# Cloud Design Patterns: MySQL Cluster with Proxy and Gatekeeper

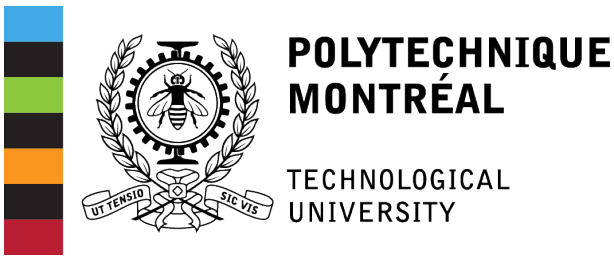
École Polytechnique de Montréal

Advanced Concepts of Cloud Computing

**LOG8415E**

2025 - 2026

**Miguel Carrasco Guirao**



# Code

The code of the assignment can be found on the following repository:
https://github.com/miguelxsm/Cloud-Design-Patterns

# 1 Benchmarking MySQL with Sysbench

Before deploying the distributed MySQL cluster and applying the Proxy and Gatekeeper patterns, each database instance was validated in standalone mode. The goal of this step is to ensure that MySQL is correctly installed, operational, and capable of handling basic workloads, as required by the assignment specification.

## 1.1 Setup and Execution Procedure

Once the EC2 instances were launched and MySQL together with the Sakila database was installed, the validation was performed by connecting to each instance via SSH and executing `sysbench` locally.

First, `sysbench` was installed on the instance:

```
sudo apt-get install -y sysbench
```

After installation, the database was prepared for benchmarking using the read-only OLTP workload provided by sysbench. The preparation step creates auxiliary tables used internally by the benchmark:

```
sudo sysbench /usr/share/sysbench/oltp_read_only.lua \
--mysql-db=sakila \
--mysql-user=mysqluser \
--mysql-password=mysqlpassword \
prepare
```

Finally, the benchmark was executed:

```
sudo sysbench /usr/share/sysbench/oltp_read_only.lua \
--mysql-db=sakila \
--mysql-user=mysqluser \
--mysql-password=mysqlpassword \
run
```

## 1.2 Observed Results

The execution completed successfully on all instances. An excerpt of the output is shown below:

```
transactions: 8068 (806.50 per sec.)
queries: 129088 (12904.05 per sec.)

Latency (ms):
avg: 1.24
95th percentile: 2.00
```

These results confirm that each MySQL instance is functioning correctly and is able to sustain a stable workload with low latency in standalone mode. This validation step provides a reliable baseline before enabling replication and introducing the Proxy and Gatekeeper patterns.

# 2 Proxy Pattern

This section describes the implementation of the *Proxy* pattern (read/write routing and load distribution) using ProxySQL as the single internal entry point to the MySQL cluster.

## 2.1 Proxy Security Group Permissions

The ProxySQL instance is treated as an internal *Trusted Host*. Therefore, inbound connectivity is restricted to the minimum set required for operation and maintenance:

- **SSH (22/tcp)** is allowed only from the operator IP (`MY_IP/32`) for administration.
- **MySQL frontend (3306/tcp)** is allowed only from:
  - the operator IP (`MY_IP/32`) for debugging/validation when needed, and
  - the Gatekeeper Security Group (source = `sg_gateway_id`), so that *all* application traffic reaches the database through the Gatekeeper first.

The corresponding inbound rules are:

```
def build_proxy_permissions(sg_gateway_id):
return [
{"IpProtocol": "tcp", "FromPort": 22, "ToPort": 22,
       "IpRanges": [{"CidrIp": f"{MY_IP}/32"}]},
{"IpProtocol": "tcp", "FromPort": 3306, "ToPort": 3306,
       "IpRanges": [{"CidrIp": f"{MY_IP}/32"}]},
{"IpProtocol": "tcp", "FromPort": 3306, "ToPort": 3306,
       "UserIdGroupPairs": [{"GroupId": sg_gateway_id}]},
]
```

This configuration enforces the intended topology: Internet traffic reaches only the Gatekeeper, and only validated requests are forwarded to the Proxy (Trusted Host). Direct access to the database nodes is not permitted.

## 2.2 ProxySQL Setup and Routing Model

ProxySQL is installed and configured to:

- expose a MySQL-compatible frontend on **3306/tcp** for client connections (used by the Gatekeeper),
- expose an admin interface on **6032/tcp** (loopback only) for configuration and runtime updates,
- maintain two `hostgroups`: **10** for the manager, and **20** for the workers.

