Polytechnique Montréal

Département de génie informatique et génie logiciel

LOG8415E - Advanced Concepts of Cloud Computing

LABORATORY REPORT 1

Cloud Design Patterns: Implementing a DB Cluster

Student Name	Student ID
1. Mehdi Ougadi	2051559

Lecturer in charge:

Vahid Majdinasab

Submission Date : 31/12/2025

Contents

1 Introduction

This laboratory assignment explores advanced cloud computing concepts through the implementation of a distributed MySQL database cluster on Amazon Web Services (AWS) with integrated security patterns. The primary objectives are to gain hands-on experience with MySQL replication, cloud design patterns including the Proxy and Gatekeeper patterns, and Infrastructure as Code (IaC) for automated cloud resource management while understanding how security-focused architectural patterns enhance system resilience and minimize attack surfaces.

My architecture consists of a three-tier system organized into public and private subnets for security isolation. The database layer contains 3 EC2 t2.micro instances configured in a master-subordinate replication setup where one manager node handles all write operations and two worker nodes distribute read workloads. The middle tier implements the Proxy pattern using a t2.large instance that acts as a trusted host, routing read queries to workers and write queries to the manager while implementing three distinct forwarding strategies: direct hit, random selection and customized ping-based routing. The public-facing tier deploys the Gatekeeper pattern through a t2.large instance that validates and authenticates all incoming requests before forwarding them to the trusted proxy, effectively creating a security buffer between external users and the internal database infrastructure. Each database instance hosts the Sakila sample database for consistent benchmarking, with MySQL replication ensuring data consistency across all nodes.

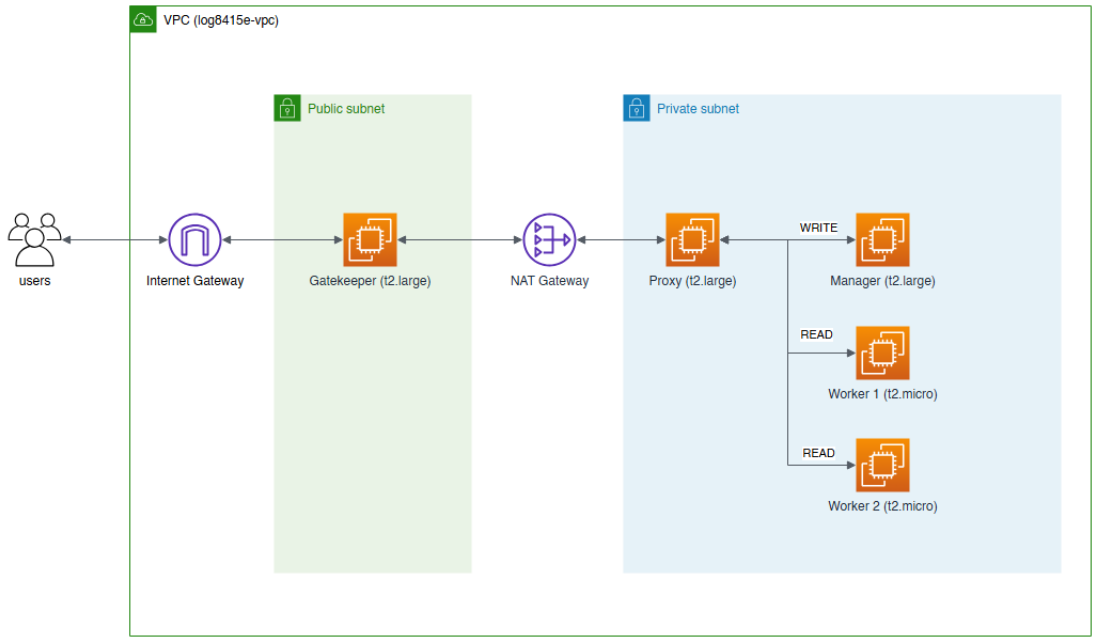

Figure 1: AWS Cloud Patterns Setup

This setup allows us to benchmark different query routing strategies under identical conditions while demonstrating how cloud design patterns improve both security and scalability in distributed database systems. The entire infrastructure is deployed using Python and AWS Boto3 SDK.

2 Gatekeeper Pattern

The Gatekeeper pattern implements a critical security layer that minimizes the attack surface by serving as the sole internet-facing component of my architecture. This pattern follows the principle of defense in depth, where the gatekeeper validates all incoming requests before forwarding them to internal trusted components. In my implementation, the Gatekeeper acts as the first line of defense, receiving all external API requests and performing authentication, authorization and input validation before communication with the Proxy.

