École Polytechnique de Montréal

Département de génie informatique et génie logiciel

Course: LOG8415E : Advanced Concepts of Cloud Computing

Project: Cloud Design Patterns: Implementing a DB Cluster

Moussa Fofana 1955968

Project GitHub repository: `https://github.com/fomou/LOG8415-project`

Demo Video link: `https://youtu.be/zfs7OSgSHdA`

April 2, 2025

Abstract

In this project, I was tasked with setting up a MySQL database on AWS EC2 and implementing cloud design patterns. This involved integrating the Gatekeeper and Proxy patterns into the architecture. I began by installing, configuring, and deploying a stand-alone MySQL instance on EC2. Subsequently, I implemented a distributed MySQL database utilizing these cloud design patterns. Benchmarking with sysbench revealed that the database handled 251,664 read queries, 71,904 write queries, and 35,952 other queries. Additionally, I benchmarked each proxy implementation with 1,000 requests. The results showed that all 1,000 write and direct implementation operations were managed solely by the manager. For the custom implementation, 296 requests were processed by the manager, 334 by worker1, and 370 by worker2. For the random implementation, 1,000 read requests were nearly evenly distributed, with 514 handled by worker2 and 486 by worker2.

Keywords— MySQL, Cloud pattern, Gatekeeper, Proxy, sysbench, Flask, AWS EC2

Contents

1 Benchmarking MySQL with sysbench

I started by benchmarking the MySQL stand-alone databases setup in workers and manager instances deployed on t2.micro in AWS. The installation of the MysQl, Sakila database, and the benchmarking command were all deployed through user-data sent as parameter during the instances creations. Once the stand-alone server had been set up, I used sysbench to perform the benchmarking with the following parameters:

- Operation: Read and Write
- Database: Sakila
- Table size: 100,000
- Threads: 6
- Maximum time: 60s

The results are recapitulated in the following tables

Table 1: Queries performed

Operations	number of queries
read	251664
write	71904
Others	35952
Total	359520

Table 2: Latency

Statistics	Latency (ms)
min:	6.11
avg:	20.03
max:	170.69
95th percentile:	27.17
sum:	359991.47

2 Implementation of The Proxy pattern.

The proxy pattern uses data replication between master/slave databases to route requests and provide read scalability on a relational database, such as MYSQL. Write requests are handled by the Manager and replicated on its workers, while read requests are processed by workers and Manager depending on the strategy we want to use. I've made a script that launches a t2.large instance and installs all necessary dependencies, such as Flask and request, who will be connecting and routing requests to the cluster.

My Proxy implementation consists of a simple Flask application that listens to 4 endpoints namely */ping*, */table_size*, */read*, and */write*. The cluster instances, Manager and workers also have the same endpoints. The */ping* endpoint is mainly used to perform an health check on the server, and */table_size* will be used to monitor the size of the database inside the managed cluster. The proxy handles the cluster on the others endpoints as follows:

2.1 Write operation

The proxy will call manager's */write* endpoint with it's private ip (i.e manager_ip/write). Once the manager receives the write request, it will insert a random item, for example person={id: 1, name_51}, in the test table and call the workers endpoint */write* with the created item as parameters to replicate it on them.

2.2 Read operation

Regarding the read operation, three strategies were implemented to handle requests.

- Direct hit: It simply directly forwards the read requests to MySQL Manager on the */read* endpoint using the private ip and straightly retrieves the data from there.
- Random: The proxy randomly chooses a worker node on the cluster and forward the request to it by calling its endpoint */read*

- Custom: the proxy starts by sending a ping request to the cluster node,s and does so by calling the */ping* endpoint of Manager and works, and after forward the read request to one with less response time.

3 Implementation of The Gatekeeper pattern

The Gatekeeper is a cloud design pattern that tends to leverage the security best practice and can lead to a minimum attack surface of the system. It does it by communicating only over internal channel with others components of the system. It is composed by two instances, The Gatekeeper is the internet-facing web (aka the main entry of the application) and an internal-facing instance called the Trusted Host.

My implementation of the Gatekeeper and the Trusted Host is a simple Flask application same as the instances described above. They have the same endpoints namely */ping*, */table_size*, */read*, and */write*, but they do an advanced pre-process on the request before sending it to internal components.