The base provisioning installs ProxySQL, moves the frontend port to 3306, resets the internal database, and configures the manager backend and the application user:

```
# Install ProxySQL and set frontend to 3306
apt-get install -y proxysql
sed -i 's/interfaces="0.0.0.0:6033"/interfaces="0.0.0.0:3306"/' /etc/
    proxysql.cnf || true

# Reset ProxySQL state and start
rm -f /var/lib/proxysql/proxysql.db || true
systemctl start proxysql
```

```
# Configure manager backend (hostgroup 10)
INSERT INTO mysql_servers(hostgroup_id,hostname,port,max_connections)
VALUES (10,'<MANAGER_PRIVATE_IP>',3306,200);

# Configure application user
INSERT INTO mysql_users(username,password,default_hostgroup)
VALUES ('<MYSQL_USER>','<MYSQL_PASS>',10);
```

Read/write routing is implemented via query rules:

- **WRITE operations** and `SELECT ... FOR UPDATE` are routed to the manager (hostgroup 10).
- **READ-only SELECT** statements are routed to workers (hostgroup 20) when a worker pool is enabled.

## 2.3 Forwarding Strategies

The proxy must support three forwarding strategies: *Direct Hit*, *Random*, and *Customized*. The strategy is selected at provisioning time through the `strategy` parameter of the ProxySQL user-data.

### 2.3.1 Direct Hit

In the *Direct Hit* strategy, all traffic is forwarded to the manager node regardless of query type. This provides a baseline with no read scaling.

```
INSERT INTO mysql_query_rules(rule_id,active,match_pattern,
    destination_hostgroup,apply)
VALUES (1,1,'^SELECT',10,1);
LOAD MYSQL QUERY RULES TO RUNTIME;
SAVE MYSQL QUERY RULES TO DISK;
```

Since no workers are configured for routing in this strategy, reads are not distributed and the proxy behaves as a pure pass-through to the manager.

### 2.3.2 Random

In the *Random* strategy, a worker pool is enabled in hostgroup 20 and read/write splitting is activated. The worker nodes are inserted with equal weights so that ProxySQL distributes reads evenly (round-robin over equally weighted servers).

**Backends (manager + workers).**

```
INSERT INTO mysql_servers(hostgroup_id,hostname,port,max_connections)
    VALUES
(10,'<MANAGER_PRIVATE_IP>',3306,200),
(20,'<WORKER1_PRIVATE_IP>',3306,200),
(20,'<WORKER2_PRIVATE_IP>',3306,200);
LOAD MYSQL SERVERS TO RUNTIME;
SAVE MYSQL SERVERS TO DISK;
```

**Read/Write split rules.**

```
INSERT INTO mysql_query_rules(rule_id,active,match_pattern,
    destination_hostgroup,apply) VALUES
(1,1,'^SELECT.*FOR UPDATE',10,1),
```

```
(2,1,'^SELECT',20,1);
LOAD MYSQL QUERY RULES TO RUNTIME;
SAVE MYSQL QUERY RULES TO DISK;
```

**Note: ^ SELECT means: all queries starting with SELECT**
With equal weights in hostgroup 20, the proxy distributes read queries across workers, while all writes (and locking reads) are routed to the manager.

### 2.3.3 Customized

The *Customized* strategy extends the Random strategy by dynamically adjusting worker weights based on measured network latency. A controller process runs periodically (every `PERIOD` seconds) and updates the weights of workers in hostgroup 20.

**Controller scheduling.** The controller is installed as a systemd service to ensure it is started automatically and restarted on failure:

```
cat > /etc/systemd/system/proxysql-ping-controller.service <<'EOS'
[Unit]
Description=ProxySQL Ping Controller (customized strategy)
After=network-online.target proxysql.service
Wants=network-online.target

[Service]
Type=simple
ExecStart=/bin/bash /usr/local/bin/proxysql_ping_controller.sh
Restart=always
RestartSec=2

[Install]
WantedBy=multi-user.target
EOS

systemctl daemon-reload
systemctl enable --now proxysql-ping-controller.service
```

**Weight computation.** Every cycle, the controller measures worker RTT using ICMP ping and applies smoothing and normalization to compute new weights:

$$\mathrm{ema}_i(t) = \alpha \cdot \mathrm{rtt}_i(t) + (1-\alpha) \cdot \mathrm{ema}_i(t-1) \tag{1}$$

$$s_i = \frac{1}{\mathrm{ema}_i + \varepsilon} \tag{2}$$

$$w_i = w_{\min} + (w_{\max} - w_{\min}) \frac{s_i}{\sum_j s_j} \tag{3}$$

In addition, a per-cycle rate limit is applied to avoid abrupt weight changes (anti-flapping): the weight update is bounded by `DELTA_MAX` per cycle.

**Applying weights in ProxySQL.** When weights change, the controller updates the `mysql_servers` table and loads the change at runtime:

```
UPDATE mysql_servers
SET weight = CASE hostname
WHEN '<WORKER1_PRIVATE_IP>' THEN <W1>
```

```
            WHEN '<WORKER2_PRIVATE_IP>' THEN <W2>
            END
            WHERE hostgroup_id=20;

            LOAD MYSQL SERVERS TO RUNTIME;
```

This results in a continuously adapted read distribution, where workers with lower measured latency receive proportionally higher weights. Writes remain routed to the manager via the same read/write split rules as in the Random strategy.

