# LOG8415
# Advanced Concepts of Cloud Computing

**Scaling Databases and Implementing Cloud Design Patterns**

Final project

Fall 2023

**By:**
Ming Xiao Yuan - 1949477

**Presented to:**
Vahid Majdinasab

Thursday, December 28 2023

To access the Github repository, click here.

The link is https://github.com/ming-xiao-yuan/scaling_db_and_cloud_design_pattern

To access the video explanation, click here.

The link is https://www.youtube.com/watch?v=_OWaNBUFunQ

# 1 Introduction

In this final project for LOG8415 at Polytechnique Montreal, we examine database scalability and cloud design patterns, focusing on a MySQL Cluster setup on Amazon EC2. Our project is aimed at exploring and comparing the performance of MySQL Server and MySQL Cluster using the Sysbench benchmark. This comparison will provide valuable insights into the efficiency and scalability of these database management systems under various workloads. We will set up both a MySQL Server and a MySQL Cluster on Amazon EC2 instances and use the Sysbench tool to benchmark their performance. This process will involve configuring and testing the systems under different scenarios to understand their behavior and limitations.

In the second phase of our project, we will implement the proxy and gatekeeper patterns to enhance our database architecture. For this purpose, we have utilized a range of technologies. We have deployed our infrastructure on Amazon EC2, using Terraform as our Infrastructure as a Service (IaaS) platform to manage and automate the deployment of our resources. The gatekeeper, trusted host, and proxy applications have been developed using Python Flask and are containerized using Docker. These Docker containers are then pushed to Dockerhub, from where they can be deployed to our EC2 instances. This architecture not only ensures a scalable and efficient database system but also incorporates essential security and management features through the gatekeeper and proxy patterns.

Our report progresses through a series of sections: comparing the performance of MySQL Stand-Alone versus MySQL Cluster, implementing and analyzing the Proxy and Gatekeeper patterns for enhanced database security and efficiency, a detailed explanation of our technical implementations, and concluding with a summary of our findings and instructions for running the implemented solutions. This cohesive approach provides a thorough understanding of database management in cloud environments.

# 2 Benchmarking MySQL stand-alone vs. MySQL Cluster

For the benchmarking of the standalone MySQL instance, the process begins with installing MySQL Server on an Ubuntu EC2 instance. We use the 'root' user with no password for simplicity. The Sakila database is downloaded and used as a test database for benchmarking. Necessary steps are taken to download Sakila onto the EC2 instance. Then, we install Sysbench, a benchmarking tool, and configure it with specific parameters: 10 tables, a table size of 10,000, 4 threads, and a 60-second test duration. This setup provides a standardized environment to evaluate the performance

of the standalone MySQL server.

For the benchmarking of the MySQL Cluster, the process is similar that of the standalone MySQL instance. We begin by setting up a MySQL Cluster on a group of Amazon EC2 instances, configuring one instance as the manager node and three others as worker nodes. Consistency in our setup is maintained by using t2.micro instances for both the manager and the worker nodes. The manager node in the MySQL Cluster plays an important role in orchestrating the operations and communication within the cluster. It oversees the worker nodes, ensuring efficient data distribution and management. The worker nodes, on the other hand, are responsible for the actual data storage and processing. They work in tandem under the guidance of the manager node to provide a distributed database system. Our benchmarking for MySQL Cluster involves similar steps to those of the standalone MySQL Server, ensuring a consistent and fair comparison. We deploy the Sakila database on the cluster and use Sysbench to benchmark its performance. We use the same parameters used in the standalone MySQL benchmark.

The configuration for Sysbench was chosen to emulate a moderately complex database environment. Selecting 10 tables offers a balanced structure, neither too simple nor excessively complex, to adequately test database operations. A table size of 10,000 rows strikes a balance between testing performance under load while avoiding overly long test durations. Utilizing 4 threads simulates a realistic concurrency scenario, typical in many applications. The 60-second test duration ensures sufficient time to observe performance trends without unduly prolonging the benchmark. This configuration provides a comprehensive yet efficient way to assess the performance of the standalone MySQL instance.

The analysis of the results are discussed in the section 'Summary of results and instructions to run your code'.

## 3 Implementation of The Proxy pattern

In our project, the implementation of the proxy pattern plays an important role, acting as an intermediary between the trusted host and the MySQL Cluster to enhance read scalability in the relational database system. This pattern utilizes data replication, where all write requests are consistently forwarded to the manager node of the cluster, regardless of the proxy implementation chosen. It's the handling of read requests that differentiates the three approaches of the proxy pattern. In the 'Direct Hit' approach, the proxy straightforwardly routes read requests to the MySQL master node without any distribution logic. The 'Random Hit' implementation introduces a level of load balancing by having the proxy randomly select a worker node in the cluster to handle read requests. In contrast, the 'Customized Hit' method is more dynamic, where the proxy assesses the response times of the worker nodes and forwards the read requests to the one with the lowest latency. Operating on a t2.large instance, the proxy efficiently supports these different strategies, ensuring robust performance and scalability of the database system.

