Cloud Design Patterns: Implementing a DB Cluster

LOG8415E - Advanced Concepts of Cloud Computing

Kouamé Behouba Manassé
behouba-manasse.kouame@polymtl.ca

December 29, 2025

1 Introduction

Design patterns offer proven solutions to common challenges in computer science. In cloud computing, they provide solutions to common challenges in distributed systems. In this project, we have implemented two patterns for database access:

- **Proxy Pattern:** Separates read and write operations to improve scalability by distributing read load across worker nodes while ensuring write consistency through a single manager node.
- **Gatekeeper Pattern:** Provides a security layer that validates incoming requests before forwarding them to internal components.

The source code is available on GitHub[1] and a demo video demonstrating the system in action can be found here[2].

2 System Architecture

2.1 Architecture Overview

The system architecture consists of five EC2 instances organized in a layered design. External requests first hit the Gatekeeper, which authenticates users and validates queries before forwarding them to the Proxy. The Proxy then routes requests to the appropriate MySQL node based on the selected strategy, with writes going to the Manager and reads distributed across Workers.

[1] `https://github.com/behouba/log8415-final`

[2] `https://polymtlca-my.sharepoint.com/:f:/g/personal/behouba-manasse_kouame_etud_polymtl_ca/IgDnOU24kJZOTbQEgxaErNoAATlc-XEKPYCOkx-EdYTM7qs?e=khTX9u`

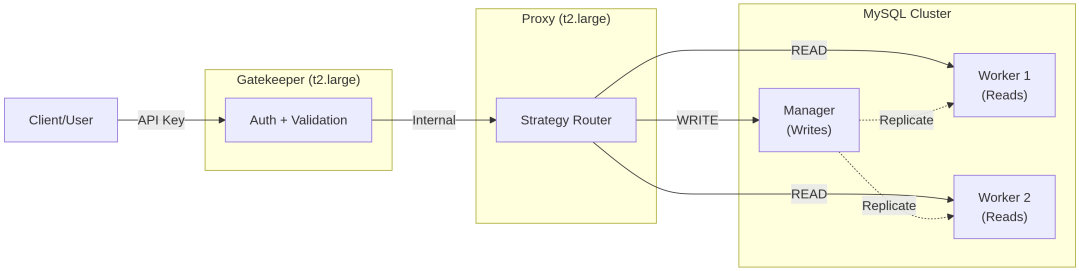

Figure 1: System Architecture: Request flow from user through Gatekeeper and Proxy to MySQL Cluster

2.2 Components

The table below presents the type of ec2 instances, their specs, roles, and network accessibility.

Table 1: EC2 Instances

Component	Instance Type	Role	Access
Gatekeeper	t2.large	Authentication, Validation	Public internet
Proxy	t2.large	Routing, Load Balancing	Internal
Manager	t2.micro	MySQL Master (Writes)	Internal
Worker 1	t2.micro	MySQL Replica (Reads)	Internal
Worker 2	t2.micro	MySQL Replica (Reads)	Internal

3 MySQL Standalone Setup and Sysbench Benchmarking

3.1 MySQL Installation

Each of the three MySQL instances is provisioned using EC2 user-data scripts that automatically perform the following actions:

1. Install MySQL Server 8.0 on Ubuntu 24.04 LTS
2. Download and import the Sakila sample database
3. Create database users with appropriate privileges
4. Configure MySQL for remote access connections

These step actions are handled by the bash script inside sscripts/install_mysql.sh.

3.2 Sysbench Benchmarking

Sysbench is used to verify the MySQL installation and measure baseline performance. The OLTP read-only workload tests database query performance.

```
43  # Run Sysbenh
44  sudo sysbench /usr/share/sysbench/oltp_read_only.lua \
45    --mysql-db=$DB_NAME \
46    --mysql-user=$DB_USER \
47    --mysql-password=$DB_PASS \
48    prepare
49
50  # Run Sysbench benchmark test
51  sudo sysbench /usr/share/sysbench/oltp_read_only.lua \
52    --mysql-db=$DB_NAME \
53    --mysql-user=$DB_USER \
54    --mysql-password=$DB_PASS \
55    run > $SYSBENCH_RESULTS
```