# 3 Gatekeeper Pattern

This section describes the implementation of the *Gatekeeper* pattern. The Gatekeeper is the only Internet-facing component and brokers all access to storage. The ProxySQL instance acts as the *Trusted Host*, meaning it only receives *validated* requests from the Gatekeeper and is not directly exposed to the Internet.

## 3.1 Gatekeeper Security Group Permissions

The inbound rules applied to the Gatekeeper Security Group are the following:

```
            IP_PERMISSIONS_GATEWAY = [
            {
                   "IpProtocol": "tcp",
                   "FromPort": 22,
                   "ToPort": 22,
                   "IpRanges": [{"CidrIp": f"{MY_IP}/32"}],
            },
            {
                   "IpProtocol": "tcp",
                   "FromPort": 80,
                   "ToPort": 80,
                   "IpRanges": [{"CidrIp": "0.0.0.0/0"}],
            },
            ]
```

These rules are justified as follows:

- **SSH (22/tcp)** is restricted to the operator's public IP (`MY_IP/32`) and is used exclusively for administrative access and debugging.
- **HTTP (80/tcp)** is exposed publicly (`0.0.0.0/0`) to allow users and API clients to send requests to the Gatekeeper service.

No database-related ports are exposed on the Gatekeeper. In particular:

- the Gatekeeper does **not** accept MySQL connections,
- it does **not** expose the ProxySQL frontend,
- it has no direct inbound connectivity to any database node.

All database access is initiated *outbound* by the Gatekeeper towards the Trusted Host (ProxySQL) over the private VPC network. This strictly enforces the Gatekeeper pattern requirement that only validated requests are forwarded to internal roles, while the Gatekeeper itself remains the sole Internet-facing component.

### 3.2 Gatekeeper Service: FastAPI Broker

A dedicated FastAPI service is deployed on the Gatekeeper instance. This service implements the brokering logic required by the assignment:

1. receive user/API requests over HTTP,
2. check authentication/authorization (simple API key, as allowed by the course staff),
3. validate and sanitize input queries to reject unsafe statements,
4. forward only validated queries to the Trusted Host (ProxySQL) through an internal channel,
5. return results back to the user.

### 3.3 User-Data Deployment (Service Installation)

The Gatekeeper is deployed end-to-end via instance user-data. The provisioning installs Python, creates a virtual environment, installs dependencies, writes the FastAPI application, and configures a systemd unit running Uvicorn:

```
apt-get install -y python3 python3-venv python3-pip ca-certificates curl

python3 -m venv venv
source venv/bin/activate
pip install fastapi uvicorn mysql-connector-python

# write server.py and start systemd service
mkdir -p {app_dir}
echo "{code}" | base64 -d > {app_dir}/server.py
...
ExecStart=/opt/gatekeeper/venv/bin/python -m uvicorn server:app \
--host 0.0.0.0 --port 80

systemctl enable --now gatekeeper
curl -fsS http://127.0.0.1:80/health
```

A `/health` endpoint is used as a local smoke-test and for runtime monitoring.

### 3.4 Authentication and Authorization

All database requests are sent to `POST /query`. The Gatekeeper requires an API key through the `X-API-Key` header. Requests missing the key or presenting an invalid key are rejected with HTTP 401 and are never forwarded to the proxy:

```
def auth_or_401(x_api_key: Optional[str]) -> None:
if not x_api_key or x_api_key != API_KEY:
raise HTTPException(status_code=401, detail="unauthorized")
```

### 3.5 Query Safety Checks (Input Validity)

The assignment explicitly requires the Gatekeeper to reject unsafe queries (e.g., destructive DDL). This is implemented by enforcing:

- **single-statement only** (reject multi-statement payloads),
- **denylist** for dangerous keywords (e.g., `DROP`, `TRUNCATE`, `ALTER`, privileged operations),

- **allowlist** for top-level statements limited to `SELECT`, `INSERT`, `UPDATE`, `DELETE`.

```
DENY_REGEX = re.compile(r"\b(DROP|TRUNCATE|ALTER|GRANT|REVOKE|...)\b",
    re.I)
ALLOW_TOPLEVEL = re.compile(r"^\s*(SELECT|INSERT|UPDATE|DELETE)\b", re.I
    )

def validate_query(sql: str) -> None:
if not is_single_statement(sql):
raise HTTPException(status_code=403, detail="multiple statements")
if DENY_REGEX.search(sql):
raise HTTPException(status_code=403, detail="forbidden keyword")
if STRICT_ALLOWLIST and not ALLOW_TOPLEVEL.match(sql):
raise HTTPException(status_code=403, detail="statement not allowed")
```

Additionally, response size is bounded to avoid accidental large result sets:

- maximum returned rows (`MAX_ROWS`),
- approximate byte cap (`MAX_RESULT_BYTES`).