The architecture separates concerns into two distinct roles: the Gatekeeper instance deployed in the public subnet with direct internet access and the Proxy instance operating in the private subnet with no direct external exposure. This separation ensures that even if the Gatekeeper is compromised, attackers cannot directly access the database cluster as the Proxy and database nodes remain isolated within the private network. The Gatekeeper validates API keys, sanitizes SQL queries to prevent injection attacks and blocks dangerous operations such as DROP TABLE or DELETE without WHERE clauses before forwarding legitimate requests to the Proxy for execution.

2.1 Security Group Configuration

The Gatekeeper security configuration implements strict ingress and egress rules to control network traffic flow. Ingress rules permit HTTP (port 80), HTTPS (port 443) and API access (port 8080) from the internet to enable external user connectivity, along with SSH access (port 22) for administrative purposes. Egress rules allow HTTPS (port 443) and HTTP (port 80) traffic for package updates and forwarding of validated requests to the Proxy on port 5000 within the private subnet. This configuration ensures the Gatekeeper can communicate with both external users and internal components while maintaining security boundaries, with port 8080 serving as the public-facing API endpoint and port 5000 used exclusively for internal communication with the Proxy.

2.2 Implementation and Deployment

The Gatekeeper implementation utilizes a Flask-based web service deployed on a t2.large EC2 instance to handle incoming API requests with sufficient compute resources for request validation and forwarding. The deployment process is fully automated through a user data script that installs Python dependencies, configures the Flask application with the Proxy's private IP address and establishes the service as a systemd daemon for automatic startup and restart capabilities.

The Flask application implements three critical security functions. First, it validates API keys through the X-API-Key header to ensure only authorized clients can access the system. Second, it performs query sanitization using regular expressions to detect and block dangerous SQL operations such as DROP TABLE, TRUNCATE, or DELETE statements without WHERE clauses that could compromise data integrity. Third, it forwards validated requests to the Proxy via HTTP POST to the internal endpoint, effectively acting as a security

gateway that protects the trusted internal infrastructure from malicious external requests.

2.3 Request Flow and Validation

When a user sends a request to the Gatekeeper, the following validation sequence occurs. The Gatekeeper first checks for the presence and validity of the API key in the request headers, returning a 401 Unauthorized error if authentication fails. Next, it extracts the SQL query from the request body and applies regex pattern matching against a list of dangerous operations, blocking requests that match these patterns with a 403 Forbidden response. For requests that pass both validation checks, the Gatekeeper forwards the complete request payload to the Proxy's /query endpoint, including the query content and routing strategy parameters. The Proxy processes the request, executes it against the appropriate database node and returns the results through the Gatekeeper back to the original client, ensuring all database interactions remain mediated through this secure validation layer.

This implementation demonstrates how the Gatekeeper pattern significantly reduces attack surface by consolidating security checks in a single internet-facing component while keeping critical infrastructure isolated in private networks. The pattern provides defense in depth, input validation and clear separation of security responsibilities that align with cloud security best practices for distributed database systems.

3 Proxy Pattern

The Proxy pattern improves scalability and performance by separating read and write operations in a MySQL cluster architecture. This pattern acts as an intermediary layer between client requests and the database cluster, controlling all access to database nodes and ensuring that no component can directly access the database instances. In my implementation, the Proxy serves as the Trusted Host, receiving validated requests from the Gatekeeper and routing them to the appropriate database nodes based on query type and selected forwarding strategy.

The architecture implements a master-subordinate replication topology where the manager node handles all write operations (INSERT, UPDATE, DELETE, CREATE, DROP) and subsequently replicates changes to worker nodes through MySQL's built-in binary log replication. Worker nodes are configured in read-only mode and handle all read operations (SELECT, SHOW, DESCRIBE, EXPLAIN), distributing the read workload across multiple instances to improve query performance and system throughput. This separation ensures data consistency while maximizing horizontal scalability for read-heavy database workloads typical in web applications and analytical systems.

3.1 Security Group Configuration

The Proxy security configuration implements restrictive ingress and egress rules to maintain its role as an internal trusted component. Ingress rules permit

API access on port 5000 exclusively from the public subnet CIDR (10.0.1.0/24) where the Gatekeeper resides, ensuring only validated requests can reach the Proxy. Egress rules allow HTTPS (port 443) and HTTP (port 80) for package updates, MySQL access (port 3306) to all database nodes within the private subnet (10.0.2.0/24) for query execution and ICMP traffic for ping-based health monitoring of worker nodes. This configuration ensures the Proxy remains isolated from direct internet access while maintaining necessary connectivity to both the Gatekeeper and the database cluster.

3.2 Implementation and Deployment

The Proxy implementation utilizes a Flask-based web service deployed on a t2.large EC2 instance with enhanced compute resources to handle concurrent database connections and implement intelligent routing logic. The deployment process is fully automated through a user data script that installs Python dependencies (Flask, PyMySQL, requests), MariaDB client tools for database connectivity and network utilities for health monitoring capabilities.