Figure 2: Sysbench Commands

3.3 Sysbench Results

The sysbench benchmark validates that MySQL is correctly installed and performing as expected on each instance:

```
final_project on  master [x!?] via 🐍v3.13.11 on ☁️  (us-east-1)
❯ ssh -i cluster-key.pem -o StrictHostKeyChecking=no ubuntu@34.230.35.28 "cat /home/ubuntu/sysbench results.txt"
Initializing random number generator from current time


Initializing worker threads...

Threads started!

SQL statistics:
    queries performed:
        read:                            105672
        write:                           0
        other:                           15096
        total:                           120768
    transactions:                        7548   (754.57 per sec.)
    queries:                             120768 (12073.13 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          10.0014s
    total number of events:              7548

Latency (ms):
         min:                                    0.85
         avg:                                    1.32
         max:                                   22.20
         95th percentile:                        2.30
         sum:                                 9979.54

Threads fairness:
    events (avg/stddev):           7548.0000/0.00
    execution time (avg/stddev):   9.9795/0.00
```

Figure 3: Sysbench Results (Manager instance)

The results confirm that the MySQL instances are operational and ready to handle the expected workload.

4 MySQL Replication Setup

4.1 Replication Architecture

MySQL replication is configured in a master-slave pattern where:

- The **Manager** node acts as the source (master)
- The **Worker** nodes act as replicas (slaves)
- Changes on the Manager are automatically propagated to Workers

4.2 Replication Configuration

The replication setup involves configuring each node with appropriate settings:

```
def setup_master(manager):
    ip = manager.public_ip_address
    print(f"Setting up master on {manager.id} with IP {ip}...")

    create_user_cmd = f"""
    sudo mysql -e "CREATE USER IF NOT EXISTS '{REP_USER}'@'%' IDENTIFIED WITH mysql_native_password BY '{REP_PASS}';"
    sudo mysql -e "GRANT REPLICATION SLAVE ON *.* TO '{REP_USER}'@'%';"
    sudo mysql -e "CREATE USER IF NOT EXISTS '{DB_USER}'@'%' IDENTIFIED BY '{DB_PASS}';"
    sudo mysql -e "GRANT ALL PRIVILEGES ON {DB_NAME}.* TO '{DB_USER}'@'%';"
    sudo mysql -e "FLUSH PRIVILEGES;"
    """

    execute_ssh_command(manager, create_user_cmd)

    print(" Getting Binary Log Coordinates...")

    status_output = execute_ssh_command(manager, "sudo mysql -e 'SHOW MASTER STATUS\\G'")

    if not status_output:
        print("Failed to get master status.")
        sys.exit(1)

    file_name = None
    position = None

    lines = status_output.split('\n')
    for line in lines:
        if 'File:' in line:
            file_name = line.split(':')[1].strip()
        elif 'Position:' in line:
            position = line.split(':')[1].strip()
```

Figure 4: Manager (Master) Configuration

```
def setup_slave(worker, master_ip, log_file, log_pos):
    ip = worker.public_ip_address
    print(f"Setting up slave on {worker.id} with IP {ip}...")

    # Create users on worker for proxy access
    create_user_cmd = f"""
    sudo mysql -e "CREATE USER IF NOT EXISTS '{DB_USER}'@'%' IDENTIFIED BY '{DB_PASS}';"
    sudo mysql -e "GRANT ALL PRIVILEGES ON {DB_NAME}.* TO '{DB_USER}'@'%';"
    sudo mysql -e "FLUSH PRIVILEGES;"
    """
    execute_ssh_command(worker, create_user_cmd)

    # Configure replication
    change_master_cmd = f"""
    sudo mysql -e "CHANGE MASTER TO MASTER_HOST='{master_ip}', MASTER_USER='{REP_USER}', MASTER_PASSWORD='{REP_PASS}', MASTER_LOG_FILE='{log_file}', MASTER_LOG_POS={log_pos};"
    sudo mysql -e "START SLAVE;"
    """

    execute_ssh_command(worker, change_master_cmd)
```

Figure 5: Worker (Slave) Configuration

4.3 Replication Verification

We verify the replication status by checking that both IO and SQL threads are running:

```python
def verify_replication(instances):
    print("Verifying replication status...")

    ok = True

    for instance in instances:
        ip = instance.public_ip_address

        cmd = 'sudo mysql -e "SHOW SLAVE STATUS\\G" | grep "Running: Yes"'
        res = execute_ssh_command(instance, cmd)

        if res and "IO_Running: Yes" in res and "SQL_Running: Yes" in res:
            print(f" Instance {instance.id} replication is running.")
        else:
            print(f" Instance {instance.id} replication is NOT running.")
            ok = False
            debug = execute_ssh_command(instance, 'sudo mysql -e "SHOW REPLICA STATUS\\G"')
            print(f" Debug info:\\n{debug}")

    return ok
```

