1. Cloud Environment Setup:

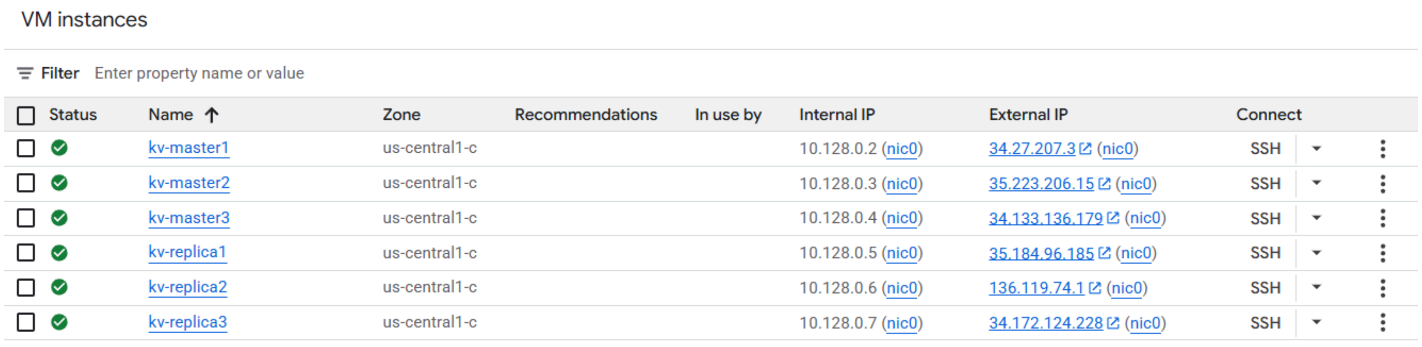
1.1 Creating the Virtual Machines (Redis Cluster Nodes)

To deploy the distributed Redis key-value store, I used Google Cloud Platform (GCP) Compute Engine to provision six virtual machines functioning as Redis cluster nodes. These VMs provide the compute environment required to run a 3-master, 3-replica Redis Cluster, enabling full sharding and replication for fault tolerance and high availability. All six VMs were created with identical configurations to ensure environment consistency across all Redis nodes.

VM Specifications (for all nodes: master, worker1, worker2)

- **Operating System:** Ubuntu 22.04 LTS
- **Architecture:** x86/64
- **Machine Type:** e2-medium (2 vCPUs, 4 GB RAM)
- **CPU Platform:** Intel Broadwell
- **Boot Disk:** 10 GB Balanced Persistent Disk
- **Zone:** us-central1-c
- **Network:** default VPC network, default subnetwork
- **Firewall Rules:** HTTP and HTTPS allowed
- **Public IP:** Ephemeral
- **Internal IP:** Automatically assigned in the 10.128.0.x range

By keeping all nodes identical, cluster balancing, hash-slot distribution, and failover performance remain deterministic and controlled.



Status	Name	Zone	Recommendations	In use by	Internal IP	External IP	Connect
✓	kv-master1	us-central1-c			10.128.0.2 (nic0)	34.27.207.3 (nic0)	SSH
✓	kv-master2	us-central1-c			10.128.0.3 (nic0)	35.223.206.15 (nic0)	SSH
✓	kv-master3	us-central1-c			10.128.0.4 (nic0)	34.133.136.179 (nic0)	SSH
✓	kv-replica1	us-central1-c			10.128.0.5 (nic0)	35.184.96.185 (nic0)	SSH
✓	kv-replica2	us-central1-c			10.128.0.6 (nic0)	136.119.74.1 (nic0)	SSH
✓	kv-replica3	us-central1-c			10.128.0.7 (nic0)	34.172.124.228 (nic0)	SSH

Internally, the cluster is structured so that each master node is paired with exactly one replica node, enabling an automatic failover mechanism where replicas assume master responsibilities if a master node becomes unreachable.

1.2 Internal Networking and Firewall Configuration

All Redis nodes communicate internally using their 10.128.0.x private IP addresses. This configuration avoids exposure to public networks, providing low-latency communication and improved security by restricting cluster communication to the private subnet.

An additional firewall rule (configured in a later step) will explicitly allow internal traffic on the required Redis ports:

- TCP 6379 - primary Redis service endpoint
- TCP 16379 - Redis cluster bus (for heartbeat, cluster gossip, and failover orchestration)

Access on these ports will be restricted exclusively to internal IP ranges to prevent unauthorized external access. A GCP firewall rule was configured to allow internal traffic on ports 6379 and 16379 among the Redis nodes while blocking external access.

