

Victor Martinez

CSCI 346: Cloud Computing (Spring 2025)

Due Date: 4/23/25

### **Project 3: Key-Value Store**

In this project, I will rapidly deploy, configure, and evaluate a distributed key-value store using Redis Cluster on a cloud platform. The focus of this project is to understand and implement essential concepts for scalable key-value storage: sharding for data distribution, replication for durability/read scaling, high availability through failover, and basic performance testing under typical key-value workloads.

#### **Project Structure:**

##### **Task 1:** Cloud Environment Setup:

- Select a cloud provider.
- Provision 3-6 virtual machines or container instances.
- Configure basic private networking and necessary security group rules to allow Redis communication between nodes.

##### **Task 2:** Basic Redis Cluster Deployment:

- Install Redis on all provisioned nodes.
- Use `redis-cli --cluster create` (or an equivalent method) to initialize a Redis Cluster. Aim for at least 3 master nodes and configure a replication factor of at least 1 to ensure key durability and read availability.
- Verify the cluster status and ensure all nodes are correctly joined and hash slots (determining key placement) are assigned.

##### **Task 3:** Failover Configuration and Testing (for Key Availability):

- Confirm that Redis Cluster is configured for automatic failover to maintain continuous access to keys.
- Perform a failover test: Manually stop one of the master nodes responsible for a subset of keys.
- Observe the cluster's reaction using cluster nodes or monitoring logs. Verify that a replica is promoted to master status, taking over responsibility for its assigned keys.
- Check that the cluster remains operational for clients.
- Restart the stopped node and verify it rejoins the cluster as a replica.

##### **Task 4:** Key-Value Workload Generator Implementation:

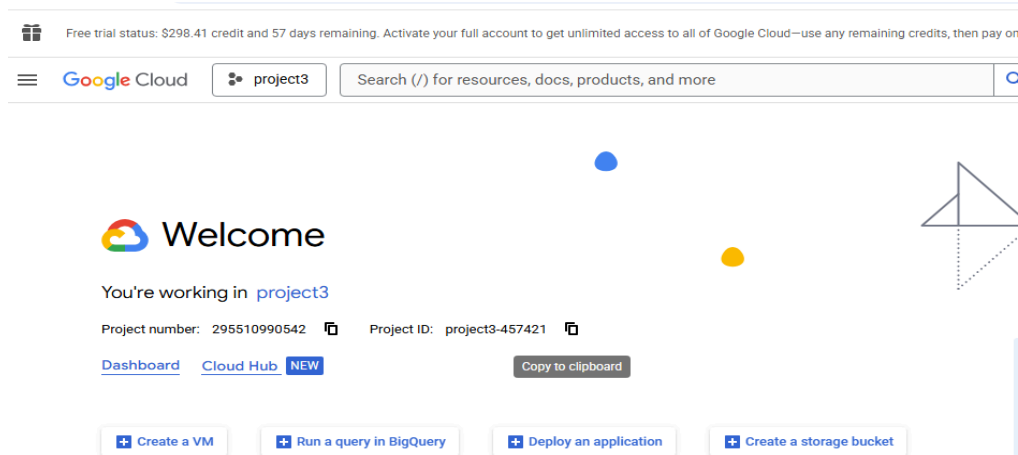
- Develop or adapt a simple workload generator program focusing on simulating key-value access patterns.
- Ensure the generator can connect to the Redis cluster.
- Implement logic primarily performing high-frequency, basic key operations like SET, GET, and potentially DEL or INCR, targeting random keys to distribute the load across shards. Usage of more complex data structures should be minimal unless justified for representing structured values.
- Incorporate concurrency to simulate multiple clients accessing keys simultaneously.
- Add basic functionality to measure latency per key operation and calculate overall throughput

