LOG8415E – Final Assignment

Cloud Design Patterns (Gatekeeper + Proxy)

Lounes Sadmi
Student ID: 2113288

December 30, 2025

Contents

1 Introduction

The goal of this assignment is to design and implement a secure and scalable database access layer on AWS using cloud design patterns. The key idea is that the database nodes should not be exposed to the Internet. Instead, external clients send requests to a single public component (the Gatekeeper), which then forwards valid requests to internal services. Behind the Gatekeeper, the Proxy is responsible for routing SQL queries to the correct database node: reads can go to replicas (workers), while writes must go to the manager.

2 Overall Architecture

2.1 Components

The deployed system contains five EC2 instances:

- **Gatekeeper**: a small HTTP API that is publicly reachable and enforces an API key.
- **Proxy**: an internal HTTP API that decides where each SQL query must run.
- **DB Manager**: MySQL primary node. Receives *all writes.*
- **DB Worker 1**: MySQL replica. Serves *read traffic.*
- **DB Worker 2**: MySQL replica. Serves *read traffic.*

2.2 Network isolation and security groups

Only the Gatekeeper accepts inbound HTTP from the Internet (port 80). The Proxy accepts inbound traffic only from the Gatekeeper (port 5000). The database nodes accept MySQL traffic only from the Proxy (port 3306), and also allow port 3306 among themselves to support replication. SSH is limited to my administrative public IP.

3 Implementation Details

3.1 Database synchronization and replication

3.1.1 Manager configuration

On the manager, MySQL is configured with:

- `server-id`, `log_bin`, and `binlog_format=ROW`
- Global Transaction Identifier (GTID): `gtid_mode=ON` and `enforce_gtid_consistency=ON`
- `log_replica_updates=ON` so the replication stream is consistent

The manager creates three users:

- `repl@'%'`: replication user with `REPLICATION SLAVE` privilege
- `app@'%'`: application user (used by the Proxy to run SQL)
- `sb@localhost`: benchmarking user (used by Sysbench)

3.1.2 Worker configuration

Each worker also has `gtid_mode=ON`, `enforce_gtid_consistency=ON`, and a unique `server-id`. Workers are configured as read-only replicas with `read_only=ON`. Replication is started with:

```
CHANGE REPLICATION SOURCE TO
  SOURCE_HOST='<manager-private-ip>',
  SOURCE_USER='repl',
  SOURCE_PASSWORD='ReplPass123!',
  SOURCE_AUTO_POSITION=1;
START REPLICA;
```

To validate replication during the demo, I use:

```
sudo mysql -e 'SHOW SLAVE STATUS\G' | egrep \
'Slave_IO_Running|Slave_SQL_Running|Seconds_Behind_Master|Last_IO_Error|Last_SQL_Error'
```

The expected output is `Slave_IO_Running: Yes`, `Slave_SQL_Running: Yes`, and `Seconds_Behind_Master: 0`.

3.2 Proxy service (proxy.py)

The Proxy is a Flask web service listening on port 5000. It exposes:

- `GET /health`: returns `ok`
- `POST /query`: receives a JSON body with `sql`, `mode`, and an optional `request_id`

3.2.1 Read vs write detection

The function `is_read(sql)` classifies a query as read if it starts with `SELECT`, `SHOW`, or `DESCRIBE`. Everything else is treated as a write.

3.2.2 Routing strategy

The routing behavior is:

- `mode=direct`: always run on the manager (for both reads and writes).
- `mode=random`: if it is a read, pick a random worker; if it is a write, use the manager.
- `mode=ping`: if it is a read, pick the worker with the best ICMP ping time; if it is a write, use the manager.

This implements the Proxy design pattern because the client never connects directly to any database node. The Proxy is the decision point that hides the internal topology.

3.2.3 Executing SQL

`run_mysql(target, sql)` opens a MySQL connection with `autocommit=True`, executes the query, and returns rows when the query produces a result set. The response includes metadata such as the request identifier, the mode, the query type, and which node executed the query.