Figure. 1 represents the implementation of the proxy security group. The "aws_security_group" designated as "proxy_sg" plays a vital role in bolstering the security framework of our proxy setup within the AWS environment. This security group is linked to the default VPC, providing an es-

```
79
80  resource "aws_security_group" "proxy_sg" {
81    name        = "proxy_security_group"
82    description = "Security group for Proxy"
83    vpc_id      = data.aws_vpc.default.id
84
85    ingress {
86      from_port   = 22
87      to_port     = 22
88      protocol    = "tcp"
89      cidr_blocks = ["0.0.0.0/0"]
90    }
91
92    ingress {
93      from_port   = 80
94      to_port     = 80
95      protocol    = "tcp"
96      cidr_blocks = ["0.0.0.0/0"]
97    }
98
99    egress {
100     from_port        = 0
101     to_port          = 0
102     protocol         = "-1"
103     cidr_blocks      = ["0.0.0.0/0"]
104     ipv6_cidr_blocks = ["::/0"]
105     security_groups  = [aws_security_group.mysql_sg.id]
106   }
107 }
108
```

Figure 1: Implementation of the proxy security group

```python
def create_ssh_tunnel(worker_node):
    """
    Creates an SSH tunnel to the specified worker node.

    Args:
        worker_node (str): The worker node address to create an SSH tunnel to.

    Returns:
        SSHTunnelForwarder: An active SSH tunnel instance or None if failed.
    """
    global ssh_tunnel_port
    try:
        tunnel = SSHTunnelForwarder(
            (manager_db_config["host"], 22),
            ssh_username="ubuntu",
            ssh_pkey="/etc/proxy/my_terraform_key",
            remote_bind_address=(worker_node, 3306),
            local_bind_address=("127.0.0.1", 9000),
        )
        tunnel.start()
        app.logger.warning("Successfully establish SSH tunnel.")
        return tunnel
    except Exception as e:
        app.logger.error(f"Failed to establish SSH tunnel: {e}")
        return None
```

Figure 2: SSH Tunnel between Proxy and chosen worker node

sential layer of network isolation and control. The ingress rules of this security group are crafted for enhanced security. It permits SSH access exclusively on port 22, a standard practice that not only facilitates necessary administrative tasks but also adheres to best security practices by limiting access to a well-known, secure channel. Additionally, the inclusion of HTTP traffic on port 80 allows for standard web communication, crucial for handling web-based requests, while maintaining robust security through controlled access.

Furthermore, the egress rules are set to ensure secure and focused communication with the MySQL Cluster. By allowing all traffic types but restricting the destinations to only the MySQL Cluster's security group, the proxy's outbound communication is effectively safeguarded, allowing it to interact with the MySQL nodes securely. This configuration of ingress and egress rules in the "proxy_security_group" is important in maintaining a secure, efficient, and reliable database operation within our cloud-based infrastructure.

Figure. 2 represents the SSH Tunnel between proxy and the chosen worker node. The function 'create_ssh_tunnel' plays an important role in establishing a secure communication channel between the proxy and the worker nodes in the MySQL Cluster. This function is designed to create an SSH tunnel to a specified worker node. The tunnel is routed through the manager node of the MySQL Cluster, which acts as an intermediary, ensuring that the connection is secure and controlled. When the proxy receives a request, depending on whether the hit is random or customized, it invokes this function with the address of the targeted worker node. The SSH tunnel is established on port 22,

```python

@app.route("/fetch_direct", methods=["POST"])
def fetch_direct():
    """
    Endpoint for fetching data directly.

    Returns:
        Response: A JSON response with fetched data or an error message.
    """
    data = request.json
    sql = data.get("sql")

    if not sql:
        return jsonify({"error": "No SQL query provided"}), 400

    # Initialize 'conn' and 'cursor' as None
    conn = None
    cursor = None

    try:
        with mysql.connector.connect(**manager_db_config) as conn:
            with conn.cursor() as cursor:
                cursor.execute(sql)
                result = cursor.fetchall()
        return jsonify(result)
    except mysql.connector.Error as err:
        return jsonify({"error": str(err)}), 500

```

Figure 3: Function for direct hit

the standard port for SSH communications, which aligns with the security best practices. The function uses the SSH private key located at "/etc/proxy/my_terraform_key" for authentication, ensuring that the connection is authenticated and encrypted. Once the tunnel is established, it binds the local address of the proxy to port 9000 and the remote address of the worker node to MySQL's default port, 3306. This binding creates a secure and dedicated pathway for the proxy to communicate with the worker node, allowing for safe data queries and transactions. The use of SSH tunnels is important for maintaining data integrity and security. It ensures that data transmitted between the proxy and the MySQL worker nodes is encrypted and protected from eavesdropping or interception. This approach is particularly important in cloud environments, where network traffic may traverse public or semi-public networks.