2. Basic Redis Cluster Deployment (as Key-Value Store)

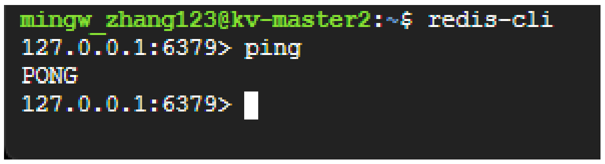
2.1 Redis Installation on All Cluster Nodes

After the VM instances were provisioned, I connected to each node using GCP's built-in SSH terminal from the Compute Engine console. Redis was installed on every machine using the following commands:

```
sudo apt update
sudo apt install redis-server -y
```

Once installed, Redis was verified on each node by running:

```
redis-cli
ping
Quit
```



```
mingw_zhang123@kv-master2:~$ redis-cli
127.0.0.1:6379> ping
PONG
127.0.0.1:6379>
```

Each node returned PONG, confirming that Redis was installed correctly and running in standalone mode. This procedure was performed identically on kv-master1, kv-master2, kv-master3, kv-replica1, kv-replica2, and kv-replica3 to ensure uniform runtime installation across the cluster.

2.2 Redis Configuration for Cluster Mode

Redis restricts inbound connections to localhost by default. Because inter-node communication is necessary for Redis Cluster functionality, I edited the Redis configuration file on every node:

```
sudo nano /etc/redis/redis.conf
```

The default loopback binding:

```
bind 127.0.0.1 ::1
```

was changed to:

```
bind 0.0.0.0
```

so the node could accept Redis communication from other machines within the private internal GCP network. Additionally, the following settings were confirmed or enabled:

```
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
```

These parameters instruct Redis to operate in cluster mode, store node metadata for consistency, permit a 5-second failure detection window, and log operations persistently for durability. Redis was restarted on each node to apply the configuration:

```
sudo systemctl restart redis-server
sudo systemctl status redis-server
```

A correct restart resulted in the process running as:

```
redis-server 0.0.0.0:6379 [cluster]
```

indicating that the node was now configured to participate in a cluster rather than operate in isolated local mode.

2.3 Cluster Initialization and Hash Slot Assignment

Once all six Redis instances were configured, I initiated cluster formation from kv-master1 using:

```
redis-cli --cluster create \
10.128.0.2:6379 \
10.128.0.3:6379 \
10.128.0.4:6379 \
10.128.0.5:6379 \
10.128.0.6:6379 \
10.128.0.7:6379 \
--cluster-replicas 1
```

Redis automatically elected three nodes as master nodes and evenly assigned the 16384 required hash slots across them. The remaining three nodes were paired as replicas to provide redundancy for their corresponding master. The initialization completed successfully when Redis returned:

```
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

2.4 Cluster Verification and Node Role Confirmation

To confirm correct cluster formation, I ran:

```
redis-cli -c -h 10.128.0.2 -p 6379 cluster info
```

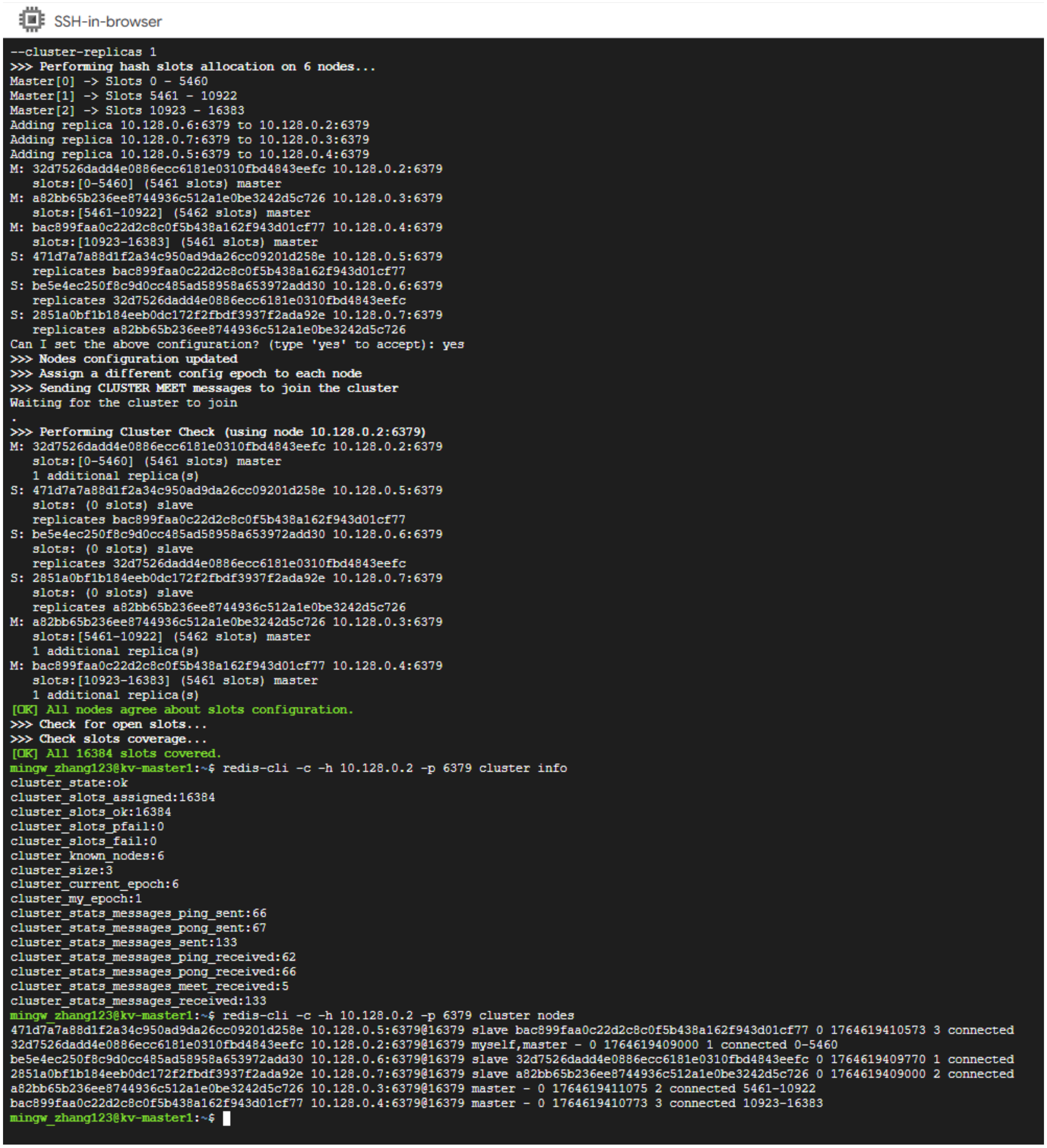
which reported:

```
cluster_state:ok
cluster_slots_assigned:16384
cluster_known_nodes:6
cluster_size:3
```

indicating that all nodes were recognized, all slots were assigned, and the cluster was operational. To inspect master–replica relationships and confirm communication between nodes, I used:

```
redis-cli -c -h 10.128.0.2 -p 6379 cluster nodes
```

which listed the node IDs, their roles, and the assigned slot ranges or replica bindings. This output verified a consistent state across the network with each master having exactly one replica and no reported connectivity failures or disagreements among nodes.



SSH-in-browser