Figure 6: Verify Replication Status

5 Proxy Pattern Implementation

5.1 Pattern Description

The Proxy pattern improves scalability by separating read and write operations:

- **Write Operations** (INSERT, UPDATE, DELETE): Are sent exclusively to the Manager node to maintain data consistency
- **Read Operations** (SELECT): Distributed to Worker nodes to balance the load

5.2 Routing Strategies

The Proxy implements three routing strategies for read operations:

5.2.1 Direct Hit Strategy

All queries (both read and write) are sent directly to the Manager node. This serves as a baseline for comparison.

```python
# Always hit the manager
def strategy_direct_hit():
    return MANAGER_IP
```

Figure 7: Direct Hit Strategy

5.2.2 Random Strategy

Read queries are randomly distributed among available Worker nodes, providing simple load balancing.

```python
# Randomly choose between manager and workers
def strategy_random():
    if not WORKER_IPS:
        return MANAGER_IP
    return random.choice(WORKER_IPS)
```

Figure 8: Random Strategy

5.2.3 Customized Strategy

Read queries are sent to the Worker with the lowest network latency, measured using ICMP ping.

```python
# Select worker with lowest ping time
def strategy_customized():
    if not WORKER_IPS:
        return MANAGER_IP

    best_ip = WORKER_IPS[0]
    best_time = 9999

    for ip in WORKER_IPS:
        t = ping_time(ip)
        print(f"Ping time to {ip}: {t} ms")
        if t < best_time:
            best_time = t
            best_ip = ip
    return best_ip
```

Figure 9: Customized (Ping-Based) Strategy

5.3 Query Classification

The Proxy determines query type by examining the SQL statement:

```python
@app.route('/query', methods=['POST'])
def proxy_query():
    data = request.json
    sql = data.get('query', '').strip()
    strategy = data.get('strategy', 'direct_hit')

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

    # Determine target IP based on strategy
    is_read = sql.lower().startswith("select")

    target_ip = MANAGER_IP
    node_role = "Manager"

    if is_read:
        if strategy == "random":
            target_ip = strategy_random()
            node_role = "Worker (Random)"
        elif strategy == "customized":
            target_ip = strategy_customized()
            node_role = "Worker (Ping Optimized)"
        else:
            # Default to direct hit
            target_ip = MANAGER_IP
            node_role = "Manager (Direct Hit)"
    else:
        # Writes always go to manager
        target_ip = MANAGER_IP
        node_role = "Manager (Write)"
```

Figure 10: Query Classification Logic

6 Gatekeeper Pattern Implementation

6.1 Pattern Description

The Gatekeeper pattern provides a security layer that:

- Acts as the only internet-facing component
- Validates authentication credentials
- Sanitizes and validates incoming queries
- Forwards valid requests to the Trusted Host (Proxy)

6.2 Authentication

All requests must include a valid API key in the HTTP header. The API key is automatically generated during setup.

```
27  @app.route('/query', methods=['POST'])
28  def gateway():
29      # Check API key authentication
30      api_key = request.headers.get('X-API-Key')
31      if api_key != API_KEY:
32          return jsonify({"error": "Unauthorized - Invalid or missing API key"}), 401
33
```

Figure 11: API Key Authentication

6.3 Query Validation

The Gatekeeper blocks potentially dangerous SQL operations:

```
12
13  BLOCKED_PATTERNS = [
14      r'\bDROP\s+TABLE\b',
15      r'\bDROP\s+DATABASE\b',
16      r'\bDELETE\s+FROM\s+\w+\s*;?\s*$',  # match DELETE without WHERE
17      r'\bTRUNCATE\b',
18  ]
19
20  def is_query_safe(query):
21      query_upper = query.upper()
22      for pattern in BLOCKED_PATTERNS:
23          if re.search(pattern, query_upper, re.IGNORECASE):
24              return False
25      return True
```

