LOG8415E: Advanced Concepts of Cloud Computing

Fall 2023

Final assignment: Scaling databases and implementing cloud design patterns

Group 01

1984959 - John Maliha

Submitted to Vahid Majdinasab

Thursday december 28th 2023

Table of Contents

1. Benchmarking MySQL stand-alone vs. MySQL Cluster 3
1.1) MySQL standalone: output of the sysbench test. 3
1.2) MySQL cluster: 4
1.2) Comparison between both type of installations 5
2) Implementation of The Proxy pattern 6
3) Implementation of The Gatekeeper pattern 8
4) Describe clearly how your implementation works 10
5) Demonstration of what to expect when project is running 10
6) Summary of results and instructions to run your code 12
7) Link to the demo video and Git hub 13

1. Benchmarking MySQL stand-alone vs. MySQL Cluster

In this section, I will discuss benchmarking the standalone and cluster installations of MySQL. The installation process for the MySQL cluster involves creating a configuration that links the manager node with the worker nodes using private IPs. These IPs were manually assigned in the Terraform script. As a result, the configuration file and the connection process are automated, eliminating the need for modifications before running the creation script. The standalone installation is straightforward; executing sudo apt-get install mysql-server is sufficient.

1.1) MySQL standalone: output of the sysbench test.

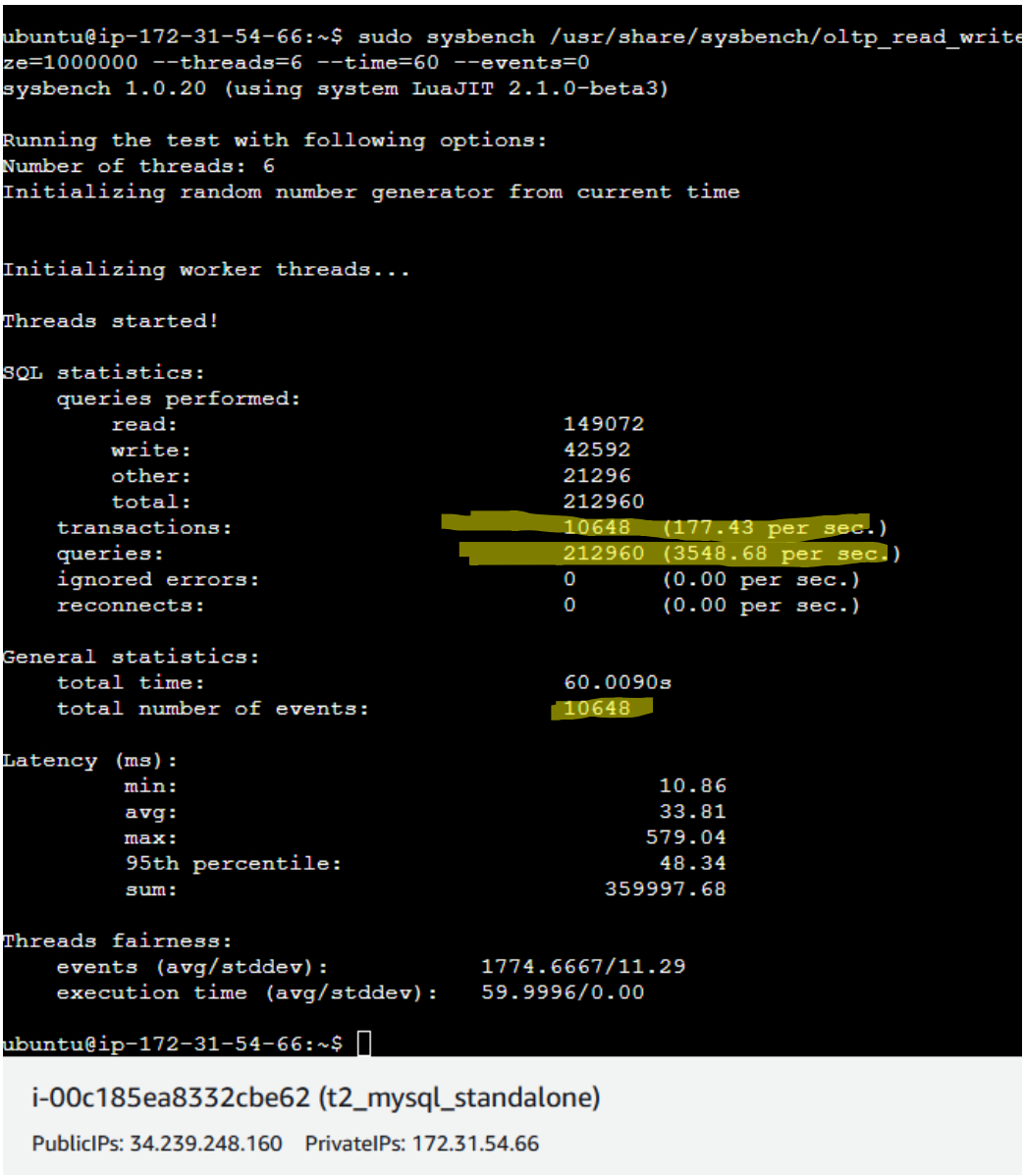


Image 1: Output of the Sysbench test for the standalone MySQL.

1.2) MySQL cluster:

Image 2: Output of the Sysbench test for the MySQL cluster.