##### **Task 5:** Performance Evaluation

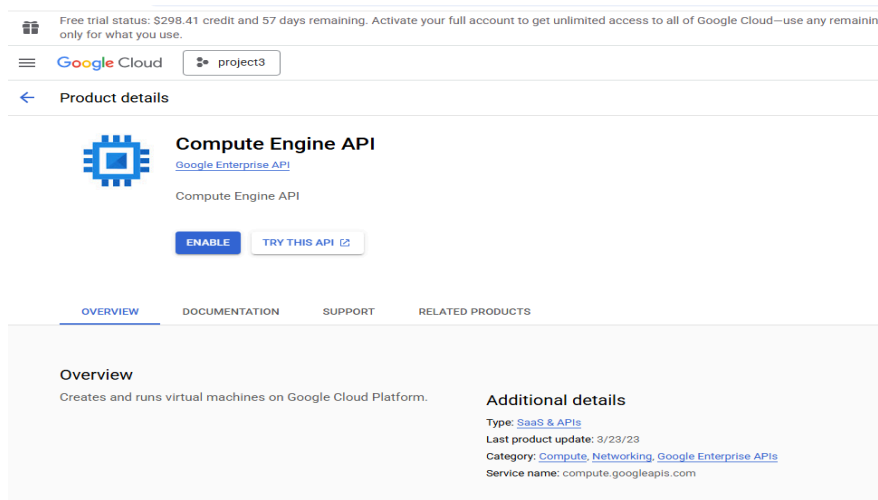
##### **Task 6:** Reporting and Finalization:

## Task 1: Cloud Environment Setup:

- For this project I decided to use Google Cloud Platform (GCP):



- I then downloaded the Computer Engine API so that my project can create the VM instances.



•The next step is to create 6 VM instances. I used e2-medium with an OS image of Debian GNU due to its lightweight, secure but yet stable deployments. Regarding networking I allowed HTTP traffic and HTTPS which is not needed for the Redis cluster setup but for any possible future web app hosting would come in handy.

Series	Description	vCPUs	Memory	CPU Platform
C4	Consistently high performance	2 - 192	4 - 1,488 GB	Intel Emerald Ra
C4A	Arm-based consistently high performance	1 - 72	2 - 576 GB	Google Axion
C4D	Consistently high performance	2 - 384	3 - 3,024 GB	AMD Turin
N4	Flexible & cost-optimized	2 - 80	4 - 640 GB	Intel Emerald Ra
C3	Consistently high performance	4 - 192	8 - 1,536 GB	Intel Sapphire Ra
C3D	Consistently high performance	4 - 360	8 - 2,880 GB	AMD Deno
E2	Low cost, day-to-day computing	0.25 - 32	1 - 128 GB	Intel Broadwell
N2	Balanced price & performance	2 - 128	2 - 864 GB	Intel Cascade La
N2D	Balanced price & performance	2 - 224	2 - 896 GB	AMD Milan
T2A	Scale-out workloads	1 - 48	4 - 192 GB	Ampere Altra
T2D	Scale-out workloads	1 - 60	4 - 240 GB	AMD Milan
N1	Balanced price & performance	0.25 - 96	0.6 - 624 GB	Intel Haswell

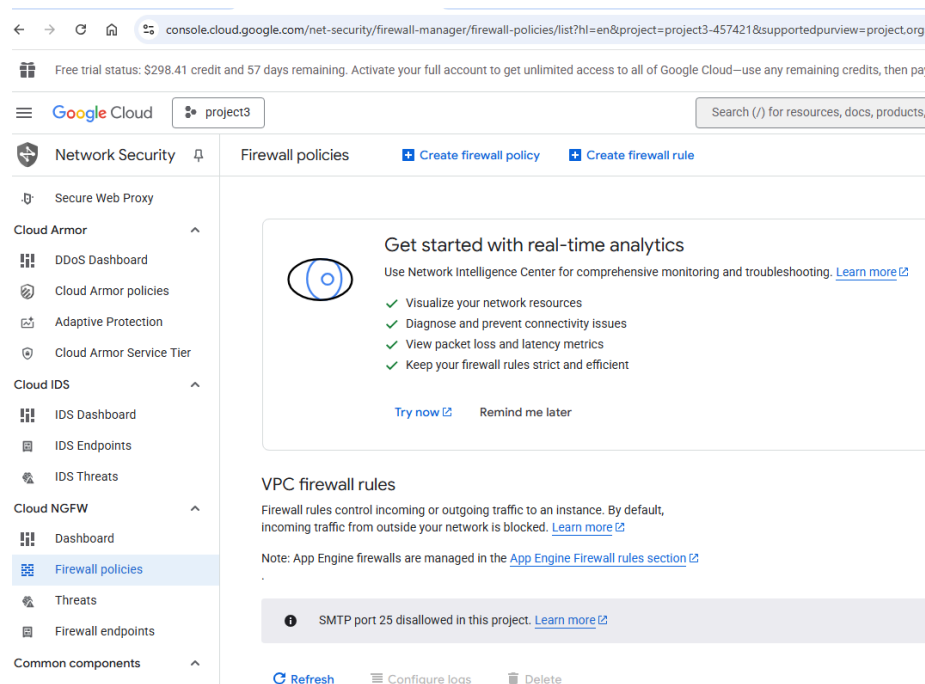
•The final set up looks like the following with six healthy running VM instances. A faster way to create 6 nodes is through A GCP console using a simple for loop.