Figure 12: Query Validation

6.4 Request Flow

1. User sends request to Gatekeeper with API key
2. Gatekeeper validates API key (401 if invalid)
3. Gatekeeper validates query safety (403 if dangerous)
4. Valid requests are forwarded to Proxy
5. Proxy routes to appropriate MySQL node
6. Response returns through the same path

7 Security Implementation

7.1 Network Security

The implementation follows security best practices:

Table 2: Security Group Rules

Port	Source	Component	Purpose
22	0.0.0.0/0	All	SSH access
5000	0.0.0.0/0	Gatekeeper	API endpoint
5000	Gatekeeper SG	Proxy	Internal communication
3306	Cluster SG	MySQL nodes	Database access

7.2 Trusted Host Protection

The Proxy (Trusted Host) is protected by:

- Removing public access to port 5000
- Only allowing connections from Gatekeeper security group
- No direct internet exposure

```
32  # Remove public access to port 5000
33  try:
34      ec2_client.revoke_security_group_ingress(
35          GroupId=cluster_sg_id,
36          IpPermissions=[
37              {'IpProtocol': 'tcp', 'FromPort': 5000, 'ToPort': 5000, 'IpRanges': [{'CidrIp': '0.0.0.0/0'}]}
38          ]
39      )
40      print("Removed public access to port 5000")
41  except Exception as e:
42      print(f"Could not remove public access (may not exist): {e}")
43
44  # Add access only from gatekeeper
45  try:
46      ec2_client.authorize_security_group_ingress(
47          GroupId=cluster_sg_id,
48          IpPermissions=[
49              {'IpProtocol': 'tcp', 'FromPort': 5000, 'ToPort': 5000,
50               'UserIdGroupPairs': [{'GroupId': gatekeeper_sg_id}]}
51          ]
52      )
53      print("Added port 5000 access only from gatekeeper")
54  except Exception as e:
55      print(f"Error adding gatekeeper access: {e}")
```

Figure 13: Securing the Proxy

8 Cluster Benchmarking

8.1 Benchmark Methodology

The benchmarking script sends 1,000 read requests and 1,000 write requests for each of the three routing strategies, totaling 6,000 requests. The benchmark uses:

- **Read Query:** `SELECT * FROM actor LIMIT 1`
- **Write Query:** `UPDATE actor SET last_update = NOW() WHERE actor_id = 1`
- **Concurrency:** 10 parallel workers using ThreadPoolExecutor

8.2 Benchmark Results

Table 3: Benchmark Results: 1000 Reads + 1000 Writes per Strategy

Strategy	Read Success	Write Success	Read Throughput	Write Throughput
Direct Hit	1000/1000	1000/1000	114.98 req/s	112.87 req/s
Random	1000/1000	1000/1000	115.97 req/s	114.26 req/s
Customized	1000/1000	1000/1000	109.92 req/s	113.18 req/s

8.3 Results Analysis

Success Rate: All 6,000 requests completed successfully with 0 failures, demonstrating the reliability of the implementation.

Read Throughput:

- **Direct Hit** (114.98 req/s): Baseline performance with all reads going to Manager
- **Random** (115.97 req/s): Slightly higher throughput due to load distribution
- **Customized** (109.92 req/s): Lower throughput certainly due to ping measurement overhead before each request

Write Throughput: Consistent across all strategies (113 req/s) as writes always go to the Manager regardless of strategy.

Observations:

1. The Random strategy provides the best read throughput for this workload
2. The Customized strategy incurs overhead from latency measurements
3. Write performance is independent of routing strategy

9 Conclusion

In this projet we implemented a distributed MySQL cluster with Proxy and Gatekeeper cloud design patterns on AWS EC2. The work completed include:

1. **MySQL Cluster:** Three t2.micro instances with master-slave replication, verified with sysbench benchmarks showing 384 transactions/second baseline performance.
2. **Proxy Pattern:** Implemented three routing strategies (direct hit, random, customized) that correctly separate read and write operations, achieving 110-116 requests/second throughput.
3. **Gatekeeper Pattern:** Implemented authentication via API keys and query validation that blocks dangerous operations, providing a secure entry point to the system.
4. **Security:** The Proxy (Trusted Host) is protected from direct internet access, with all external traffic routed through the Gatekeeper.
5. **Automation:** Complete Infrastructure as Code implementation using boto3, enabling reproducible deployments.
6. **Reliability:** 100% success rate across 6,000 benchmark requests validates the correctness of the implementation.