- **Gatekeeper**: When it receives a message from a client on any of its endpoints, will performs a quick sanity check and sanitize the request if needed. Once the pre-process is done, it will forward the message to the Trusted Host through it private ip on a special port that is 5050 in my case.
- **Trusted Host**: It is only accessible through the internal channel, and doesn't have any unnecessary port and service running. When it receives a request, before taking any further action, it will ensure that the request came from the Gatekeeper by comparing the host address to the address of the gatekeeper that it knows. After, it will ensure again that the invoked strategy is one of {direct, random, custom} and will perform further sanitizing on the message. Once everything is validated, it will forward the request to the proxy for processing on the appropriate route through the internal channel(only private ip address)

4 How the implementation works.

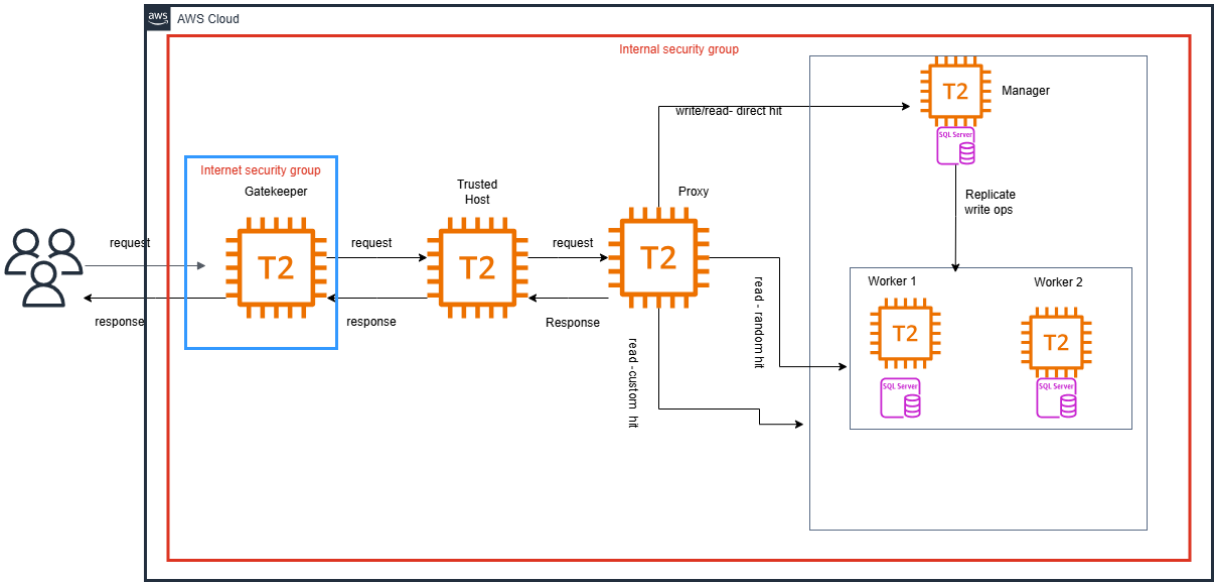


Figure 1: Architecture of the system

As we have already seen above, each instance runs a Flask application that has four routes inside namely */ping*, */table_size*, */read*, and */write*. The implementation can be described as something we can call a "**pipeline of API calls**", and can be explained in the following sequences

1. step 1: clients sends a request to the system by calling one of its endpoints (i.e: gatekeeper_dns/read?req_type =random)
2. step 2 : Gatekeeper does a quick request sanitization and forwards it to the Trusted Host through the internal channel
3. step 3: Once the trusted host receives the request, will do more validation on it before forwarding it to the proxy for processing
4. step 4: When the Proxy receives the request, will process it according to its implementation described in section 2

5. step 5: Once the Proxy is done with processing the request will send the response in json format the Trusted Host, which will send it to the gatekeeper, and the gatekeeper will display it the client.