Both the MySQL cluster and standalone installations were benchmarked using Sysbench. I conducted tests on both types of installations using a read-write test with a table size of 1,000,000. Each test was run with 6 threads and lasted for 60 seconds. This setup provides a comparative analysis of the performance under similar workload conditions, allowing for an assessment of how each installation type handles concurrent read and write operations over a significant dataset.

Here is the code used to run the tests on both the standalone and the cluster: sudo sysbench /usr/share/sysbench/oltp_read_write.lua run --db-driver=mysql --mysql-db=sakila --mysql-user=root --mysql-password --table-size=1000000 --threads=6 --time=60 --events=0

1.2) Comparison between both type of installations

The standalone installation operates on a single EC2 instance, with the Sakila database installed locally on this instance. In contrast, the cluster installation utilizes four EC2 instances, comprising one manager node and three worker nodes. On the manager instance, MySQL and the testing database, presumably SakilaDB, are installed. Once the connection between the manager and worker nodes is successfully established, MySQL and SakilaDB are then propagated to the worker nodes.

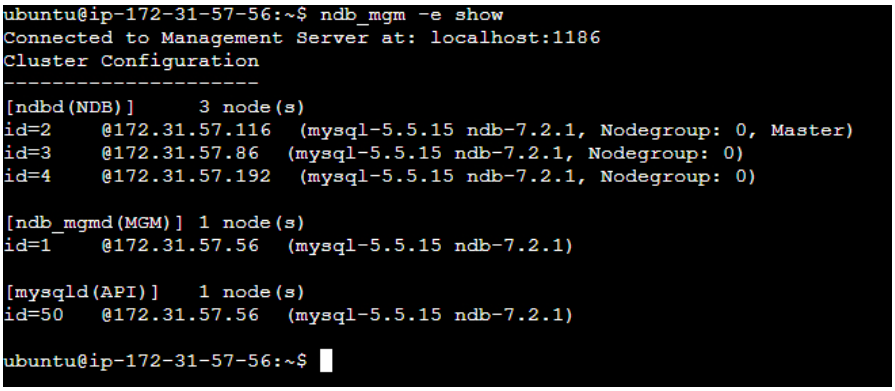


Image 3: Connection status of the worker nodes using ndb_mgmd command (management node)

The MySQL cluster is faster than the standalone MySQL installation. It is evident that the cluster installation can accomplish more events than the standalone within a 60-second runtime. Specifically, the cluster processed 2,122 more events compared to the standalone. Furthermore, the MySQL standalone processed 10,648 transactions and 212,960 queries, averaging 177.43 transactions per second and 3,548.68 queries per second. In comparison, the MySQL cluster managed 12,770 transactions and 255,400 queries, averaging 212.76 transactions per second and 4,255.30 queries per second.