Status	Name	Zone	Recommendations	In use by	Internal IP	External IP	Connections
Running	redis-node-1	us-central1-c			10.128.0.2 (nic0)	34.10.72.230 (nic0)	SSH
Running	redis-node-2	us-central1-c			10.128.0.3 (nic0)	107.178.222.158 (nic0)	SSH
Running	redis-node-3	us-central1-c			10.128.0.4 (nic0)	34.31.116.86 (nic0)	SSH
Running	redis-node-4	us-central1-c			10.128.0.5 (nic0)	34.172.95.8 (nic0)	SSH
Running	redis-node-5	us-central1-c			10.128.0.6 (nic0)	146.148.60.167 (nic0)	SSH
Running	redis-node-6	us-central1-c			10.128.0.7 (nic0)	35.232.192.113 (nic0)	SSH

- Now when dealing with Redis communication between nodes you need to set up a private network with security group rules. This is done because the redis cluster splits data across multiple nodes through sharding where these nodes will be communicating about how to deal with failovers and how to sync replicas with masters. The setup can be found under Network Security followed by firewall policies.

- Private networking brings secure and low latency communication, and for this to happen the redis nodes need to be on the same virtual private cloud(VPC) while using internal Ips which is shown in the image above.

- To ensure this you need to navigate to the Network Security and locate firewall policies. It should look like the following:



- Creating a firewall rule looks like the following:

- I named the rule “redis-allow” for easy understanding and then set the direction of traffic to Ingress to control the incoming traffic to my VM instances. I want my Redis nodes to be able to communicate through both the internal and client connections making ingress the correct direction.

[←](#) Create a firewall rule

Firewall rules control incoming or outgoing traffic to an instance. By default, incoming traffic from outside your network is blocked. [Learn more](#)

Name \*

redis-allow

Lowercase letters, numbers, hyphens allowed

Description

Logs

Turning on firewall logs can generate a large number of logs which can increase costs in Logging. [Learn more](#)

☐ On

☒ Off

Network \*

default

Priority \*

1000

[Compare](#)

Priority can be 0 - 65535

Direction of traffic

☒ Ingress

☐ Egress

- Next I set Targets to “all instances in the network” to allow all the VMs in my network to communicate with each other. I then set the source IP ranges to 10.128.0.0/9 because Google Cloud VPC networks typically use private IP address ranges for internal communication between resources.

Targets

All instances in the network

Source filter

IPv4 ranges

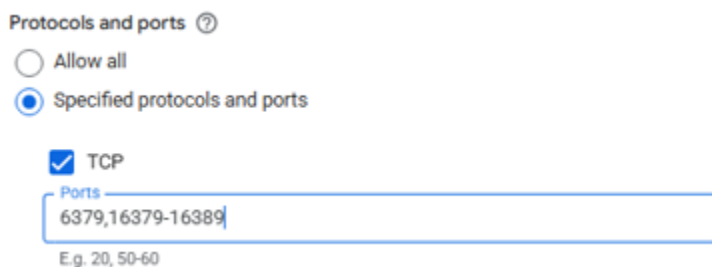
Source IPv4 ranges \*

10.128.0.0/9 × for example, 0.0.0.0/0, 192.168.2.0/24

Second source filter

None

- I then configured the protocol and ports. Now I choose “Specified protocols and ports” and checked TCP with the ports of 6379, and 16379-65535 inside my firewall rules to ensure that Redis can properly communicate through the cluster with clients.
- TCP port 6379 is the default port for Redis server client communication making any application whether internal or external be able to interact with Redis and always connect to this port.
- TCP:16379-65535 is the Redis cluster ports with the range being used for internal cluster communication between the cluster nodes. This is important because the nodes need to be able to communicate as mentioned before and with this range clusters are given multiple ports to use.
- Redis can also use these ports for rebalancing slots and for replicas as well.