### 3.6 Forwarding to the Trusted Host

Once authenticated and validated, the request is forwarded to the Trusted Host (ProxySQL) using a MySQL connection pool. The pool connects to the Proxy over the private network:

```
conn_kwargs = dict(
host=PROXY_HOST,
port=PROXY_PORT,
user=DB_USER,
password=DB_PASSWORD,
autocommit=True,
connection_timeout=5,
)
_pool = mysql.connector.pooling.MySQLConnectionPool(
pool_name=POOL_NAME, pool_size=POOL_SIZE, **conn_kwargs
)
```

## 4 Request Flow Overview

The lifecycle of a request in the final system follows a clear separation of concerns between components:

1. A client sends an HTTP request containing a SQL query to the Gatekeeper, the only Internet-facing component.
2. The Gatekeeper authenticates the request using an API key and validates the query to ensure it is safe to execute.
3. Valid requests are forwarded over the private network to the Trusted Host (ProxySQL).
4. The Proxy classifies the query as a READ or WRITE and routes it to the appropriate database node according to the active strategy.
5. The database executes the query and the result is returned to the client through the Gatekeeper.

This flow ensures that all database access is mediated by the Gatekeeper, while routing and load balancing are handled exclusively by the Proxy.

# 5 Benchmarking the Cluster

This section describes the benchmarking methodology used to evaluate the *end-to-end* system. The assignment requires sending **1000 READ** and **1000 WRITE** requests *for each* proxy forwarding strategy and showing that the cluster receives and processes them appropriately. This requirement is satisfied by a deterministic workload executed through the Gatekeeper HTTP API.

## 5.1 Benchmark Client Overview

A dedicated Python client (`bench.py`) is used as a workload generator and measurement probe. It sends SQL queries to the Gatekeeper endpoint (`POST /query`) and measures:

- request success/failure (HTTP 200 vs non-200),
- end-to-end latency per request (ms),
- throughput over time (TPS) for READ and WRITE requests.

The benchmark targets the system *as deployed*: all queries traverse the Gatekeeper validation logic and are forwarded to ProxySQL as in a real execution path. This avoids measuring internal components in isolation.

## 5.2 Workload Model (Parallel READ/WRITE Streams)

The benchmark implements a mixed workload using exactly two concurrent streams:

- one sequential READ stream,
- one sequential WRITE stream,

running in parallel using two threads. Each stream issues a fixed number of requests, resulting in exactly:

$$N_{\text{reads}} = 1000, \quad N_{\text{writes}} = 1000, \quad N_{\text{total}} = 2000.$$

This approach provides a **more realistic workload** than strictly serialized execution because reads and writes contend simultaneously for shared resources (proxy routing, manager write path, locks, networking). At the same time, it avoids an overly aggressive multi-threaded generator that could shift the bottleneck to the client.

**Fixed queries.** To ensure reproducibility, both streams use fixed SQL statements:

```
FIXED_READ_SQL = (
"SELECT actor_id, first_name, last_name "
"FROM sakila.actor "
"WHERE actor_id = 1"
)

FIXED_WRITE_SQL = (
"INSERT INTO sakila.bench_events (created_at, payload) "
"VALUES (NOW(6), 'x')"
)
```

### 5.3 Execution Path and Request Measurement

Each request is sent as a JSON payload through the Gatekeeper endpoint using an API key:

```
code, body = http_post_json(
endpoint,
api_key,
{"query": sql},
timeout_s=timeout_s
)
```

The client measures end-to-end latency as the wall-clock duration of the HTTP call:

```
t0 = time.perf_counter()
code, body = http_post_json(...)
t1 = time.perf_counter()
lat_ms = (t1 - t0) * 1000.0
ok = 1 if code == 200 else 0
```

Each completion is timestamped (`time.time()`) to build a time series, and a record is stored containing:

- kind (`read`/`write`),
- success flag and HTTP status,
- latency in milliseconds,
- completion time (used for 1-second bucketing).

**Parallel execution.** The two streams run concurrently and join at the end of the benchmark:

```
th_r = threading.Thread(target=run_stream, args=("read", ...))
th_w = threading.Thread(target=run_stream, args=("write", ...))

th_r.start(); th_w.start()
th_r.join(); th_w.join()
```

### 5.4 Derived Metrics and Output Artifacts

After the run completes, raw request records are aggregated into three output CSV files. Together, these outputs provide all evidence required by the assignment.