Figure. 3 represents the function for the Direct hit, Figure. 4 represents the function for the Random hit and Figure. 5 represents the function for the Customized hit. In our proxy application, we have implemented three distinct endpoints:'fetch_direct', 'fetch_random', and 'fetch_customized', each suited to a specific proxy pattern strategy. The 'fetch_direct' function aligns with the Direct Hit strategy, directly executing SQL queries against the MySQL master node, ensuring immediate access to the most current data. The 'fetch_random' function, aligns with the Random Hit approach, it fetches data from a randomly selected worker node, utilizing an SSH tunnel for secure communication and achieving basic load balancing. Finally, the 'fetch_customized' function, aligns

```python

@app.route("/fetch_random", methods=["POST"])
def fetch_random():
    """
    Endpoint for fetching data from a randomly chosen node.

    Returns:
        Response: A JSON response with fetched data or an error message.
    """
    global random_ssh_tunnel, random_request_counter
    data = request.json
    sql = data.get("sql")

    if not sql:
        return jsonify({"error": "No SQL query provided"}), 400

    if random_ssh_tunnel is None or not random_ssh_tunnel.is_active:
        return jsonify({"error": "SSH tunnel is not established"}), 500

    conn = None
    cursor = None  # Initialize cursor to None

    # Use the existing SSH tunnel for the database connection
    manager_tunnel = {
        "host": manager_db_config["host"],
        "user": "root",
        "password": "",
        "database": "main_db",
    }
    try:
        with mysql.connector.connect(**manager_tunnel) as conn:
            with conn.cursor() as cursor:
                cursor.execute(sql)
                result = cursor.fetchall()

        random_request_counter += 1
        if random_request_counter >= 20:
            close_random_ssh_tunnel()
        return jsonify(result)
    except mysql.connector.Error as err:
        return jsonify({"error": "MySQL Error: {}".format(str(err))}), 500

```

Figure 4: Function for random hit

```python
@app.route("/fetch_customized", methods=["POST"])
def fetch_customized():
    """
    Endpoint for fetching data from the node with the lowest ping time.

    Returns:
        Response: A JSON response with fetched data or an error message.
    """
    data = request.json
    sql = data.get("sql")

    if not sql:
        return jsonify({"error": "No SQL query provided"}), 400

    conn = None
    cursor = None

    try:
        initialize_customized_ssh_tunnel()

        if not customized_ssh_tunnel or not customized_ssh_tunnel.is_active:
            return jsonify({"error": "SSH tunnel is not established"}), 500

        manager_tunnel = {
            "host": manager_db_config["host"],
            "user": "root",
            "password": "",
            "database": "main_db",
        }

        with mysql.connector.connect(**manager_tunnel) as conn:
            with conn.cursor() as cursor:
                cursor.execute(sql)
                result = cursor.fetchall()
        return jsonify(result)
    except mysql.connector.Error as err:
        return jsonify({"error": "MySQL Error: {}".format(str(err))}), 500

```

Figure 5: Function for customized hit

```
27
28  resource "aws_security_group" "gatekeeper_sg" {
29    name        = "gatekeeper_security_group"
30    description = "Allow web traffic to Gatekeeper"
31    vpc_id      = data.aws_vpc.default.id
32
33    ingress {
34      from_port        = 0
35      to_port          = 0
36      protocol         = "-1"
37      cidr_blocks      = ["0.0.0.0/0"]
38      ipv6_cidr_blocks = ["::/0"]
39    }
40
41    egress {
42      from_port        = 0
43      to_port          = 0
44      protocol         = "-1"
45      cidr_blocks      = ["0.0.0.0/0"]
46      ipv6_cidr_blocks = ["::/0"]
47      security_groups  = [aws_security_group.trusted_host_sg.id]
48    }
49  }
50
```

Figure 6: Gatekeeper security group

with the Customized Hit strategy, optimizes data retrieval by selecting the worker node with the lowest ping time, establishing an SSH tunnel to ensure efficient and responsive data access. Collectively, these functions demonstrate the versatility of the proxy application and how it handles diverse data requests while maintaining security and performance.