Therefore, the MySQL cluster can handle 19.91% more queries per second than the MySQL standalone installation.

2) Implementation of The Proxy pattern

The proxy pattern is implemented as a Flask application running in a Docker container. This proxy can operate in three modes: direct-hit, random-hit, and custom-hit.

The direct-hit proxy implementation directly forwards incoming requests to the MySQL master node without any additional processing.

The random-hit proxy implementation randomly selects a worker node from the MySQL cluster and forwards the request to that node. This functionality is achieved using Python's random function. The code for this, as shown in image 3, involves taking the list of worker IPs and selecting one at random.

```python
def get_random_worker(workers):
    return random.choice(workers)
```

Image 4: return a random worker.

The custom-hit proxy implementation measures the ping time of all the servers (workers and the data node) and forwards the request to the server with the lowest response time, thus targeting the fastest node. This feature is implemented using the pythonping library. Firstly, I created the find_speed_of_workers_ms() function that takes a worker's IP as a parameter and returns the average ping time (as seen in image 4). Then, the find_fastest_worker_node() method takes the list of worker IPs and returns the fastest worker along with its response time.

```python
def find_speed_of_workers_ms(worker_ip):
    try:
        response = ping(worker_ip, count=1, timeout=1)
        if response.success():
            # The attrib rrt_avg_ms returns the ping time.
            # https://stackoverflow.com/questions/67476432
            avg_time_ms = response.rtt_avg_ms
            # print(avg_time_ms)
            return avg_time_ms
        else:
            return None
    except Exception as e:
        print(f"No response was returned: {e}")
        return None
```

Image 5: implementation of the finding the ping time.

```python
def find_fastest_worker_node(all_workers):
    min_avg_time = float('inf')

    for worker in all_workers:
        avg_time = find_speed_of_workers_ms(worker)
        if avg_time < min_avg_time:
            min_avg_time = avg_time
            fastest_node = worker
    return fastest_node,min_avg_time

# convert the ip address to string
def to_string_ip(ip_address):
    return str(ip_address)
```

Image 6: Returns the fastest node with the average time.

Using SSHTunnelForwarder, I established a connection between my proxy and the MySQL cluster nodes. I then connected to the local testing database installed on my worker node using the PyMySQL library. Requests sent to the proxy are redirected to a worker node in the MySQL cluster. The function returns the response with "Response from the manager/worker nodes", as shown in image 6. Once the tunnel is established, we connect to the Sakila database running on the manager node, which is also accessible on the worker nodes due to their connection to the master node. Queries can then be run on the database. Note: queries are passed as parameters in the proxy's URL. After each request is completed and the response is received, I chose to close the tunnel to avoid maintaining a constant connection.

```python
def ssh_connection_handler(manager_ip,worker_ip,sql_query):
    resp = "Response from the manager/data nodes : \n"
    if not manager_ip:
        raise Exception("Manager IP is required")

    with SSHTunnelForwarder(
        (worker_ip,22),
        ssh_username='ubuntu',
        ssh_pkey='final_assignment.pem',
        remote_bind_address=(manager_ip, 3306),
        #local_bind_address=('0.0.0.0', 5000)
    ) as tunnel:

        # connect to the sakila db using mysql via the tunnel
        connection = pymysql.connect(
            host=manager_ip,
            user='root',
            password='', # i did not set any passwords for my sakila db.
            db='sakila',
            port=3306,
            autocommit=True
    )

    try:
        with connection.cursor() as cursor:
            print(f"Connection to data-master nodes : Tunnel established to manager node {mana
            cursor.execute(sql_query)
            response = cursor.fetchall()
            for row in response: # look into response.json() and json.dumps()       You, 16 mi
                resp = resp + str(row)
                print(resp)
    except Exception as e:
        print(f"Connection to data-master nodes : error establishing SSH tunnel: {e}")
    finally:
        connection.close()
        print("Connection to data-master nodes : Tunnel closed")
    return resp
```

Image 7: Implementation of the SSH connection from the proxy to the MYSQL cluster nodes.