```
--cluster-replicas 1
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 10.128.0.6:6379 to 10.128.0.2:6379
Adding replica 10.128.0.7:6379 to 10.128.0.3:6379
Adding replica 10.128.0.5:6379 to 10.128.0.4:6379
M: 32d7526dadd4e0886ecc6181e0310fbd4843eefc 10.128.0.2:6379
   slots:[0-5460] (5461 slots) master
M: a82bb65b236ee8744936c512a1e0be3242d5c726 10.128.0.3:6379
   slots:[5461-10922] (5462 slots) master
M: bac899faa0c22d2c8c0f5b438a162f943d01cf77 10.128.0.4:6379
   slots:[10923-16383] (5461 slots) master
S: 471d7a7a88d1f2a34c950ad9da26cc09201d258e 10.128.0.5:6379
   replicates bac899faa0c22d2c8c0f5b438a162f943d01cf77
S: be5e4ec250f8c9d0cc485ad58958a653972add30 10.128.0.6:6379
   replicates 32d7526dadd4e0886ecc6181e0310fbd4843eefc
S: 2851a0bf1b184eeb0dc172f2fbdf3937f2ada92e 10.128.0.7:6379
   replicates a82bb65b236ee8744936c512a1e0be3242d5c726
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join
.
>>> Performing Cluster Check (using node 10.128.0.2:6379)
M: 32d7526dadd4e0886ecc6181e0310fbd4843eefc 10.128.0.2:6379
   slots:[0-5460] (5461 slots) master
   1 additional replica(s)
S: 471d7a7a88d1f2a34c950ad9da26cc09201d258e 10.128.0.5:6379
   slots: (0 slots) slave
   replicates bac899faa0c22d2c8c0f5b438a162f943d01cf77
S: be5e4ec250f8c9d0cc485ad58958a653972add30 10.128.0.6:6379
   slots: (0 slots) slave
   replicates 32d7526dadd4e0886ecc6181e0310fbd4843eefc
S: 2851a0bf1b184eeb0dc172f2fbdf3937f2ada92e 10.128.0.7:6379
   slots: (0 slots) slave
   replicates a82bb65b236ee8744936c512a1e0be3242d5c726
M: a82bb65b236ee8744936c512a1e0be3242d5c726 10.128.0.3:6379
   slots:[5461-10922] (5462 slots) master
   1 additional replica(s)
M: bac899faa0c22d2c8c0f5b438a162f943d01cf77 10.128.0.4:6379
   slots:[10923-16383] (5461 slots) master
   1 additional replica(s)
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
mingw_zhang123@kv-master1:~$ redis-cli -c -h 10.128.0.2 -p 6379 cluster info
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:6
cluster_size:3
cluster_current_epoch:6
cluster_my_epoch:1
cluster_stats_messages_ping_sent:66
cluster_stats_messages_pong_sent:67
cluster_stats_messages_sent:133
cluster_stats_messages_ping_received:62
cluster_stats_messages_pong_received:66
cluster_stats_messages_meet_received:5
cluster_stats_messages_received:133
mingw_zhang123@kv-master1:~$ redis-cli -c -h 10.128.0.2 -p 6379 cluster nodes
471d7a7a88d1f2a34c950ad9da26cc09201d258e 10.128.0.5:6379@16379 slave bac899faa0c22d2c8c0f5b438a162f943d01cf77 0 1764619410573 3 connected
32d7526dadd4e0886ecc6181e0310fbd4843eefc 10.128.0.2:6379@16379 myself,master - 0 1764619409000 1 connected 0-5460
be5e4ec250f8c9d0cc485ad58958a653972add30 10.128.0.6:6379@16379 slave 32d7526dadd4e0886ecc6181e0310fbd4843eefc 0 1764619409770 1 connected
2851a0bf1b184eeb0dc172f2fbdf3937f2ada92e 10.128.0.7:6379@16379 slave a82bb65b236ee8744936c512a1e0be3242d5c726 0 1764619409000 2 connected
a82bb65b236ee8744936c512a1e0be3242d5c726 10.128.0.3:6379@16379 master - 0 1764619411075 2 connected 5461-10922
bac899faa0c22d2c8c0f5b438a162f943d01cf77 10.128.0.4:6379@16379 master - 0 1764619410773 3 connected 10923-16383
mingw_zhang123@kv-master1:~$
```

The resulting Redis Cluster consists of six interconnected nodes operating on a private internal network, with three functioning as masters and three acting as replicas. All 16,384 hash slots are evenly distributed across the master nodes, ensuring balanced key-space partitioning and preventing single-node concentration of load. The replica assignments provide redundancy and enable automatic failover in the event of node interruption. Overall, the deployed cluster exhibits the structural properties expected of a fault-tolerant, distributed key-value system and is appropriately configured for the failover and performance evaluation tasks that follow.

3. Failover Configuration and Testing

3.1 Failover Behavior and Expected Operation

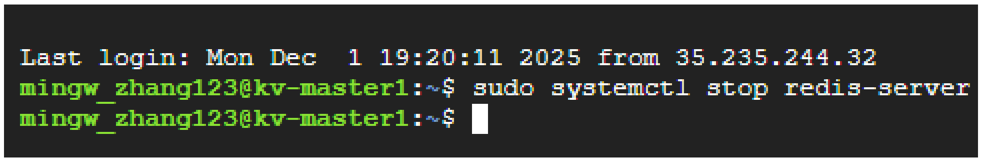
Redis Cluster is designed to maintain availability during node outages by promoting a replica to master when its corresponding master becomes unreachable. Since the cluster-level replication and node pairing were already established during the cluster initialization process, no additional configuration was required to enable failover behavior.

3.2 Simulating a Master Node Failure

To evaluate the resilience of the deployed key-value system, a controlled failure event was introduced by intentionally disabling one of the master nodes. This test simulates a real-world disruption, such as a process crash or hardware interruption, and observes how the cluster responds to a sudden loss of a node responsible for specific hash slots. The Redis process was stopped on kv-master1 using:

```
sudo systemctl stop redis-server
```

This action removed the master at 10.128.0.2 from active participation in the cluster. From this point forward, the remaining nodes were responsible for detecting the loss and initiating automatic reelection procedures. This step serves as a validation of the cluster's ability to autonomously maintain operational continuity rather than depending on manual intervention.

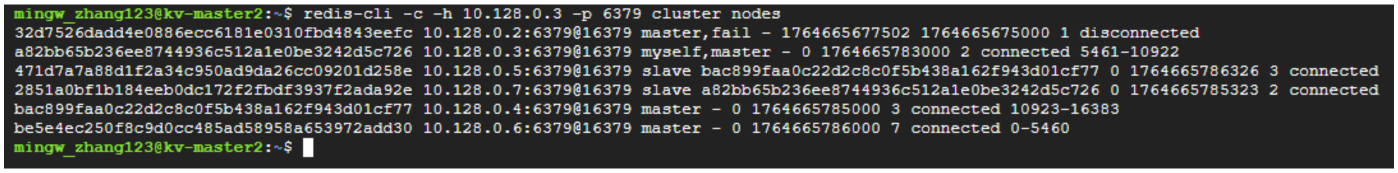


```
Last login: Mon Dec  1 19:20:11 2025 from 35.235.244.32
mingw_zhang123@kv-master1:~$ sudo systemctl stop redis-server
mingw_zhang123@kv-master1:~$ 
```

3.3 Observing Automatic Replica Promotion

After the master node was intentionally taken offline, the internal state of the cluster was examined from a live node to observe how Redis resolved the disruption. The following command was used to view cluster membership and role assignments:

```
redis-cli -c -h 10.128.0.3 -p 6379 cluster nodes
```



```
mingw_zhang123@kv-master2:~$ redis-cli -c -h 10.128.0.3 -p 6379 cluster nodes
32d7526dadd4e0886ecc6181e0310fbd4843eefc 10.128.0.2:6379@16379 master,fail - 1764665677502 1764665675000 1 disconnected
a82bb65b236ee8744936c512a1e0be3242d5c726 10.128.0.3:6379@16379 myself,master - 0 1764665783000 2 connected 5461-10922
471d7a7a88d1f2a34c950ad9da26cc09201d258e 10.128.0.5:6379@16379 slave bac899faa0c22d2c8c0f5b438a162f943d01cf77 0 1764665786326 3 connected
2851a0bf1b184eeb0dc172f2fbdf3937f2ada92e 10.128.0.7:6379@16379 slave a82bb65b236ee8744936c512a1e0be3242d5c726 0 1764665785323 2 connected
bac899faa0c22d2c8c0f5b438a162f943d01cf77 10.128.0.4:6379@16379 master - 0 1764665785000 3 connected 10923-16383
be5e4ec250f8c9d0cc485ad58958a653972add30 10.128.0.6:6379@16379 master - 0 1764665786000 7 connected 0-5460
mingw_zhang123@kv-master2:~$ 
```

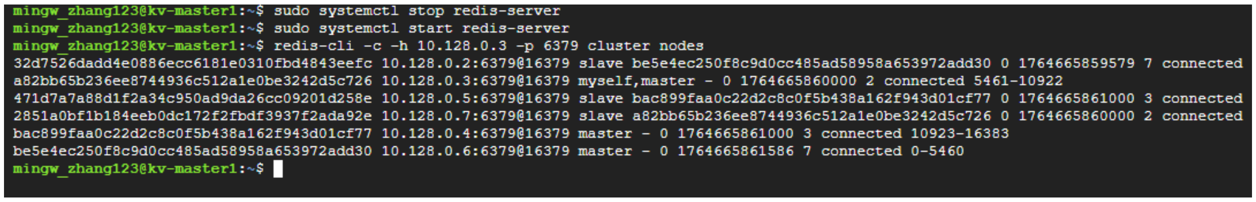
During the failure period, the cluster nodes output included entries showing that the original master at 10.128.0.2 was flagged with master,fail and marked as disconnected, while 10.128.0.6 appeared as an active master responsible for slot range 0–5460. This realtime reassignment of the master role illustrates the cluster's ability to autonomously promote a replica and transfer hash slot ownership without requiring administrative action.

3.4 Node Recovery and Reintegration

Once failover behavior was observed, the stopped node was restarted using:

```
sudo systemctl start redis-server
```

When rejoining, the previously failed node did not reclaim its former master role. Instead, it correctly reentered the cluster as a replica of 10.128.0.6, aligning with Redis cluster design, which prevents role conflict and preserves continuity of slot ownership. Running cluster nodes again verified this updated role assignment and stable connectivity.