## 4 Implementation of The Gatekeeper pattern

The ‘aws_security_group‘ defined as ‘gatekeeper_sg‘ in our project serves as a critical security measure for the Gatekeeper component, which is an internet-facing instance. Designed with the purpose of allowing web traffic to the Gatekeeper, this security group plays an important role in our network security architecture, especially for the demonstration aspects of our assignment. The ingress rules of ‘gatekeeper_sg‘ are configured to allow all types of traffic from any source, as indicated by the wide-open CIDR blocks ‘0.0.0.0/0‘ for IPv4 and ‘::/0‘ for IPv6. This setup means that the Gatekeeper can receive any kind of request, irrespective of the protocol or port, aligning with the need to demonstrate a broad range of interactions and potential security scenarios in a controlled environment. By not restricting the ingress traffic by port or protocol, we can simulate various types of web traffic and attacks, providing a comprehensive view of how the Gatekeeper reacts and handles different situations. Similarly, the egress rules are set to allow all types of outbound traffic

```
# SSH Tunnel setup
server = SSHTunnelForwarder(
    (trusted_host_dns, 22),  # Remote SSH server
    ssh_username="ubuntu",
    ssh_pkey="/etc/gatekeeper/my_terraform_key",
    remote_bind_address=(trusted_host_dns, 80),  # Trusted Host server port
    local_bind_address=("127.0.0.1", 80),
)

try:
    server.start()  # Start SSH tunnel
    app.logger.warning("SSH Tunnel successfully established!")
except Exception as e:
    app.logger.error(f"Error establishing SSH Tunnel: {e}")
    raise
```

Figure 7: Gatekeeper SSH Tunnel

to any destination, with a specific route to the 'trusted_host_sg'. This implies that the Gatekeeper can send requests to any external entity, facilitating interactions beyond the immediate AWS infrastructure. However, it's particularly tailored to communicate securely with the trusted host, as defined by the link to 'aws_security_group.trusted_host_sg.id'. This configuration of 'gatekeeper_sg' is important for the security demonstration aspect of our assignment. It allows us to showcase how the Gatekeeper can handle a wide range of internet-based interactions, both inbound and outbound, while maintaining a secure and controlled connection with the trusted host. This open yet focused approach is key to understanding and demonstrating the complexities of web traffic and security in a cloud-based environment.

Figure. 7 represents the SSH Tunnel between the Gatekeeper and the Trusted Host. In our Gatekeeper application, the setup of an SSH tunnel is a crucial aspect of ensuring secure communication with the trusted host. The provided code snippet demonstrates the establishment of this SSH tunnel. The 'SSHTunnelForwarder' object, named 'server' in the script, is configured to create an encrypted tunnel between the Gatekeeper and the trusted host. The tunnel connects to the trusted host at its DNS address on port 22, which is the standard port for SSH connections. This ensures that the tunnel is leveraging the security features of SSH, such as strong encryption and secure authentication, using the specified SSH key located at "/etc/gatekeeper/my_terraform_key". The remote binding of the tunnel is set to port 80 of the trusted host, which is typically used for HTTP traffic. This implies that any HTTP request received by the Gatekeeper is intended to be securely forwarded through this tunnel to the trusted host. Conversely, the local binding on the Gatekeeper side is also set to port 80, aligning with the standard port for receiving web traffic. Upon execution, the script attempts to start the SSH tunnel, and if successful, logs a warning message indicating that the tunnel has been successfully established. This setup is important for the security of our system, as it ensures that all traffic between the Gatekeeper and the trusted host is encrypted and securely transmitted over the SSH tunnel. This mechanism is essential not only for protecting the data in transit but also for maintaining the integrity of the interactions between these two critical components of our infrastructure. Any failure in establishing this tunnel is logged

as an error, highlighting the importance of this secure connection in the overall functionality of the Gatekeeper.

The Gatekeeper application features three distinct endpoints: '/fetch_direct', '/fetch_random', and '/fetch_customized', each designed to handle specific types of data fetch requests in alignment with the chosen proxy strategy. The '/fetch_direct' endpoint processes direct fetch requests by forwarding them to the trusted host's '/fetch_direct' URL. It captures the response from the trusted host and relays it back to the original requester, effectively acting as a transparent intermediary. Similarly, the '/fetch_random' endpoint deals with requests that require fetching data from a randomly selected node. It redirects these requests to the trusted host's '/fetch_random' URL, following the same pattern of capturing and relaying the response. Lastly, the '/fetch_customized' endpoint caters to requests that seek the most efficient data retrieval, based on the lowest ping time to a node. It forwards these requests to the trusted host's '/fetch_customized' URL and handles the responses in a similar manner. All three functions log their respective activities and handle exceptions by logging errors and returning appropriate error messages and HTTP status codes. This setup ensures that the Gatekeeper can effectively manage various types of data requests, maintaining a seamless flow of information between the client, the Gatekeeper, and the trusted host.