Protocols and ports ?

☐ Allow all

☒ Specified protocols and ports

☒ TCP

Ports

6379,16379-16389

E.g. 20, 50-60

## **Task 2: Basic Redis Cluster Deployment:**

- For task 2 I needed to set up a Redis cluster with 3 masters and 3 replicas. Now when dealing with this project using an older version of Redis brought more trial and error. So to overcome this, I made sure to download the latest version of Redis using the following commands on each node to ensure that all nodes have Redis installed:

```
sudo add-apt-repository ppa:redislabs/redis
sudo apt update
sudo apt install -y redis
```

- This command simply adds the Redis Labs personal package archive to my systems package manager while telling the advanced package tool to update and finally telling the APT to install.

- I was met with the following output of a successful redis download.

```
vmop232@redis-node-1:~$ sudo add-apt-repository ppa:redislabs/redis
sudo apt update
sudo apt install -y redis
sudo: add-apt-repository: command not found
Get:1 file:/etc/apt/mirrors/debian.list Mirrorlist [30 B]
Get:2 file:/etc/apt/mirrors/debian-security.list Mirrorlist [39 B]
Hit:7 https://packages.cloud.google.com/apt google-compute-engine-bookworm-stable InRelease
Hit:8 https://packages.cloud.google.com/apt cloud-sdk-bookworm InRelease
Hit:3 https://deb.debian.org/debian bookworm InRelease
Hit:4 https://deb.debian.org/debian bookworm-updates InRelease
Hit:9 https://packages.cloud.google.com/apt google-cloud-ops-agent-bookworm-2 InRelease
Hit:5 https://deb.debian.org/debian bookworm-backports InRelease
Hit:6 https://deb.debian.org/debian-security bookworm-security InRelease
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
All packages are up to date.
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  libatomic1 liblzfl redis-server redis-tools
Suggested packages:
  ruby-redis
The following NEW packages will be installed:
  libatomic1 liblzfl redis redis-server redis-tools
0 upgraded, 5 newly installed, 0 to remove and 0 not upgraded.
Need to get 1107 kB of archives.
After this operation, 6284 kB of additional disk space will be used.
Get:1 file:/etc/apt/mirrors/debian.list Mirrorlist [30 B]
Get:2 https://deb.debian.org/debian bookworm/main amd64 libatomic1 amd64 12.2.0-14 [9328 B]
Get:3 https://deb.debian.org/debian bookworm/main amd64 liblzfl amd64 3.6-3 [10.2 kB]
Get:4 https://deb.debian.org/debian bookworm/main amd64 redis-tools amd64 5:7.0.15-1~deb12u3 [990 kB]
```

- I then used this command to ensure that I was using the latest model of Redis on each of my nodes.

```
redis-server --version
```

- I was met with the following output for each node.

```
vmop232@redis-node-1:~$ redis-server --version
Redis server v=7.0.15 sha=00000000:0 malloc=jemalloc-5.3.0 bits=64 build=c89c70d1d28059e4
vmop232@redis-node-1:~$ sudo nano /etc/redis/redis.conf
```

- Next I needed to edit the Redis config file on each node.