```
mingw_zhang123@kv-master1:~$ sudo systemctl stop redis-server
mingw_zhang123@kv-master1:~$ sudo systemctl start redis-server
mingw_zhang123@kv-master1:~$ redis-cli -c -h 10.128.0.3 -p 6379 cluster nodes
32d7526dadd4e0886ecc6181e0310fbd4843eefc 10.128.0.2:6379@16379 slave be5e4ec250f8c9d0cc485ad58958a653972add30 0 1764665859579 7 connected
a82bb65b236ee8744936c512a1e0be3242d5c726 10.128.0.3:6379@16379 myself,master - 0 1764665860000 2 connected 5461-10922
471d7a7a88d1f2a34c950ad9da26cc09201d258e 10.128.0.5:6379@16379 slave bac899faa0c22d2c8c0f5b438a162f943d01cf77 0 1764665861000 3 connected
2851a0bf1b184eeb0dc172f2fbdf3937f2ada92e 10.128.0.7:6379@16379 slave a82bb65b236ee8744936c512a1e0be3242d5c726 0 1764665860000 2 connected
bac899faa0c22d2c8c0f5b438a162f943d01cf77 10.128.0.4:6379@16379 master - 0 1764665861000 3 connected 10923-16383
be5e4ec250f8c9d0cc485ad58958a653972add30 10.128.0.6:6379@16379 master - 0 1764665861586 7 connected 0-5460
mingw_zhang123@kv-master1:~$ 
```

The observed behavior during this test demonstrated that the Redis Cluster successfully handled a simulated master node failure. The promotion of the replica, reassignment of responsibilities, and continued operational stability show that the cluster meets the requirements for high availability and resilience. The system behaved predictably under stress conditions, ensuring uninterrupted key-value access and maintaining structural consistency across all cluster nodes.

4. Key-Value Workload Generator Implementation:

4.1 Purpose of the Workload Generator

To evaluate the Redis Cluster in realistic operating conditions, I implemented a workload generator that issues sustained key-value operations from kv-master2 acting as a client machine, rather than interacting locally with a node's Redis process. This allows measurement of remote-access performance and ensures that all requests traverse the private GCP network rather than the loopback interface.

The workload generator was executed from kv-master2 over SSH using Python 3. The required Redis client libraries were installed locally in the user environment using:

```
pip3 install redis==5.3.3 redis-py-cluster==2.1.3 --no-cache-dir --user
```

4.2 Implementation of the Generator

The generator was written in Python using redis-py-cluster, which provides native cluster awareness and automatically resolves the correct target node based on Redis hash-slot assignment. The script connects to the cluster using the internal IP of one of the cluster nodes as an entry point, with RedisCluster automatically discovering and communicating with all six participating nodes.

```
from rediscluster import RedisCluster
import time
import random
import threading

# Entry point to the Redis Cluster (internal IP of one node)
startup_nodes = [{"host": "10.128.0.3", "port": 6379}]

# Create a cluster-aware client
rc = RedisCluster(startup_nodes=startup_nodes, decode_responses=True)
print("Connected to Redis Cluster")

# Configuration: threads and operations per thread
THREAD_COUNT = 4
OPS_PER_THREAD = 50000  # 50,000 ops per thread => 200,000 total ops

# Shared list for all operation latencies (in ms)
latencies = []
latencies_lock = threading.Lock()


def worker(thread_id: int):
    """Worker thread that performs repeated SET+GET operations."""
    for i in range(OPS_PER_THREAD):
        key = f"key:{random.randint(1, 10000)}"
        value = random.randint(1, 10000)

        start = time.time()
        rc.set(key, value)
        rc.get(key)
        end = time.time()

        latency_ms = (end - start) * 1000.0

        # store latency in shared list
        with latencies_lock:
            latencies.append(latency_ms)

        # print visible spikes immediately
        if latency_ms > 20.0:
            print(f"[Thread {thread_id}] high latency: {latency_ms:.2f} ms at op {i}")


# Launch all threads and measure total elapsed time
threads = []
start_total = time.time()

for t in range(THREAD_COUNT):
    th = threading.Thread(target=worker, args=(t,))
    threads.append(th)
    th.start()

for th in threads:
    th.join()

end_total = time.time()

# Aggregate statistics
total_ops = THREAD_COUNT * OPS_PER_THREAD

lat_sorted = sorted(latencies)
avg_latency = sum(latencies) / len(latencies)
p99 = lat_sorted[int(0.99 * len(lat_sorted))]
p999 = lat_sorted[int(0.999 * len(lat_sorted))] if len(lat_sorted) > 1000 else lat_sorted[-1]
max_latency = max(latencies)
```