Figure. 9 represents the security group of the trusted host. The implementation of the 'trusted_host_sg' security group for the Trusted Host plays an important role in fortifying its security within our cloud architecture. This security group is configured to bolster the security measures around the Trusted Host, aligning with the assignment's emphasis on reinforced security protocols. It specifically allows only SSH and HTTP requests through ports 22 and 80, respectively. Such a configuration adds a layer of security by restricting access to these critical communication ports. Notably, the ingress rules are narrowly defined with CIDR blocks set to '172.31.49.42/32', which means that only requests originating from the Gatekeeper are permitted. This selective approach ensures that the Trusted Host is safeguarded against unauthorized access, thereby maintaining its integrity as a core component of our system. Furthermore, the egress rules of the 'trusted_host_sg' are designed to allow communications exclusively with the Proxy security group, as indicated by the inclusion of 'aws_security_group.proxy_sg.id'. This ensures that outbound traffic from the Trusted Host is strictly limited to interactions with the Proxy, aligning with the system's design where the Trusted Host acts as an intermediary between the Gatekeeper and the Proxy. By allowing unrestricted protocol and port ranges for egress to the specified security group, the Trusted Host can efficiently handle various types of data requests while maintaining a secure and controlled communication channel with the Proxy.

Figure. 10 represents the SSH Tunnel from the Trusted host to the Proxy. The Trusted Host in our system employs an SSH tunnel setup, analogous to the one between the Gatekeeper and the Trusted Host, to securely communicate with the Proxy. This setup is important for ensuring that data transmitted between these two components is protected and private. The SSH tunnel is established using the 'SSHTunnelForwarder' object, 'server', which is configured to connect to the Proxy's DNS on port 22, the standard for SSH communications. This setup guarantees the use of SSH's robust security features, including strong encryption and secure authentication, utilizing an SSH key specified at "/etc/trusted_host/my_terraform_key". The remote binding of the tunnel is set to the Proxy's server on port 80, typically reserved for HTTP traffic, indicating that the Trusted

```python

@app.route("/fetch_direct", methods=["POST"])
def fetch_direct():
    app.logger.warning("Received direct fetch request")
    try:
        trusted_host_url = f"http://{trusted_host_dns}/fetch_direct"
        response = requests.post(trusted_host_url, json=request.get_json())
        return jsonify(response.json()), response.status_code
    except Exception as e:
        app.logger.error(f"Error in /fetch_direct: {e}")
        return jsonify({"error": str(e)}), 500


@app.route("/fetch_random", methods=["GET", "POST"])
def fetch_random():
    app.logger.warning("Received random fetch request")
    try:
        trusted_host_url = f"http://{trusted_host_dns}/fetch_random"
        response = requests.post(trusted_host_url, json=request.get_json())
        return jsonify(response.json()), response.status_code
    except Exception as e:
        app.logger.error(f"Error in /fetch_direct: {e}")
        return jsonify({"error": str(e)}), 500


@app.route("/fetch_customized", methods=["GET", "POST"])
def fetch_customized():
    app.logger.warning("Received customized fetch request")
    try:
        trusted_host_url = f"http://{trusted_host_dns}/fetch_customized"
        response = requests.post(trusted_host_url, json=request.get_json())
        return jsonify(response.json()), response.status_code
    except Exception as e:
        app.logger.error(f"Error in /fetch_direct: {e}")
        return jsonify({"error": str(e)}), 500
```

Figure 8: Routes for Gatekeeper

```
50
51  resource "aws_security_group" "trusted_host_sg" {
52    name        = "trusted_host_security_group"
53    description = "Security group for Trusted Host"
54    vpc_id      = data.aws_vpc.default.id
55
56    ingress {
57      from_port   = 22
58      to_port     = 22
59      protocol    = "tcp"
60      cidr_blocks = ["172.31.49.42/32"] # Only accept SSH from Gatekeeper
61    }
62
63    ingress {
64      from_port   = 80
65      to_port     = 80
66      protocol    = "tcp"
67      cidr_blocks = ["172.31.49.42/32"] # Only accept HTTP from Gatekeeper
68    }
69
70    egress {
71      from_port        = 0
72      to_port          = 0
73      protocol         = "-1"
74      cidr_blocks      = ["0.0.0.0/0"]
75      ipv6_cidr_blocks = ["::/0"]
76      security_groups  = [aws_security_group.proxy_sg.id]
77    }
78  }
79
```