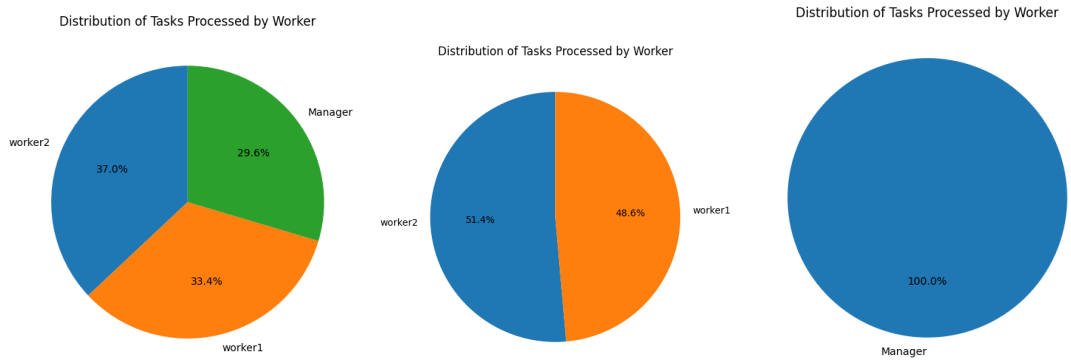
The implementation has two helper endpoints that help in monitoring the system's health which are */ping*, which will check if all the instances are up and running, while */tables_size* will help in counting the number of items inside each database.

5 Benchmarking the clusters.

To benchmark my implementation, I conducted four testing campaigns.

1. **Write Requests**: In the first campaign, 1000 simultaneous write requests were sent, and the database sizes were printed before and after processing. Initially, the size of all cluster databases was 0, but after completing the workload, the size became 1000 for all instances. This confirms that the manager successfully replicated the write operations across all worker instances.
2. **Direct Strategy**: The second campaign involved sending 1000 simultaneous read requests using the direct-hit strategy. Results showed that all 1000 requests were handled exclusively by the manager, verifying that this strategy relies entirely on the manager for read operations without involving the workers.
3. **Random Strategy**: In the third campaign, 1000 simultaneous read requests were processed using the random strategy. The workload was evenly distributed between the two workers, with Worker1 handling 486 requests and Worker2 processing the remaining 514. This demonstrates the random strategy's capability to balance the read workload effectively between workers.
4. **Custom Strategy**: The final campaign tested 1000 simultaneous read requests using the custom strategy. The results showed an uneven distribution, with the manager handling 296 requests, Worker1 processing 334 requests, and Worker2 handling 370 requests. This reflects Worker2's higher responsiveness and demonstrates the custom strategy's logic for allocating requests across the instances.

6 Summary of results



(a) Custom (b) Random (c) Direct

Figure 2: Tasks distribution for each 3 implementation

The workload distribution described in the three pie charts highlights the different strategies employed by the proxy implementations: **Custom**, **Random**, and **Direct**, and their respective impacts on request handling.

1. **Custom**: requests were distributed across the three instances, with Worker2 handling the largest share (370 requests, 37%), followed by Worker1 (334 requests, 33.4%), and the Manager handling the least (296 requests, 29.6%). This suggests that Worker2 was the most responsive, while the Manager was the least responsive, indicating a bias in workload distribution
2. **Random**: requests were almost evenly split between Worker1 and Worker2, with Worker1 processing 486 requests (48.6%) and Worker2 handling 514 requests (51.4%). The Manager did not process any requests, consistent with the random strategy selecting only between Worker1 and Worker2
3. **Direct**: 100% of the workload were assigned to the Manager, indicating a centralized handling approach with no involvement of the Workers.

   These distributions illustrate how different proxy strategies influence the balance and efficiency of resource utilization.

7 Instructions to run the code

To run our code, follow the instructions below:

- Clone the git repository of the project: ***git clone https://github.com/fomou/LOG8415-project.git***
- Navigate to the project folder: ***cd LOG8415-Project***
- Eventually, you may want to ensure that all necessary files are in the folder: ***ls***, should see: main.py, launch-script.sh, manager_user_data.sh, proxy_user_data.sh,exec_ssh.py, mysql_user_data.sh, and benchmark_worload.py
- Make **launch-script.sh** executable: ***chmod +x launch-script.sh***
- Execute the script by using either ***./launch-script.sh*** or ***source launch-script.sh***
- You may be prompted to enter your password as we are running sudo command, enter your password and hit enter to continue.