3.3 Gatekeeper service (gatekeeper.py)

The Gatekeeper is the only service exposed publicly. It listens on port 80 and provides:

- `GET /health`: returns `ok`
- `POST /query`: checks the `X-API-Key` header and forwards the request to the Proxy

If the API key is missing or incorrect, the Gatekeeper responds with `401 Unauthorized`. If the key is correct, it forwards the JSON body to the Proxy and returns the Proxy response to the client. This implements the Gatekeeper design pattern by enforcing a policy at the system boundary.

3.4 Orchestration (provision.py)

The orchestration script provisions the full architecture in AWS using Boto3:

- Finds the latest Ubuntu 22.04 AMI from Canonical (owner `099720109477`).
- Uses the default VPC and one default subnet.
- Creates three security groups (Gatekeeper, Proxy, Database) with least-privilege inbound rules.
- Launches all instances with cloud-init user data that installs dependencies and configures services.
- Installs the Git repository code on Proxy and Gatekeeper, then starts them as systemd units.

A key point is that the Gatekeeper environment file (`/etc/log8415e_gatekeeper.env`) is created using the Proxy *private* IP, so the internal calls do not depend on public networking and remain inside the VPC.

4 Benchmarking and Results

4.1 Baseline: Sysbench OLTP

I ran Sysbench locally on each worker node against its local MySQL using the `sb` user. With 2 tables of 100,000 rows, 4 threads, and a 30-second duration:

- Worker 1 achieved about **185.61 transactions per second**.
- Worker 2 achieved about **232.16 transactions per second**.

This gives a rough baseline for single-node database throughput without the HTTP layer.

4.2 End-to-end latency benchmark (1,000 operations)

I also measured end-to-end latency by sending HTTP requests to the Gatekeeper. The benchmark performs 1,000 writes and 1,000 reads in each mode.

Table 1: End-to-end latency summary for 1,000 operations (milliseconds)

Mode	Type	Average	P50	P95	Max
direct	write	121.92	120.52	128.79	370.33
direct	read	124.20	120.13	144.38	421.72
random	write	121.54	120.97	126.94	178.12
random	read	122.53	120.25	131.48	456.01
ping	write	122.12	121.11	128.53	396.85
ping	read	204.85	119.50	131.94	2246.51