Finally, the proxy URL is formatted as follows: http://NODE_ADDRESS/PROXY_TYPE?query=SELECT COUNT(*) FROM film; where NODE_ADDRESS is the address of the manager or worker nodes from the MySQL cluster, and PROXY_TYPE is the type of proxy implementation (direct-hit, custom-hit, or random-hit).

3) Implementation of The Gatekeeper pattern

The gatekeeper pattern is divided into two components: the trusted host, which receives requests from the gatekeeper and forwards them to the proxy, and the gatekeeper itself, which accepts requests from any source, verifies their safety, and then sends them to the trusted host. Let's first examine the trusted host.

The trusted host is a Flask application running in a Docker container, deployed on a t2.large EC2 instance. This instance's security group is configured to allow traffic solely from the gatekeeper.

To connect to the proxy, SSHTunnelForwarder is used again for SSH connections. Once the connection is established via HTTP, a request is constructed using the proxy's IP address, the proxy type, and the SQL query. These parameters (the proxy type and SQL query) are added as query parameters in the request URL. This setup then establishes a connection with the proxy machine via the tunnel and sends the request to the proxy. The proxy in turn returns the response it receives from the MySQL cluster.

The gatekeeper is also a Flask app running in a Docker container, deployed on a t2.large instance, and operates with similar logic. The key distinction lies in its security group settings: the gatekeeper is internet-facing and accepts requests from all sources. It redirects traffic to the trusted host machine. To send a request to the gatekeeper, one could use request.py, Postman, or a web browser. After establishing a tunnel between the gatekeeper and the trusted host, an HTTP request is used to send the required query and proxy implementation type to the trusted host. Once the SSH tunnel is established between the two instances, the trusted host's DNS address is used to forward the request. Although the public IP could have been used, the DNS was chosen based on a recommendation found in a Stack Overflow post. In image 8, the DNS variable contains the address appended with /trusted_host, indicating it’s the route for the trusted host with two parameters.

```
def ssh_handler(trusted_host_dns, proxy_type, sql_query):
    response = "Response from trusted host: \n"

    with SSHTunnelForwarder(
        (trusted_host_dns,22),
        ssh_username='ubuntu',
        ssh_pkey='final_assignment.pem',
        remote_bind_address=(trusted_host_dns, 80) # we will use http to send the request.
    ) as tunnel:
        try:
            dns = f'http://{trusted_host_dns}/trusted_host?proxy_type={proxy_type}&query={sql_
            # send a request to the trusted host via http. The trusted host, will redirected i
            res = requests.get(dns)
            # print(f'http://{trusted_host_dns}/trusted_host?proxy_type={proxy_type}&query={sq
            print(res.text)
            response = response + ' ' + str(res.text)

        except Exception as e:
            print(f"Connection to trusted host: error establishing SSH tunnel: {e}")
        finally:
            print("Connection to trusted host : Tunnel closed")
    return response
```

Image 8: The implementation of the gatekeeper with SSHTunnelForwarder and HTTP (requests).

```python
@app.route('/gatekeeper', methods=['GET'])
def sending_to_trusted_host():
    proxy_type = request.args.get('proxy_type')
    params = request.args.get('query')
    trusted_host_dns = get_trusted_host_address_to_dns()
    print(f"Connection gatekeeper : Sending requests to trusted_host with proxy type : {proxy_
    return ssh_handler(trusted_host_dns=trusted_host_dns, proxy_type=proxy_type, sql_query=para
```

Image 9: The code of the route for the gatekeeper.