#### 5.4.1 `summary.csv` (Assignment compliance: 1000 READ + 1000 WRITE)

A single-row summary is generated per run. It explicitly reports the number of requests sent and successfully processed:

```
return {
       "total_sent": total,
       "read_sent": reads,
       "write_sent": writes,
       "ok_total": ok_total,
       "ok_read": ok_reads,
       "ok_write": ok_writes,
       "duration_s": ...,
       "avg_tps_ok": ...,
}
```

This file is the direct evidence that each strategy run issues exactly 1000 read requests and 1000 write requests and that the cluster processes them (HTTP 200).

#### 5.4.2 `tps_timeseries.csv` (Throughput vs time)

Throughput is computed by binning request *completions* into 1-second buckets. For each second $t$, the script counts total, read, and write completions, producing:

$$\text{TPS}(t) = \#\text{completions in second } t.$$

```
b = int(r.t_wall_end) # 1s bucket
buckets[b]["total"] += 1
buckets[b][r.kind] += 1
```

The resulting time series contains: `total_tps`, `read_tps`, `write_tps` and their raw counts, enabling plots of throughput stability and READ/WRITE balance.

#### 5.4.3 `latency_timeseries.csv` (Latency vs time, tail percentiles)

Latency is aggregated in the same 1-second buckets, *restricted to successful requests* (HTTP 200), and summarized using:

- mean,
- p50 (median),
- p95,
- p99,
- max,

computed separately for READ and WRITE operations.

```
if r.ok != 1:
continue
buckets[b][r.kind].append(r.lat_ms)
```

Percentiles are computed deterministically from the bucket samples. The p95 metric is used to represent tail latency (worst-case behavior excluding extreme outliers), which is a standard indicator of user-perceived performance under load.

### 5.5 Running the Benchmark per Strategy

The benchmark is executed once per forwarding strategy. The proxy strategy is activated externally (ProxySQL configuration), while the benchmark client simply labels the output directory:

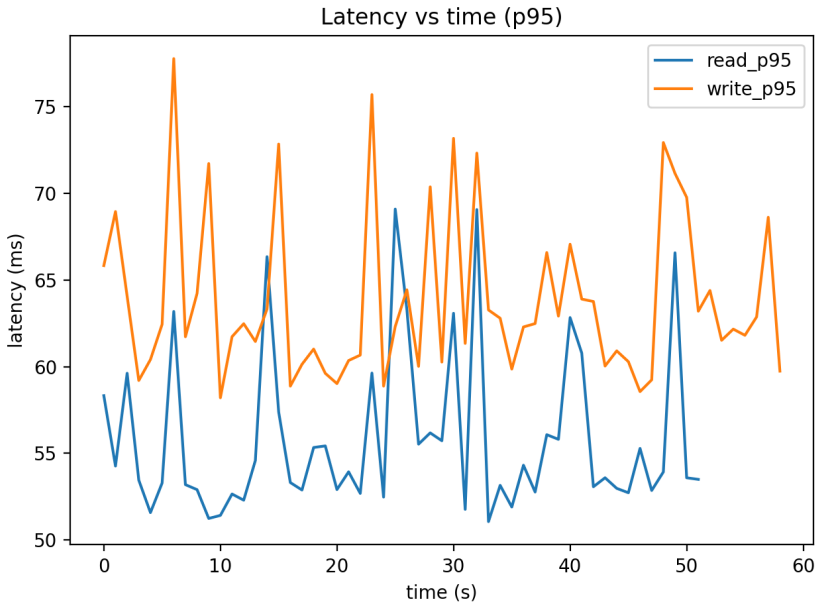
```
python3 bench.py --gateway-url http://<GATEWAY_PUBLIC_IP> \
--api-key <KEY> --strategy direct --reads 1000 --writes 1000 \
--outdir ./benchmarking/direct

python3 bench.py --gateway-url http://<GATEWAY_PUBLIC_IP> \
--api-key <KEY> --strategy random --reads 1000 --writes 1000 \
--outdir ./benchmarking/random

python3 bench.py --gateway-url http://<GATEWAY_PUBLIC_IP> \
--api-key <KEY> --strategy customized --reads 1000 --writes 1000 \
--outdir ./benchmarking/customized
```