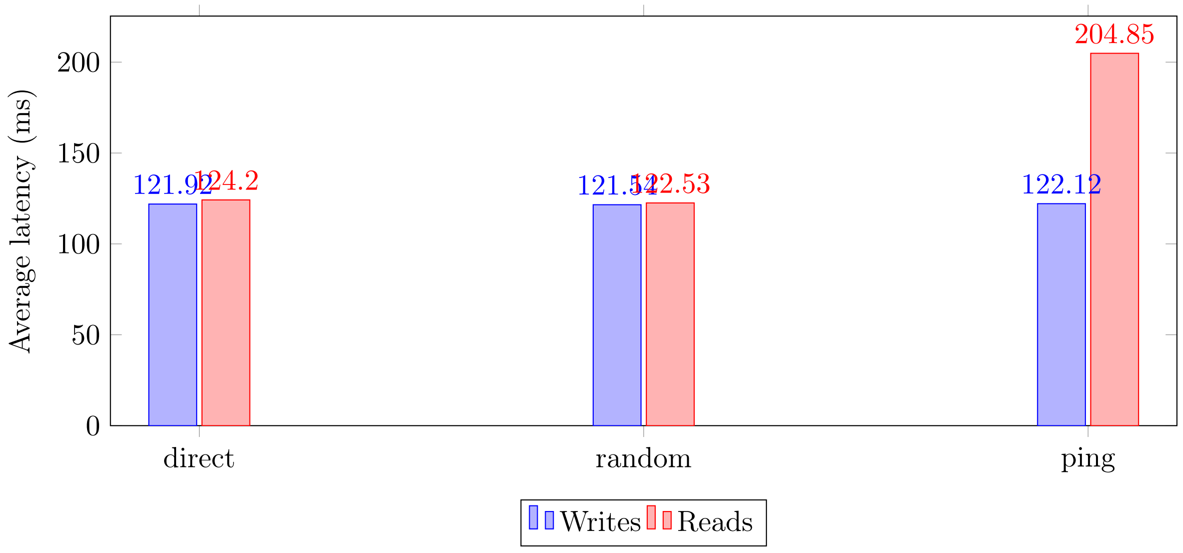

Figure 1: Average end-to-end latency for 1,000 operations

4.3 Visualization

4.4 Discussion

The `direct` and `random` strategies have similar average latency in my setup (around 120 ms). The `ping` strategy shows an important limitation: even if the median read latency is close to the other modes, the average is higher and the maximum can be very large (over 2 seconds). This is expected because ICMP ping is not a stable indicator of application-level latency under load, and a single slow outlier can significantly increase the average.

4.5 Additional latency plots (1,000 requests)

Figure 2 focuses on the 95th percentile latency, which is more representative of the "slow tail" than the average. Figure 3 highlights the maximum observed read latency, mainly to show that the `ping` mode can occasionally pick a suboptimal worker when network conditions fluctuate.

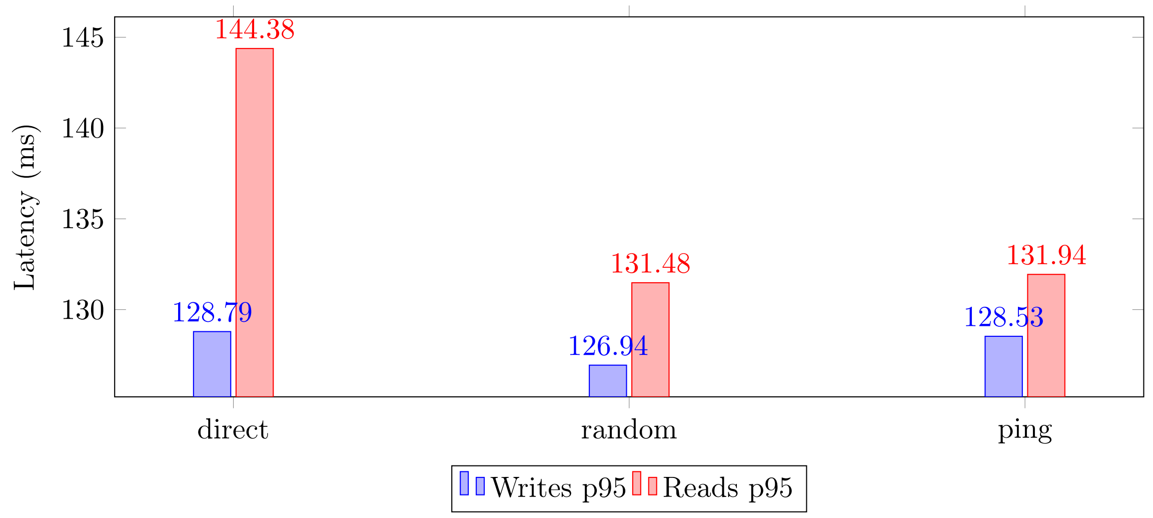

Figure 2: 95th percentile latency (p95) for 1,000 writes and 1,000 reads.