As illustrated in image 9, the route for the gatekeeper Flask application is set to /gatekeeper. This means that the address to send requests to is formatted as follows: http://GATEKEEPER_IP/gatekeeper?proxy_type=TYPE&query=QUERY_TO_EXECUTE. Once a request is made to this address, the processing of the request is handled automatically by the ssh_handler() function. This setup streamlines the process, allowing for efficient and secure handling of requests through the gatekeeper.

```
resource "aws_security_group" "final_projet_security_group_trusted_host" {
  name        = "final_projet_security_group_trusted_host"
  description = "Allow traffic to the trusted group"
  vpc_id      = data.aws_vpc.default.id

  # Define your security group rules here
  You, 4 days ago | 1 author (You)
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["172.31.50.248/32"] # private ip of the gatekeeper. the trus
  }
   You, 3 days ago | 1 author (You)
   ingress {
    from_port   = 80 # for when we bind the trusted host via http. (remote_binc
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["172.31.50.248/32"]
  }
  You, 4 days ago | 1 author (You)
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Image 10: Security group for the trusted host

As seen in image 10, the trusted host must be secured, meaning that it is not accessible by anyone and is not internet facing. The gatekeeper we receive all requests potentially from anywhere and reject those that are not secure. Then the gatekeeper will forward the allowed requests to the trusted host. The gatekeeper is the only module whitelisted in the trusted host's security group.

4) Describe clearly how your implementation works

My project is organized into five distinct sections: the EC2 instances configured in Terraform for creating the MySQL cluster, the proxy, the trusted host, the gatekeeper, and the handling of requests. Each component, except for the terraform, is a Flask application running inside a container hosted on Docker Hub. Upon creation of the instances — t2.micro for the clusters and t2.large for the proxy, gatekeeper, and trusted host — a script is executed on each machine to install Docker and pull the relevant Docker image. To retrieve the dynamically allocated public IPs of each instance, I utilized the boto3 library. The instances are created via terraform an infrastructure-as-code software tool to allow the creation of complex AWS architecture.

The cluster was constructed based on the guidelines provided in the assignment instructions. The proxy, gatekeeper, and trusted host components were developed through online research and implemented in Python. The overarching concept is that the gatekeeper is internet-facing, serving as the system's entry point, while the other components are more secure and not exposed to the internet.

Requests are sent to the gatekeeper, which, after ensuring the request and SQL query are secure, forwards them to the proxy. The proxy then processes these requests based on the type of proxy implementation (direct, custom, or random) specified by the user. Once the proxy identifies the appropriate data or worker nodes, it forwards the request to them. The manager/data nodes subsequently respond with a message that includes "Response from the manager/data nodes:" followed by the database's response.