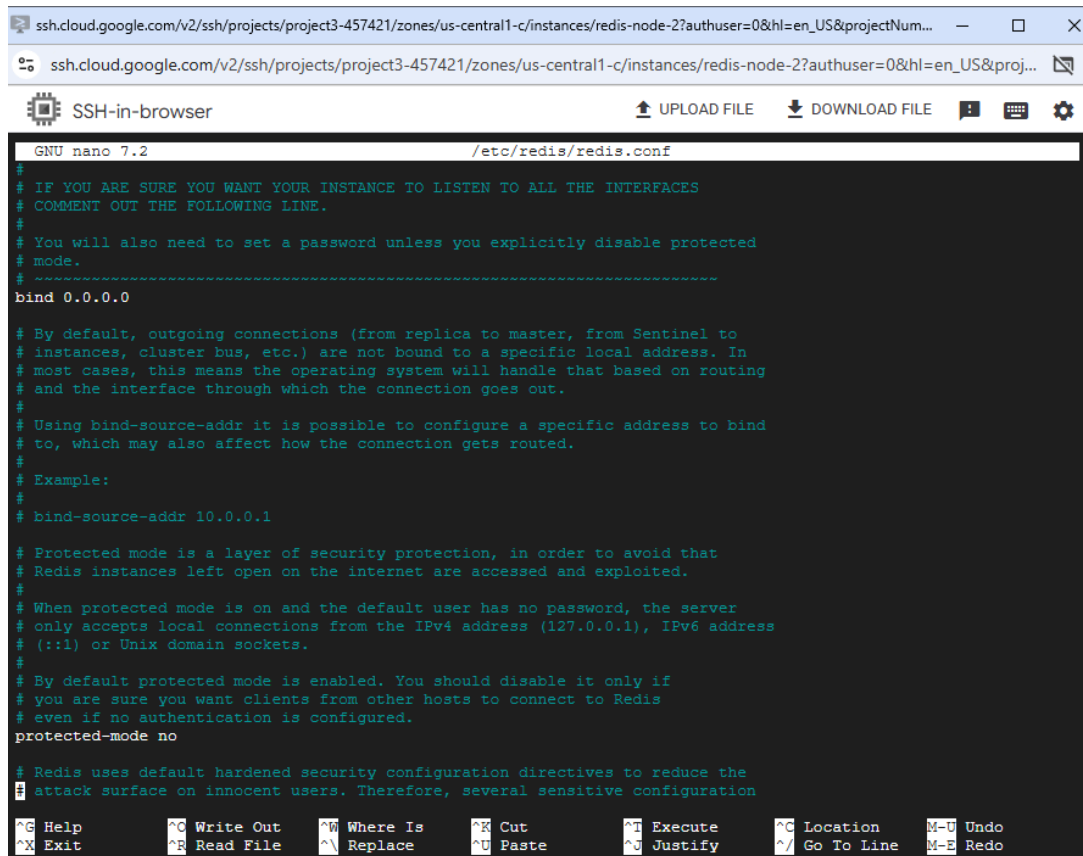
To enter the config file I used the following command:

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

- To edit the following:

- I first edited the bind to 0.0.0.0 and set protected-mode to no. Binding to 0.0.0.0 allows redis to accept connection from any IP address which is perfect for testing in cloud environments and setting protected-mode to no disables Redis built in security features preventing any external connections to the Redis instance. This is great practice for again a testing environment but should not be used in a real world scenario.

- This is show the config filed look after:



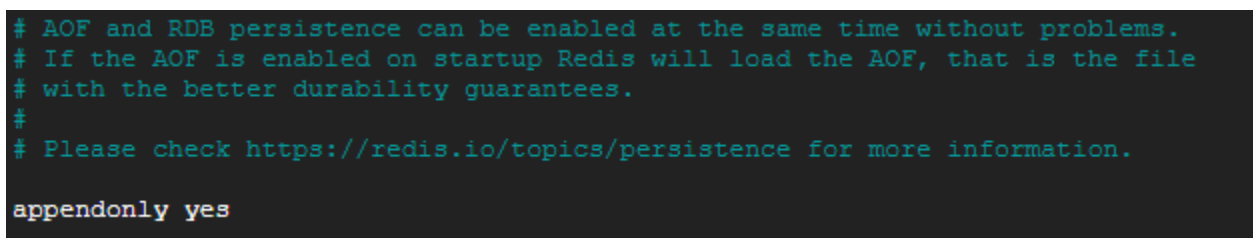
```
GNU nano 7.2 /etc/redis/redis.conf
#
# IF YOU ARE SURE YOU WANT YOUR INSTANCE TO LISTEN TO ALL THE INTERFACES
# COMMENT OUT THE FOLLOWING LINE.
#
# You will also need to set a password unless you explicitly disable protected
# mode.
# ~~~~~
bind 0.0.0.0

# By default, outgoing connections (from replica to master, from Sentinel to
# instances, cluster bus, etc.) are not bound to a specific local address. In
# most cases, this means the operating system will handle that based on routing
# and the interface through which the connection goes out.
#
# Using bind-source-addr it is possible to configure a specific address to bind
# to, which may also affect how the connection gets routed.
#
# Example:
#
# bind-source-addr 10.0.0.1

# Protected mode is a layer of security protection, in order to avoid that
# Redis instances left open on the internet are accessed and exploited.
#
# When protected mode is on and the default user has no password, the server
# only accepts local connections from the IPv4 address (127.0.0.1), IPv6 address
# (::1) or Unix domain sockets.
#
# By default protected mode is enabled. You should disable it only if
# you are sure you want clients from other hosts to connect to Redis
# even if no authentication is configured.
protected-mode no

# Redis uses default hardened security configuration directives to reduce the
# attack surface on innocent users. Therefore, several sensitive configuration
```