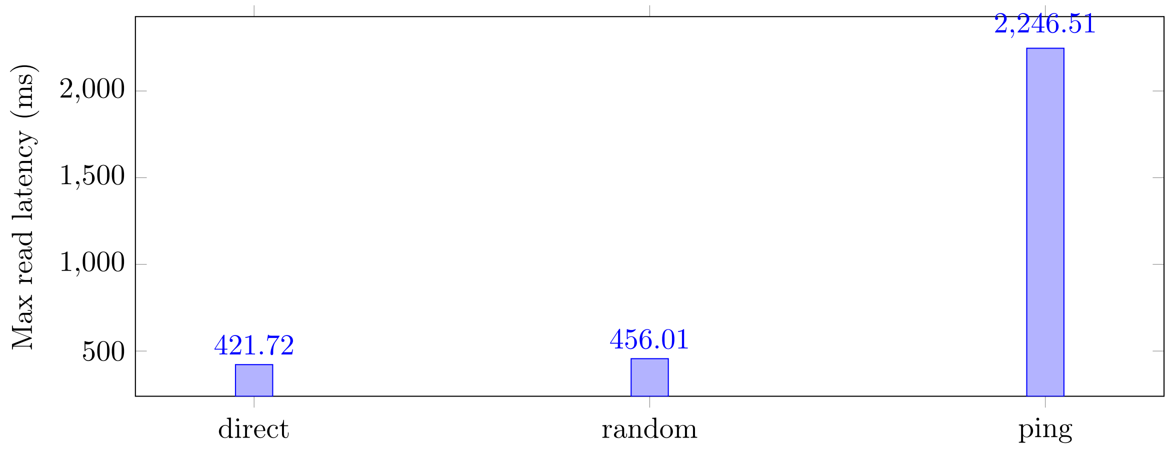

Figure 3: Maximum read latency for 1,000 reads. The `ping` mode shows rare but large outliers.

4.6 Sysbench results on the workers

To validate that the database nodes can handle concurrent OLTP traffic locally (before involving the proxy and gatekeeper), I ran `sysbench` on each worker instance against its local MySQL service (`127.0.0.1`). With 2 tables of size 100,000, 4 threads, and 30 seconds:

- Worker 1 achieved **185.61 transactions/sec** and **3712.16 queries/sec**, with an average latency of **21.54 ms**.
- Worker 2 achieved **232.16 transactions/sec** and **4643.28 queries/sec**, with an average latency of **17.22 ms**.

These numbers are not meant to be directly compared with the end-to-end HTTP benchmark (which includes Flask and network hops), but they confirm that the underlying MySQL instances are stable and performant.

5 How to Run (Reproducibility)

5.1 Provision

From AWS CloudShell:

```
python3 provision.py
```

The script prints public endpoints for the Gatekeeper.

5.2 Test

```
curl -X POST http://<GATEKEEPER_PUBLIC_IP>/query \
  -H "Content-Type: application/json" \
  -H "X-API-Key: MY_SECRET_KEY_123" \
  -d '{"sql":"SELECT COUNT(*) FROM sakila.actor;","mode":"random"}'
```

5.3 Replication status

SSH into each worker and run:

```
sudo mysql -e 'SHOW SLAVE STATUS\G' | egrep \
'Slave_IO_Running|Slave_SQL_Running|Seconds_Behind_Master'
```

6 Conclusion

This project demonstrates a secure and modular design for database access on AWS using Gatekeeper and Proxy patterns. The architecture isolates internal services, enforces a simple authentication policy at the boundary, and allows different read routing strategies without changing the client interface. My replication setup keeps the worker databases synchronized with the manager, and my benchmark results show that random replica selection is stable in this environment, while ping-based selection can introduce latency spikes. Future improvements could include more robust load-aware routing metrics, better SQL validation at the Gatekeeper, and additional monitoring for replication lag.

7 Sources

- Amazon Web Services (AWS) documentation: EC2 instances, security groups, and user data scripts: https://docs.aws.amazon.com/ec2/
- Boto3 (AWS SDK for Python) documentation: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html
- MySQL 8.0 documentation on replication (GTID, replication configuration): https://dev.mysql.com/doc/refman/8.0/en/replication.html
- Sysbench documentation (OLTP workloads and usage): https://github.com/akopytov/sysbench
- Flask documentation (HTTP API service): https://flask.palletsprojects.com/
- MySQL Connector/Python documentation: https://dev.mysql.com/doc/connector-python/en/