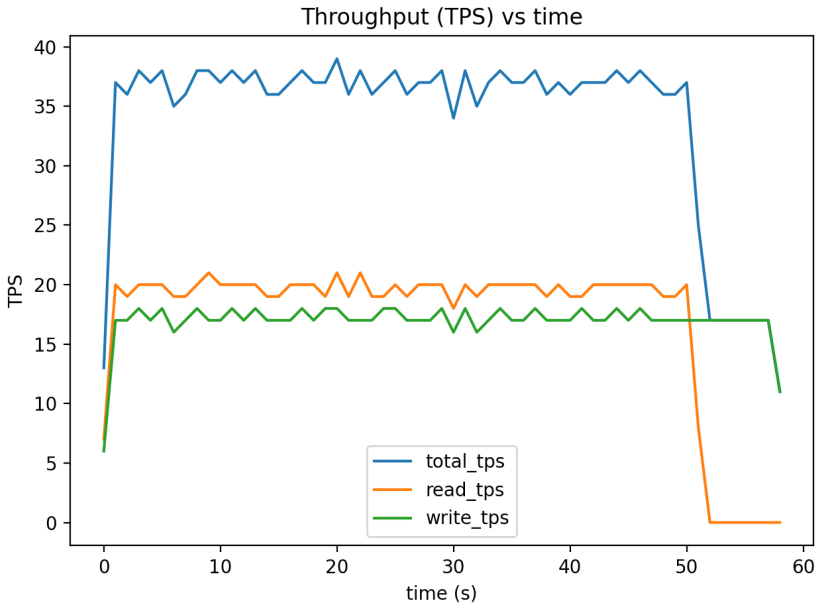
For each strategy, the produced CSV artifacts (`summary.csv`, `tps_timeseries.csv`, `latency_timeseries.csv`) are then used to generate the plots reported in the Results section (throughput vs time, p95 latency vs time, and cross-strategy bar comparisons).

# 6 Results

This section presents the experimental results obtained from benchmarking the cluster under the three proxy forwarding strategies: *Direct Hit*, *Random*, and *Customized*. All experiments were executed with the same workload: 1000 READ and 1000 WRITE requests issued in parallel through the Gatekeeper.

## 6.1 Direct Hit Strategy

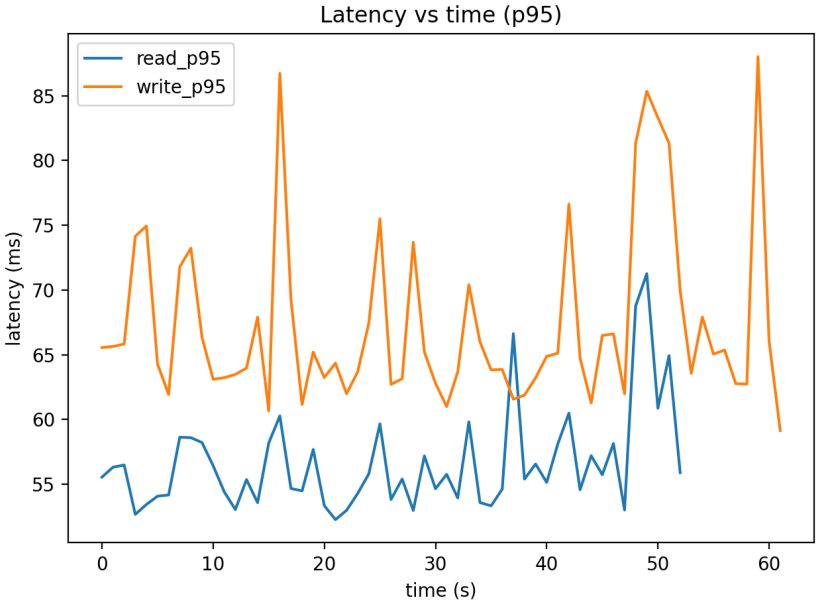


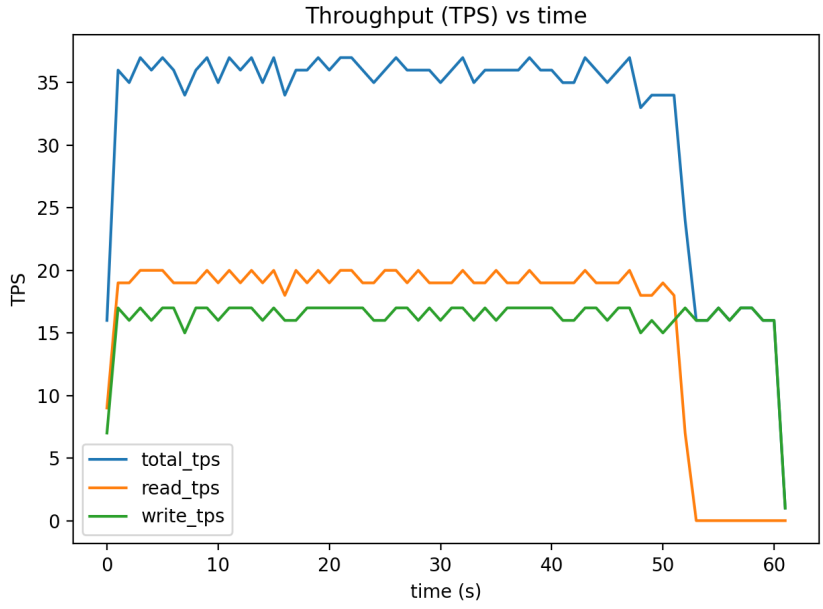



| Metric | Value |
|---|---|
| Total requests sent | 2000 |
| READ requests | 1000 |
| WRITE requests | 1000 |
| Successful requests | 2000 |
| Total duration (s) | 57.99 |
| Average TPS (OK) | 34.49 |

In the Direct Hit strategy, all requests (READ and WRITE) are routed to the manager node. This provides a baseline reference for throughput and latency when no load distribution is applied.