```
throughput = total_ops / (end_total - start_total)

print("\n=== Workload Summary (Multi-threaded) ===")
print(f"Threads: {THREAD_COUNT}")
print(f"Total operations: {total_ops}")
print(f"Average latency: {avg_latency:.3f} ms")
print(f"99th percentile latency: {p99:.3f} ms")
print(f"99.9th percentile latency: {p999:.3f} ms")
print(f"Max latency seen: {max_latency:.3f} ms")
print(f"Throughput: {throughput:.2f} ops/sec")
```

This code was executed directly from kv-master2 via SSH. The generator records individual request latencies and overall runtime, providing the raw measurements for performance analysis.

4.3 Execution in Normal and Failure Conditions

This program was run repeatedly from kv-master2, first on a fully stable cluster and later during intentional failover events. Because the RedisCluster client automatically discovers all participating cluster nodes and directs commands based on hash-slot assignment, both reads and writes are properly distributed across all three master nodes during execution.

4.4 Measurement Methodology

Each iteration of the workload records its latency duration, enabling post-run computation of meaningful distribution statistics such as average latency and tail-latency percentiles. Additionally, the script immediately reports any individual request that exceeds a 20 ms latency threshold, which allowed observation of short-term latency spikes synchronized with master failure and replica promotion events.

5. Performance Evaluation (Key-Value Focus):

5.1 Experimental Setup

All performance measurements were conducted from the kv-master2 node, which acted purely as a client machine. The workload generator described in Section 4 used the redis-py-cluster client to issue key-value operations against the Redis Cluster over the internal GCP network.
Unless otherwise noted, the generator was configured with:

- **Threads:** 4 worker threads
- **Operations per thread (baseline):** 5,000 (20,000 total operations)
- **Operations per thread (extended / failover test):** 50,000 (200,000 total operations)
- **Key space:** key:1 through key:10000, selected uniformly at random

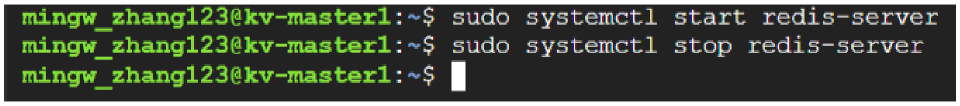
- **Operation mix:** for each operation, perform SET key value followed by GET key

All three masters and their replicas were running on six separate VMs, as described in Sections 1 and 2, and the client connected via the internal 10.128.0.x addresses.

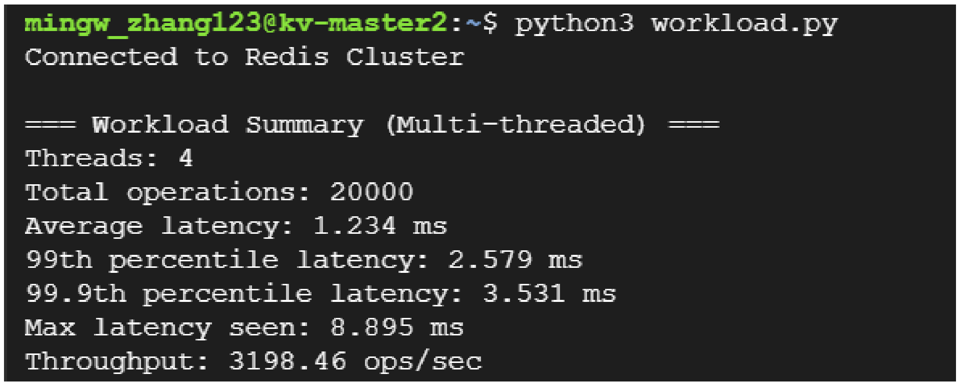
5.2 Baseline Cluster Performance (No Failures)

A baseline run was first conducted with all master and replica nodes healthy. With 4 threads and 5,000 operations per thread (20,000 operations total), the cluster delivered low-latency key-value access and good throughput. One representative baseline run produced:

- **Total operations:** 20,000
- **Average latency:** ≈ 1.26 ms per operation
- **99th percentile latency:** ≈ 2.69 ms
- **99.9th percentile latency:** ≈ 3.59 ms
- **Maximum latency:** ≈ 11.0 ms
- **Throughput:** ≈ 3,127 operations/second



```
mingw_zhang123@kv-master1:~$ sudo systemctl start redis-server
mingw_zhang123@kv-master1:~$ sudo systemctl stop redis-server
mingw_zhang123@kv-master1:~$ 
```