Figure 9: Trusted Host Security group

```
14
15  # SSH Tunnel setup
16  server = SSHTunnelForwarder(
17      (proxy_dns, 22),  # Remote SSH server
18      ssh_username="ubuntu",
19      ssh_pkey="/etc/trusted_host/my_terraform_key",
20      remote_bind_address=(proxy_dns, 80),  # Proxy server port
21      local_bind_address=("127.0.0.1", 80),
22  )
23
24  try:
25      server.start()  # Start SSH tunnel
26      app.logger.warning("SSH Tunnel successfully established!")
27  except Exception as e:
28      app.logger.error(f"Error establishing SSH Tunnel: {e}")
29      raise
30
```

Figure 10: Trusted Host SSH Tunnel

Host forwards HTTP requests securely to the Proxy via this tunnel. Conversely, the local binding on the Trusted Host's side is also set to port 80, aligning with standard practices for receiving web traffic. Upon initiation, the script attempts to start the tunnel and, if successful, logs a message confirming its establishment. However, in case of any failure in setting up the tunnel, an error is logged, and an exception is raised. This setup not only parallels the Gatekeeper's SSH tunnel configuration but also forms a critical part of our system's security strategy, ensuring encrypted and secure communication between the Trusted Host and the Proxy.

Figure. 11 represents the different routes for the Trusted Host for the Direct Hit, Random Hit and Customized Hit. In our Trusted Host application, the implementation of the routes '/fetch_direct', '/fetch_random', and '/fetch_customized' mirrors the approach taken in the Gatekeeper, each catering to a specific type of data request in accordance with the proxy pattern. The '/fetch_direct' route is designated for handling Direct Hit requests, where it receives a direct fetch request and forwards it to the Proxy using its DNS. Similarly, the '/fetch_random' route deals with Random Hit requests, directing these to the Proxy's '/fetch_random' endpoint. This route randomly selects a node in the MySQL cluster for data retrieval, demonstrating a basic form of load balancing. Lastly, the '/fetch_customized' route is aligned with the Customized Hit strategy, where it sends requests to the Proxy's '/fetch_customized' endpoint. This route is designed to optimize data retrieval by selecting the node with the lowest response time, ensuring efficient data access. Each of these routes logs the reception of requests and attempts to forward them to the respective endpoints on the Proxy, handling responses and errors accordingly.

# 5 Describe clearly how your implementation works

Our project's infrastructure is orchestrated using Terraform, an Infrastructure as Code tool, which automates the deployment of a complex cloud architecture on AWS. Through the 'main.tf' file, we initiate the creation of seven AWS EC2 instances, each designated for a specific function: the Gatekeeper, Trusted Host, Proxy, MySQL Cluster Manager, and three MySQL Cluster Workers. Alongside these instances, Terraform also establishes four distinct security groups to ensure secure and controlled network access for each component. The Gatekeeper instance, responsible for initial web request handling, is configured via user data scripts to download Docker and pull its dedicated Flask application from Docker Hub. Similarly, the Trusted Host and Proxy instances use Docker to deploy their respective Flask applications, crucial for request processing and routing within our architecture. The MySQL Cluster Manager, equipped with the necessary libraries and configuration files, awaits the connection of Worker nodes, which are also set up through user data scripts to install required packages and establish communication with the Manager. Once the cluster is fully operational, the Sakila database is integrated, and Sysbench is employed to benchmark the system's performance under varying loads. Furthermore, 'direct_table', 'random_table', and 'customized_table' are created in the 'main_db' to support the processing of requests based on the Direct, Random, and Customized hit strategies.

Upon the successful setup of all instances, the configuration of the MySQL cluster, and the completion of benchmarking, our system shifts focus to demonstrating its operational capabilities through the 'send_request.py' application. This application plays a crucial role in testing the

```python

@app.route("/fetch_direct", methods=["POST"])
def fetch_direct():
    app.logger.warning("Received direct fetch request")
    try:
        proxy_url = f"http://{proxy_dns}/fetch_direct"
        response = requests.post(proxy_url, json=request.get_json())
        return jsonify(response.json()), response.status_code
    except Exception as e:
        app.logger.error(f"Error in /fetch_direct: {e}")
        return jsonify({"error": str(e)}), 500


@app.route("/fetch_random", methods=["GET", "POST"])
def fetch_random():
    app.logger.warning("Received random fetch request")
    try:
        proxy_url = f"http://{proxy_dns}/fetch_random"
        response = requests.post(proxy_url, json=request.get_json())
        return jsonify(response.json()), response.status_code
    except Exception as e:
        app.logger.error(f"Error in /fetch_direct: {e}")
        return jsonify({"error": str(e)}), 500


@app.route("/fetch_customized", methods=["GET", "POST"])
def fetch_customized():
    app.logger.warning("Received customized fetch request")
    try:
        proxy_url = f"http://{proxy_dns}/fetch_customized"
        response = requests.post(proxy_url, json=request.get_json())
        return jsonify(response.json()), response.status_code
    except Exception as e:
        app.logger.error(f"Error in /fetch_direct: {e}")
        return jsonify({"error": str(e)}), 500

