friendly programming framework for us to use. Additionally we already had an outline set up in phase two to use spark, so In phase 3, it was easy to make adjustments in the same notebook

- Which programming framework did you use for the web tier (e.g., Spring, Flask, other)? Why?

We used the go fast http web framework. When we started this project, we decided as a team that using go might be a good option for achieving higher performance without any explicit effort, as go web frameworks generally tend to perform better than java or python. We picked fast/http because it did not require ust to learn a lot of new syntax and package specific implementation steps which was essential for us spending time on the actual logical implementation of each service.

- Which kind of database did you use for Twitter Service e.g., SQL or NoSQL? Explain why and compare to other options by discussing their advantages and disadvantages.

We used MySQL for our Twitter Service database. We mostly chose this out of comfort (we knew we had the skills to implement it and make performance adjustments). Additionally, mySQL, and all relational databases, support structured and complex relationships between tables. In our original implementation (3 table) this was an important aspect to maintain. NoSQL databases like DynamoDB (which we briefly explored) support a more flexible schema where items are stored in key value pairs. If it had not been so costly to implement ($0.125 per million reads), we likely would have used something like this after transitioning to a one table schema.

- 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.

Our system architecture in Phase 3 gains significant performance:

Service	Throughput Phase 2	Throughput Phase 3
Blockchain	21542	46906
QRcode	20728	99059

Twitter	53	5119

Some of the increase in performance in twitter and qr code service was due to more efficient implementation (change in how we rotated the qr code object and reduction from 3 tables to 1 table). However, a lot of the initial improvements also came from our use of an Elastic Load Balancer.

Cost:

Phase 2	Phase 3
$.71/hour	$1.22/hour

Though the Phase 3 architecture was almost 2 times as expensive as phase three, it was well worth it in terms of performance. The benefit of performance is also supplemented by the ease of management, making the additional cost to implement the Phase 3 version of our systems worth it.

In terms of deployment, both architectures required relatively the same amount of effort. But using managed services in phase 3 made it significantly easier to maintain and monitor than the manually implemented architecture in phase 2.

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

We learned a lot about how improved code efficiency can help improve the performance of a service. For example, while we were trying to improve our throughput for phase 3, Eric was able to make the QR code logic more efficient which increased our RPS from around 40000 to around 98000 by avoiding rotating the QR code matrix. What we think is cool is that the way we found out the matrix rotation was inefficient was by using the builtin Go profiler library, which gave us a complete report with the hottest functions/routines in our workflow.
This allowed us to reduce the resources dedicated to qr code service and put them towards twitter, improving the overall performance of our system without losing any accuracy.