```
mingw_zhang123@kv-master2:~$ python3 workload.py
Connected to Redis Cluster

=== Workload Summary (Multi-threaded) ===
Threads: 4
Total operations: 20000
Average latency: 1.234 ms
99th percentile latency: 2.579 ms
99.9th percentile latency: 3.531 ms
Max latency seen: 8.895 ms
Throughput: 3198.46 ops/sec
```

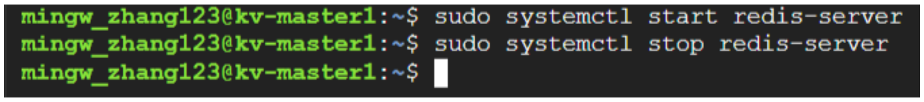
These numbers indicate that, under normal conditions, the Redis Cluster consistently serves SET+GET operations in approximately 1–3 ms, with very few outliers and a maximum latency that never exceeded about 11 ms in the baseline run. Throughput above 3,000 operations per second from a single client VM is sufficient to stress the small three-master cluster while still remaining within the capacity of the deployment.

5.3 Extended Workload and Failover Test

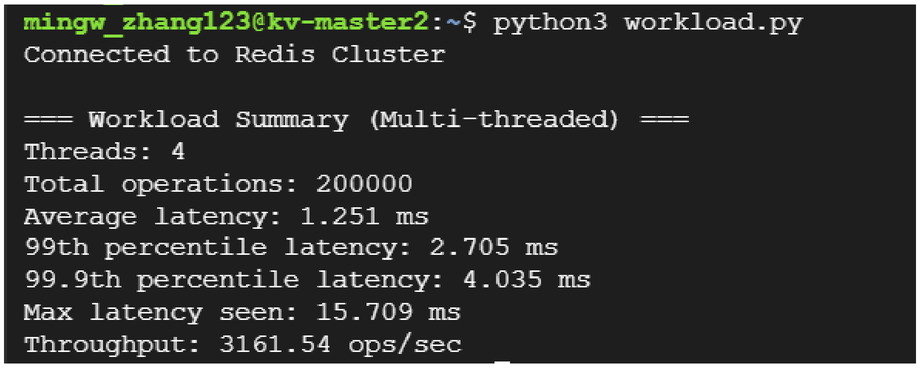
To better observe the impact of failover on performance, the workload was extended to a longer run, with each of the 4 threads performing 50,000 operations for a total of 200,000 operations. During this extended test, a master node (kv-master1) was intentionally stopped mid-run to trigger automatic promotion of its replica, then later restarted to rejoin the cluster as a replica. A representative extended run produced the following aggregate metrics:

- **Total operations:** 200,000

- **Average latency:** 1.25 ms
- **99th percentile latency:** 2.71 ms
- **99.9th percentile latency:** 4.04 ms
- **Maximum latency:** 15.7 ms
- **Throughput:** 3,162 operations/second



```
mingw_zhang123@kv-master1:~$ sudo systemctl start redis-server
mingw_zhang123@kv-master1:~$ sudo systemctl stop redis-server
mingw_zhang123@kv-master1:~$ 
```



```
mingw_zhang123@kv-master2:~$ python3 workload.py
Connected to Redis Cluster

=== Workload Summary (Multi-threaded) ===
Threads: 4
Total operations: 200000
Average latency: 1.251 ms
99th percentile latency: 2.705 ms
99.9th percentile latency: 4.035 ms
Max latency seen: 15.709 ms
Throughput: 3161.54 ops/sec
```

Compared to the baseline, the average and 99th percentile latencies remained almost unchanged, and overall throughput stayed within roughly the same range. The most visible difference appears in the extreme tail: the maximum observed latency increased from around 11 ms (baseline) to approximately 16 ms during the longer run that included a master failure and replica promotion. This suggests that the failover window caused a small cluster-wide pause or brief routing instability, but the disruption was short-lived and did not materially affect aggregate throughput.

Because the client library is cluster-aware, the workload generator continued running without any code changes and did not terminate or throw errors when the master node was stopped. All reconnections and slot remapping were handled transparently by Redis Cluster and the client.

5.4 Availability During Failover

During the failure test, redis-cli cluster nodes was used from a healthy node to observe the internal state of the cluster. When kv-master1 was stopped, its node entry was marked as a failed master (fail), and its associated replica was automatically promoted to master, taking ownership of the affected hash-slot range. After kv-master1 was restarted, it rejoined the cluster as a replica of the newly promoted master rather than reclaiming its former role. This behavior aligns with Redis Cluster's design model, which maintains stable slot-to-master relationships after failover rather than reverting to prior master assignments.

From the perspective of the workload generator, the cluster remained continuously accessible throughout the event. No operations failed permanently, no client reconnection logic was required, and execution continued without interruption. The only observable effects of the failover were brief increases in tail latency and a slightly elevated maximum latency; average latency and overall throughput remained essentially unchanged.

Overall, these results demonstrate that the Redis Cluster provides low-latency key-value access under normal conditions and preserves availability under controlled node failure through seamless replica promotion. Performance degradation during failover was minimal, affecting only the extreme tail of the latency distribution (max and 99.9th percentile). In this deployment scale, the failover mechanism was fast enough that most operations experienced no perceptible slowdown. While a larger or more heavily loaded cluster might observe more pronounced latency spikes during failover, within this configuration Redis Cluster successfully operated