```

Figure 11: Trusted Host Routes

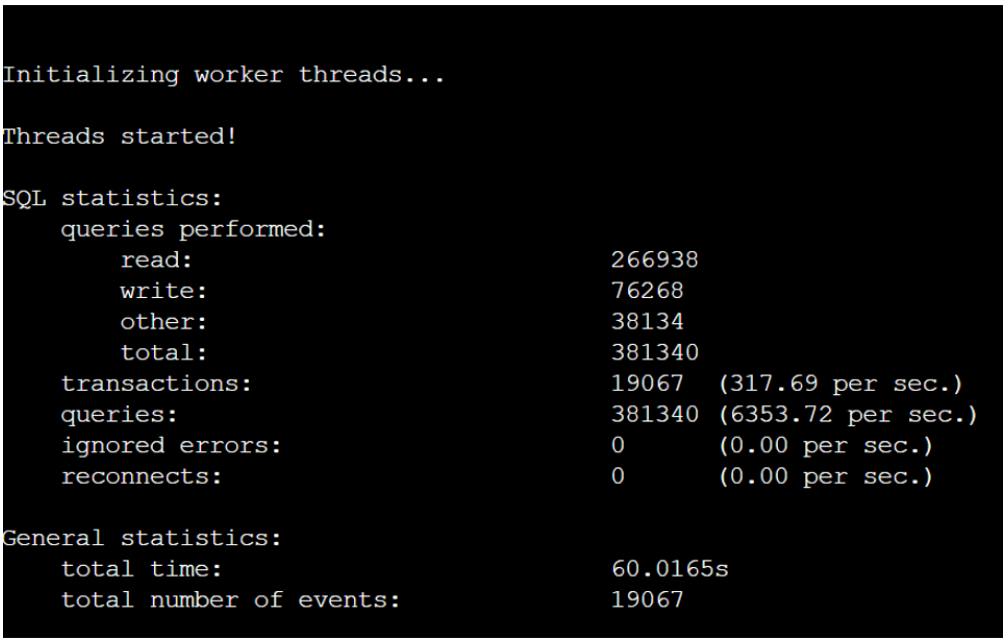


Figure 12: Benchmark results of standalone MySQL Server

entire data flow within our architecture. It initiates by sending 20 write requests to each of the three SQL tables: 'direct_table', 'random_table', and 'customized_table'. These requests, formatted as 'INSERT INTO table (column1, column2) VALUES ('column1_value', 'column2_value')', are designed to populate the tables with dummy values. They are routed through the Gatekeeper, which forwards them to the Trusted Host. The Trusted Host, in turn, passes these requests to the Proxy, and eventually, they reach the MySQL Cluster Manager, where the data is written into the respective tables. Each table receives 20 entries, ensuring a uniform data distribution for subsequent read operations. The system then proceeds to execute 20 read requests for each hit type: Direct, Random, and Customized. These requests, structured as 'SELECT * FROM direct_table LIMIT 1', aim to retrieve the first value from the respective tables. The response to these read requests is critical as it demonstrates the successful redirection and processing of requests through the entire chain—from the Gatekeeper to the Trusted Host, then to the Proxy, and finally to the appropriate node in the MySQL Cluster. The consistent retrieval of the expected values in the responses confirms that the system's data routing and processing mechanisms are functioning as intended.

## 6 Summary of results and instructions to run your code

Figure. 12 and Figure. 13 represents the benchmark results for the MySQL standalone while Figure. 14 and Figure. 15 represents the benchmark results for the MySQL cluster. The comparative analysis of the benchmark results between the MySQL standalone server and the MySQL Cluster reveals significant performance differences. Focusing first on the SQL statistics, the MySQL Cluster outperforms the standalone server in terms of overall throughput. The Cluster achieved higher read (266,938), write (76,268), and total transactions (381,340) compared to the standalone server's

```
Latency (ms):
         min:                                    4.91
         avg:                                   12.59
         max:                                   97.23
         95th percentile:                       16.71
         sum:                                239979.35

Threads fairness:
    events (avg/stddev):           4766.7500/3.34
    execution time (avg/stddev):   59.9948/0.00
```

Figure 13: Benchmark results of standalone MySQL Server

```
Initializing worker threads...

Threads started!

SQL statistics:
    queries performed:
        read:                            185640
        write:                           53040
        other:                           26520
        total:                           265200
    transactions:                        13260  (220.94 per sec.)
    queries:                             265200 (4418.73 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          60.0151s
    total number of events:              13260
```