The Flask application implements sophisticated query routing logic through multiple components. The `is_read_query()` function analyzes incoming SQL statements by examining query keywords to classify operations as either read or write, ensuring correct routing decisions. The `execute_query()` function establishes MySQL connections using PyMySQL, executes queries against the target host and returns structured responses including query results for reads or affected row counts for writes. A background health monitoring thread continuously measures worker node latency using ICMP ping requests, updating a shared health status dictionary every 10 seconds to enable intelligent routing decisions for the customized strategy.

3.3 Forwarding Strategies

The Proxy implements three distinct forwarding strategies through conditional logic in the `handle_query()` endpoint function. The strategy parameter is extracted from the incoming JSON request payload and defaults to 'random' if not specified. The implementation uses a series of conditional branches to determine the target host based on both query type and selected strategy.

3.3.1 Direct Hit Implementation

The direct hit strategy is implemented through a simple conditional check where `strategy == 'direct'` immediately sets the target host to `DB_CONFIG['manager_host']` regardless of query type. This bypasses all worker selection logic and routing, creating a straightforward path where the `handle_query()` function assigns the manager's IP address directly to the host variable. The implementation requires no additional helper functions or state management, making it the most computationally efficient routing option with minimal execution overhead. When this strategy is selected, the subsequent `execute_query()` call always connects to the manager node, whether processing read or write operations.

3.3.2 Random Strategy Implementation

The random strategy implementation leverages Python's `random.choice()` function from the standard library to select a target worker from the `worker_hosts` list stored in the database configuration dictionary. The implementation occurs within the read query conditional branch, where `random.choice(DB_CONFIG['worker_hosts'])` provides uniform probability distribution across all available workers. This approach requires no pre-computation, state tracking or health monitoring infrastructure. The random selection executes in constant time O(1) with minimal CPU overhead, making it suitable for high-throughput environments. The implementation treats all workers as equivalent candidates, relying on statistical distribution over many requests to achieve balanced load across the worker pool.

3.3.3 Customized Strategy Implementation

The customized strategy implementation relies on a dedicated background monitoring thread and shared state management using Python's threading primitives. The `background_health_monitor()` function runs as a daemon thread, continuously executing a monitoring loop that invokes `get_ping_time()` for each worker host every 10 seconds. The `get_ping_time()` function uses Python's `subprocess.run()` to execute ICMP ping commands with a 3-packet burst and 5-second timeout, parsing the standard output to extract average round-trip time from the statistics line. Health data is stored in the `worker_health` dictionary, protected by a `threading.Lock()` to prevent race conditions between the monitoring thread and request handler threads.

When a read query arrives with the customized strategy, the implementation acquires the health lock and uses Python's `min()` function with `key=worker_health.get` to identify the worker with the lowest ping latency. This operation executes in O(n) time where n is the number of workers, acceptable given the small worker pool size. The implementation includes a fallback mechanism where an empty health dictionary triggers default routing to the manager node, ensuring service continuity during initialization periods or complete worker failure scenarios. This graceful degradation prevents request failures while the monitoring thread collects initial health metrics.

3.4 Query Execution Flow

The complete query execution flow begins when the Gatekeeper forwards a validated request to the Proxy's `/query` endpoint. The `handle_query()` function extracts the SQL query string and strategy parameter from the JSON payload using Flask's `request.get_json()` method. The `is_read_query()` function is immediately invoked to classify the operation by checking if the uppercase query string starts with any write keywords or read keywords. Write operations bypass strategy selection logic entirely and set the target host directly to the manager node.

For read operations, the implementation enters a conditional cascade that evaluates the strategy parameter. The direct hit branch sets the manager as the target, the customized branch executes the health-based selection with lock ac-

quisition and the default branch performs random selection. Once the target host is determined, the `execute_query()` function establishes a PyMySQL connection with dictionary cursor configuration, executes the query and returns a structured response containing success status, query results or affected row counts and the processing host's IP address. The host metadata enables verification of correct routing behavior during benchmarking and operational monitoring. All database connections are properly closed in the finally block to prevent connection pool exhaustion and exceptions are caught to return structured error responses with 500 status codes for failed operations.

This implementation demonstrates how the Proxy pattern effectively separates read and write workloads while providing flexible routing strategies that can be optimized for different performance requirements. The pattern enables horizontal scaling of read operations, reduces load on the manager node and maintains data consistency through MySQL replication, aligning with distributed database best practices for high-availability systems.