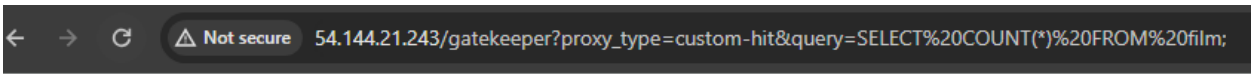


Image 11: Response returned from the database when sending a request to the gatekeeper with parameters: custom-hit and a select count query.

In image 10, the phrase "Response from the manager/data nodes: (1000,)" represents the response that is returned from the data nodes to the proxy. This output is then relayed as the "response from proxy," which is what the trusted host receives. Finally, the "response from trusted host" is the output that the gatekeeper receives, and this is the response that is ultimately presented to the user. This sequence of responses illustrates the flow of data from the data nodes through the system's various components to the end user.

5) Demonstration of what to expect when project is running

See the video linked below in section 7.

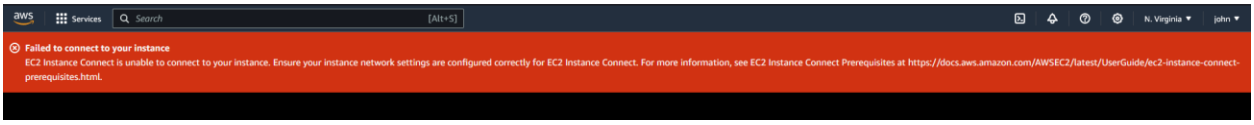


Image 12: Connection to ec2 instance trusted host

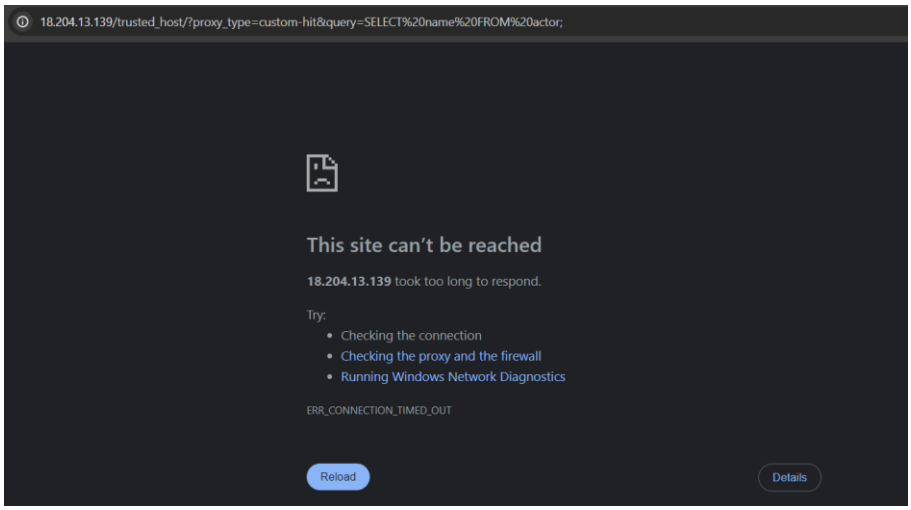
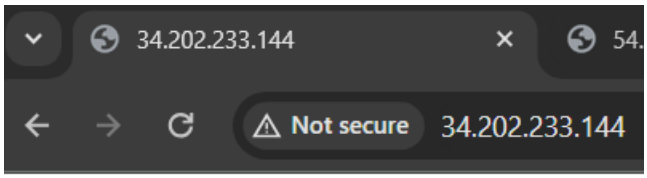


Image 13: Browser response when sending a request to the trusted host.

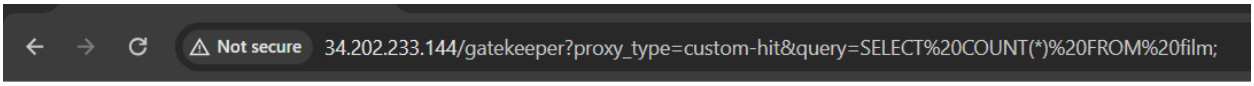
In images 11 and 12, we can observe that the security group for the trusted host is functioning effectively, ensuring the security of the instance. This setup restricts access so that only the gatekeeper can send requests to the trusted host, blocking all other sources.

For testing and debugging, each of my EC2 instances is equipped with a log file named 'logs.log', which records all outputs from the commands executed in their respective scripts. To view the contents of this log file, one can use commands such as cat logs.log or tail -f logs.log. This approach aids in monitoring and troubleshooting the system's operations.



ec2-18-204-13-139.compute-1.amazonaws.com

Image 14: The default response from the gatekeeper.



Response from trusted host: Response from proxy: Response from the manager/data nodes : (1000,)

Image 14: Response (seen from a users point-of-view) from the gatekeeper when a proxy type and a query is added.

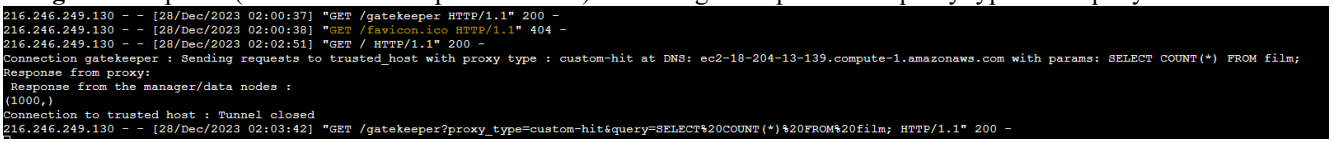


Image 15: Logs generated while handling and redirecting the request.

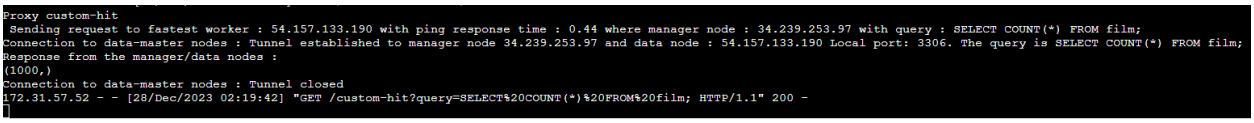


Image 16: Logs generated in the proxy after the trusted host forwards the request.

In images 14, 15, and 16, a request is sent to the gatekeeper to execute the SQL query SELECT COUNT(*) FROM Film using the custom-hit proxy implementation. Image 16 specifically shows the proxy processing this request. It indicates that the end user has chosen the custom-hit option, and the fastest node identified by the proxy is 54.157.133.190, with a response time of 0.44. Following this, the image displays the response returned by the data nodes. This sequence of images effectively demonstrates the flow of the request through the system, from the gatekeeper to the proxy and then to the data nodes, along with the proxy's decision-making process in selecting the fastest node.