Figure 14: Benchmark results of MySQL cluster

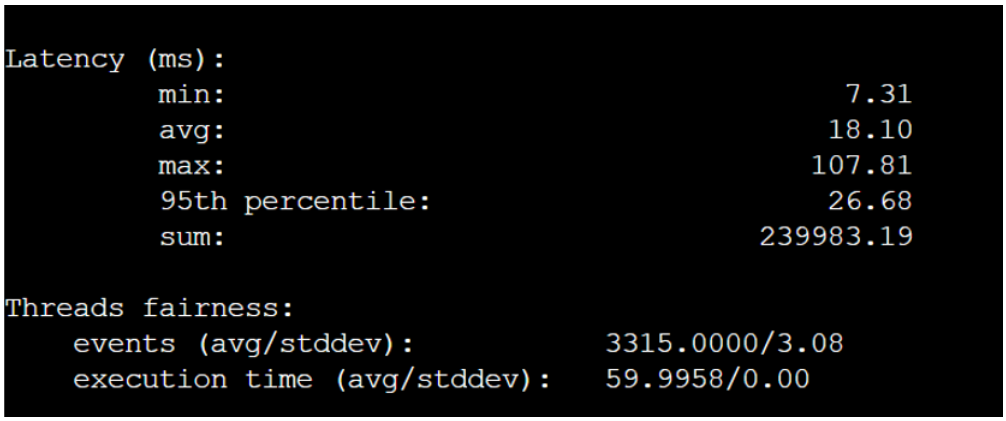


Figure 15: Benchmark results of MySQL cluster

read (185,640), write (53,040), and total transactions (265,200). This indicates a robust handling of data operations in the clustered environment. The transaction rate of the Cluster (317.69 per sec) is notably higher than that of the standalone server (220.94 per sec), suggesting better scalability and efficiency under load.

Analyzing the general statistics, both setups completed their tests in a similar timeframe (around 60 seconds), ensuring a fair comparison. However, the total number of events processed was higher for the Cluster (19,067) compared to the standalone server (13,260), underlining the Cluster's superior data handling capacity. For the latency statistics, the MySQL Cluster demonstrates a lower minimum latency (4.91 ms) and a lower average latency (12.59 ms) compared to the standalone server's minimum (7.31 ms) and average (18.10 ms) latencies. The maximum latency and the 95th percentile latency are also lower for the Cluster, indicating more consistent performance under stress. The threads fairness data, which shows the distribution of events and execution time across threads, reveals a similar pattern in both environments, indicating an efficient utilization of resources. Overall, the MySQL Cluster shows a marked advantage in handling larger volumes of transactions and maintaining lower latency, highlighting its suitability for scenarios requiring high scalability and performance.

In the designed system, the intention was to demonstrate how requests sent from the 'send_requests.py' script would be processed and logged across the Gatekeeper, Trusted Host, and Proxy instances on AWS EC2. However, due to the expiration of AWS EC2 access provided to the students for this semester, it's not possible to show the actual terminal outputs where these requests would be printed. Had the AWS EC2 instances been active, one would expect to see logs indicating how the requests are forwarded between the services, with all requests ideally responding with a 200 status code. This would confirm successful communication and execution of the 60 write requests. For the reading operations, the terminals of the three instances would display the results of direct, random, and customized hits. Specifically, the terminal for each instance would show the first element of the 'direct_table' being returned 20 times, followed by the first element of the 'random_table' 20 times, and finally, the first element of the 'customized_table' 20 times. This sequential retrieval and logging would provide a clear demonstration of the flow and handling of requests in the distributed system set up across the Gatekeeper, Trusted Host, and Proxy servers.

To run our project's code, users first need to install Python and Terraform on their local machine, ensuring they have a working AWS EC2 session. The next step involves retrieving AWS credentials, specifically the access key, secret access key, and session token. With these credentials at hand, users should navigate to the script folder and execute the 'bash run.sh' command. This script prompts for the AWS credentials; upon providing them, it automates the deployment of the infrastructure using Terraform. This process includes the creation of eight instances on AWS (Gatekeeper, Trusted Host, Proxy, MySQL Cluster Manager, and three MySQL Cluster Workers), which can be verified on the AWS dashboard. Following the successful setup, the 'run.sh' script also initiates the 'send_scripts.py' to send requests through the system. For a comprehensive understanding and verification of the request flow and system functionality, users are advised to manually log into the Gatekeeper, Trusted Host, and Proxy instances to observe the successful transmission and handling of these requests. This step-by-step procedure ensures a complete setup and operational verification of our cloud-based architecture.