4 Benchmark

4.1 MySQL with sysbench

Sysbench benchmarks using the `oltp_read_only` workload revealed significant performance variation across nodes. Worker-1 achieved the highest throughput at 884.3 transactions per second, establishing itself as the most performant instance. Worker-2 recorded 580.0 TPS, representing approximately 66% of Worker-1's capacity, while the manager node showed the lowest performance at 552.9 TPS. These substantial differences result from EC2 t2.micro instance variability due to shared underlying hardware resources and the burstable performance characteristics of the t2 family. The manager node's lower throughput is additionally impacted by replication overhead, as it must maintain binary logs for data synchronization with worker nodes. Worker-1's 60% performance advantage over the manager suggests favorable hardware placement and indicates it should theoretically be the preferred target for the customized routing strategy.

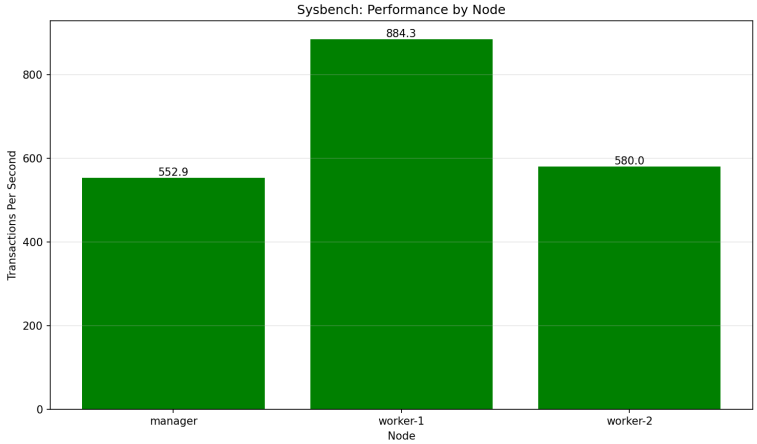

Figure 2: Sysbench Performance Comparison Across Database Nodes

4.2 Cluster

Testing with 1000 read and 1000 write requests per strategy yielded remarkably similar results. The direct strategy averaged 71.0ms for reads and 75.5ms for writes, while the random strategy recorded 71.2ms and 76.1ms respectively. The customized strategy showed 71.9ms for reads and 76.6ms for writes. The minimal variance of less than 1.5% across all strategies indicates that network latency dominates response times rather than database processing capacity. Write operations consistently showed 5-7% overhead compared to reads due to transaction logging, index updates and disk synchronization requirements. The customized strategy's unexpected higher latency stems from several factors: ping measurements do not accurately reflect actual MySQL query response times, the 10-second monitoring interval creates temporal mismatches with real-time network conditions and the additional overhead of lock acquisition and dictionary lookups adds processing time. Despite the similar response times, the strategies successfully distributed read workload across workers, demonstrating proper implementation of the Proxy pattern and enabling horizontal scalability under higher concurrent loads.

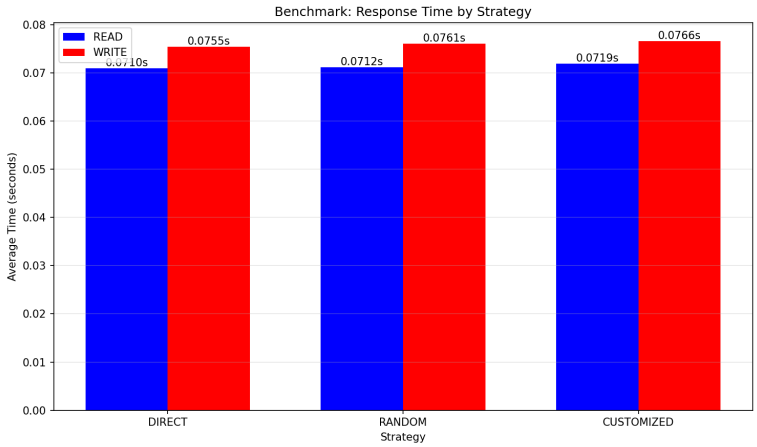

Figure 3: Response Time Analysis by Forwarding Strategy

5 Conclusion

This assignment successfully implemented a distributed MySQL cluster with Gatekeeper and Proxy patterns, fully automated using Python and Boto3. The architecture achieved workload separation through master-subordinate replication and security isolation through defense-in-depth. Benchmark results revealed network overhead as the primary bottleneck rather than database processing, with ¡ 1.5% performance variance across routing strategies at moderate load. The customized strategy's ping-based approach proved insufficient due to measurement methodology limitations. Sysbench results showing 60% variance across identical instances highlight the unpredictability of cloud infrastructure performance. The architecture successfully separates read/write operations, with all writes correctly routed to the manager for data consistency while reads distribute across workers. The Gatekeeper effectively minimizes attack surface through centralized security validation. Production deployment would require connection pooling, comprehensive health monitoring including replication lag, automatic failover mechanisms, and TLS encryption. The implementation demonstrates horizontal scalability potential that becomes valuable under high concurrent workloads.

Video and instructions to run the code can be found in the README file.