### 6.2 Random Strategy

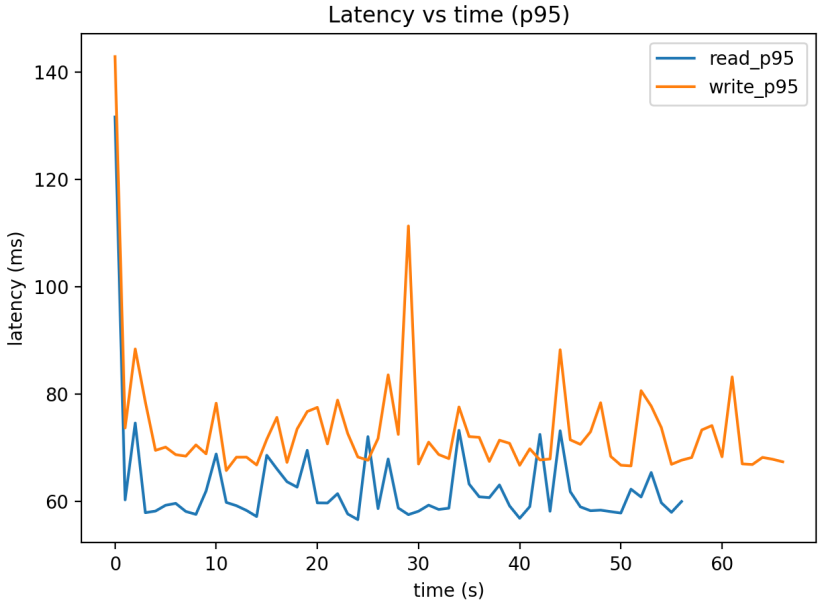


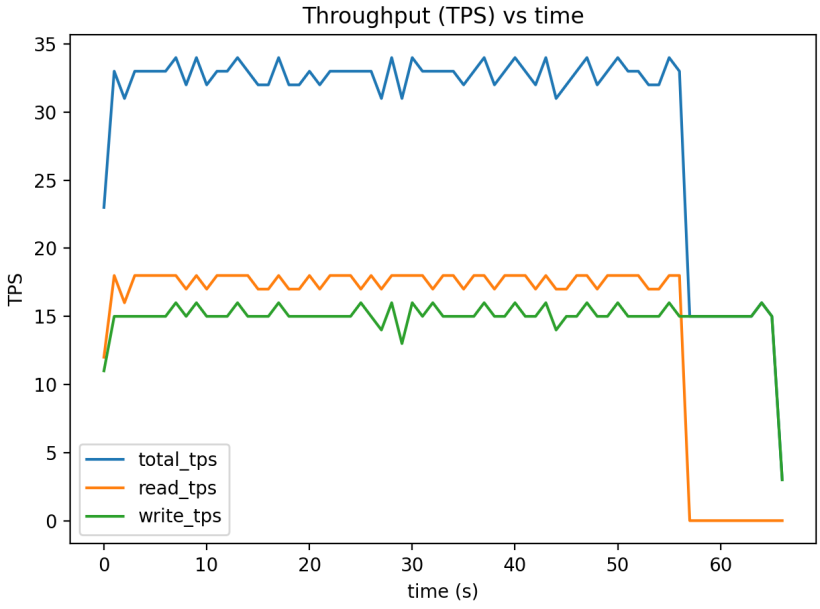



| Metric | Value |
|---|---|
| Total requests sent | 2000 |
| READ requests | 1000 |
| WRITE requests | 1000 |
| Successful requests | 2000 |
| Total duration (s) | 60.48 |
| Average TPS (OK) | 33.07 |

Under the Random strategy, READ requests are distributed uniformly across worker nodes while WRITE requests continue to be routed to the manager. This introduces load balancing for reads without considering node performance.

### 6.3 Customized Strategy



| Metric | Value |
|---|---|
| Total requests sent | 2000 |
| READ requests | 1000 |
| WRITE requests | 1000 |
| Successful requests | 2000 |
| Total duration (s) | 66.26 |
| Average TPS (OK) | 30.19 |

In the Customized strategy, READ requests are routed dynamically based on worker responsiveness. Worker weights are updated periodically according to measured ping latency, resulting in adaptive load distribution.