6) Summary of results and instructions to run your code

To conclude, I successfully installed both the MySQL standalone and MySQL cluster on separate t2.micro EC2 instances. Additionally, I implemented a pipeline to send requests from either a browser or a requests Docker container. This container includes a Flask app designed to send requests to the gatekeeper with two parameters: the sql_query, which is intended for execution on the test database sakila-db, and the proxy_type, which indicates the type of proxy (direct-hit, random-hit, or custom-hit).

Each request follows a specific path: from the gatekeeper to the trusted host (which is secured), then to the proxy, and finally to either the data nodes or management nodes, depending on the nature of the query. Read requests are managed by the data nodes (MySQL cluster workers), while write operations are handled by the MySQL workers. At each step, the process and the server's response are logged, allowing users to track the progress and outcomes by running tail -f logs.log on the proxy and gatekeeper EC2 instances.

The results show that the MySQL cluster outperforms the standalone MySQL installation by approximately 19%, which aligns with expectations.

Due to some unexpected issues with the AWS learner account, I couldn't complete the implementation of request.py for automating the request-sending process. However, using Postman or a web browser with the public IP of the gatekeeper, formatted as gatekeeperIP/gatekeeper?proxy_type=…&query=…, serves as an effective alternative for making requests.

Before running the run_project.sh script, make sure to create all the required credentials.py files and the terraform.tfvars containing your aws access_key, secret_key and token. Also, it is important to create the RSA .pem key. To do so:

1. On AWS dashboard, click on the key pairs option.
2. Click on create key pair.
3. Type a name, I personally used final_assignment
4. Make sure to select the .pem extension

5. Copy the key in the following folders: proxy, gatekeeper, trusted host, and requests.

Here are the instructions to run the code:

1) cd LOG8415-Final-Project
2) To run the project with one command
    a. If on Windows: bash run_project.sh
    b. If on Linux: ./run_project.sh
3) The ec2 instances will create and run the require code to install MYSQL. The flask apps containing the proxy, trusted host and gatekeeper will compile in a docker image and they will all be pushed via docker hub.
    a. After waiting for 2-3 minutes (to make sure everything finished installing) run the docker image requests.
    b. To do so, run the script create_request_docker.sh.

I also implemented the option to run each component manually. For example, the create_docker_****.sh allows to create tag and push to docker hub the docker image.

The create_terraform.sh allows to run and create the ec2 instances.

The nuke_terraform.sh allows to delete all the instances running on AWS.

7) Link to the demo video and Git hub

The GitHub repo: https://github.com/JohnMaliha/LOG8415-Final-Project

Link to the demo video: https://youtu.be/7bFJIyPy_-0