### 6.4 Cross-Strategy Analysis

Comparing the three strategies reveals clear trade-offs:

- **Throughput:** Direct Hit achieves the highest average throughput, as it avoids proxy-level decision overhead and replication-induced read routing. Random incurs a small throughput penalty, while Customized shows the lowest average TPS due to dynamic weight adjustments and additional control logic.
- **Latency behavior:** Tail latency (p95) for READ operations is generally lower and more stable under Random and Customized strategies compared to Direct Hit, as read load is offloaded from the manager. Customized shows smoother latency trends, at the cost of reduced throughput.
- **Stability vs performance:** The Random strategy improves scalability over Direct Hit with minimal complexity. The Customized strategy prioritizes stability and responsiveness, adapting to worker performance variations while accepting reduced peak throughput.

### 6.5 Final Conclusions

The experimental results shows that:

- Direct Hit serves as a high-throughput baseline but does not scale read load.
- Random read distribution provides a good balance between simplicity and scalability.
- Customized routing offers adaptive behavior and improved latency stability under heterogeneous conditions, at the cost of lower overall throughput.

Together, these results validate the functional correctness of the proxy strategies and illustrate the performance trade-offs inherent in each design choice.

## 7 Instructions to Run the Code

This section describes the steps required to deploy the infrastructure, configure the proxy strategy, and execute the benchmark. The deployment process is automated but **order-sensitive**; components must be created in the correct sequence to ensure correct connectivity and security isolation.

### 7.1 Prerequisites

Before execution, the following requirements must be satisfied:

- An AWS account with permissions to create EC2 instances, Security Groups, and networking resources.
- AWS credentials configured locally (e.g., via `aws configure`).
- Python 3 environment with required dependencies installed.
- The repository cloned locally, with the working directory set to the project root.

### 7.2 Infrastructure Deployment Script

The deployment is controlled by a single Python entry point that orchestrates:

- Security Group creation,
- Database instance creation,
- Proxy instance creation,
- Gatekeeper instance creation.

The script supports partial or full deployment via command-line flags. If no flags are provided, the full infrastructure is created in the correct order automatically.

### 7.3 Deployment Order and Rationale

The correct creation order is the following:

1. **Security Groups** Must be created first, as all instances depend on their inbound and outbound rules.
2. **Main database instances (manager and workers)** These instances form the MySQL cluster and must exist before the proxy is configured.
3. **Proxy instance (ProxySQL)** Requires the private IPs of the database nodes and the selected forwarding strategy.
4. **Gatekeeper instance** Depends on the proxy private IP and acts as the only Internet-facing component.

The script enforces this order internally when executed without flags.

### 7.4 Full Deployment (Recommended)

To deploy the entire system in one step using the default strategy (`directhit`):

```
python3 main.py
```

This command performs the following actions:

- creates all required Security Groups,
- launches the MySQL manager and worker instances,
- deploys the ProxySQL instance,
- deploys the Gatekeeper instance.

## 7.5 Selecting the Proxy Strategy

The proxy forwarding strategy can be selected at deployment time using the `-strategy` flag. Supported values are:

- `directhit`
- `random`
- `customized`

Example:

```
python3 main.py --strategy random
```

This flag affects only the ProxySQL configuration. The benchmark client uses the strategy name solely as a label for result organization.

## 7.6 Partial Deployment (Advanced Usage)

Individual components can be created selectively if needed:

```
python3 main.py --sg
python3 main.py --instances
python3 main.py --proxy --strategy customized
python3 main.py --gateway
```

This mode is intended for debugging or iterative development. The user is responsible for respecting the correct order.

## 7.7 Destroying the Infrastructure

To tear down all deployed resources:

```
python3 main.py --destroy
```

This command deletes all EC2 instances and associated Security Groups created.

## 7.8 Benchmark Execution

Once the infrastructure has been successfully deployed (Gateway, Proxy, and MySQL cluster running), the benchmarking phase can be executed from any external machine with network access to the Gatekeeper public endpoint.

### 7.8.1 Execution Command

The benchmark is executed by running:

```
python3 bench.py \
--gateway-url http://<GATEWAY_PUBLIC_IP> \
--api-key <API_KEY> \
--strategy <direct|random|customized> \
--reads 1000 \
--writes 1000 \
--outdir ./benchmarking/<strategy>
```

The most relevant parameters are:

- `-gateway-url`: public HTTP endpoint of the Gatekeeper instance.
- `-api-key`: API key required by the Gatekeeper for authentication.

- `-strategy`: label identifying the proxy routing strategy used in the deployment.
- `-reads`, `-writes`: number of READ and WRITE requests issued by the client.
- `-outdir`: output directory where all benchmark artifacts are stored.

### 7.8.2 Generated Outputs

For each execution, the benchmark client produces the following artifacts:

- `summary.csv`: aggregated metrics (sent vs successful requests, average TPS).
- `tps_timeseries.csv`: throughput per second, split into READ, WRITE, and TOTAL.
- `latency_timeseries.csv`: latency per second with mean, p50, p95, p99, and max values.
- `raw_requests.csv` (optional): per-request audit log for debugging and validation.

These CSV files are subsequently used to generate the plots and tables presented in the Results section.