- Next I set appendonly to yes which enables Redis to continue data by writing every write operation to the disk in an append only file which ensures the data stays in contact after any restarts.



```
# AOF and RDB persistence can be enabled at the same time without problems.
# If the AOF is enabled on startup Redis will load the AOF, that is the file
# with the better durability guarantees.
#
# Please check https://redis.io/topics/persistence for more information.

appendonly yes
```

- Afterwards I set **cluster-enabled to yes** which enables the Redis Cluster mode. This is crucial for running Redis in a clustered environment.
- Then I set the **cluster config file to nodes.conf** which confirms the file where Redis Cluster stores the state.



- Finally I set the **Cluster Node Timeout to 15000 milliseconds** for Redis to detect if a node is unresponsive.

- This is show the config filed look after:

```
##### REDIS CLUSTER #####
# Normal Redis instances can't be part of a Redis Cluster; only nodes that are
# started as cluster nodes can. In order to start a Redis instance as a
# cluster node enable the cluster support uncommenting the following:
#
cluster-enabled yes

# Every cluster node has a cluster configuration file. This file is not
# intended to be edited by hand. It is created and updated by Redis nodes.
# Every Redis Cluster node requires a different cluster configuration file.
# Make sure that instances running in the same system do not have
# overlapping cluster configuration file names.
#
cluster-config-file nodes.conf

# Cluster node timeout is the amount of milliseconds a node must be unreachable
# for it to be considered in failure state.
# Most other internal time limits are a multiple of the node timeout.
#
cluster-node-timeout 15000
```

- I then saved the file and ran the following command with the purpose of restarting Redis with the changes made:

```
sudo systemctl restart redis
```

- The next step for task 2 is to use `redis-cli --cluster create` to initialize a Redis Cluster. Aiming for at least 3 master nodes and configuring a replication factor of at least 1 to ensure key durability and read availability.

- I first started by running the command:

```
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
```

- The purpose of the command is to use the cluster flag to alert the Redis server that I want to use the Redis cluster commands. The alert will create a new cluster with the specified nodes using their internal IP followed by the port number of 6379. This creates a Redis cluster by assigning hash slots which involves assigning a portion of the keyspace to each node and setting up the internal connections between the nodes which is key for replication.

- Finally ending with the --cluster-replicas 1 which focuses on the number of replicas nodes for each master node in the cluster. Meaning now each master node found in the cluster will have a replica node.

- The replicas will be used for data redundancy meaning if something goes wrong with the master nodes the replica will step up/be promoted to take its place allowing the cluster to stay usable.

- This is an image of the output; showing the hash slot allocation on all 6 nodes and a successful execution of Node configuration stating in green at the bottom of the image how all “all nodes agree about slot confirmation” along with which nodes are masters and replicas.

```
vmop232@redis-node-1:~$ sudo systemctl restart redis
vmop232@redis-node-1:~$ 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
>>> 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: 708e9901616c408e9feb3ef6c087d2360698ed74 10.128.0.2:6379
slots:[0-5460] (5461 slots) master
M: 23d934a7e95b2fea9eca5730b0fea8e429dc88af 10.128.0.3:6379
slots:[5461-10922] (5462 slots) master
M: 1f42079c4c9961b6bceec8e2d9f521f82313eab6 10.128.0.4:6379
slots:[10923-16383] (5461 slots) master
S: 49de2b2b6e054fb868ab23806852680d8cf793fe 10.128.0.5:6379
replicates 1f42079c4c9961b6bceec8e2d9f521f82313eab6
S: f7cff51cfb5367ab68fae8ba32dale69f971257a 10.128.0.6:6379
replicates 708e9901616c408e9feb3ef6c087d2360698ed74
S: 7e3a53581e1ab2ba06e9f307d601fb6ef39f77d1 10.128.0.7:6379
replicates 23d934a7e95b2fea9eca5730b0fea8e429dc88af
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: 708e9901616c408e9feb3ef6c087d2360698ed74 10.128.0.2:6379
slots:[0-5460] (5461 slots) master
1 additional replica(s)
S: f7cff51cfb5367ab68fae8ba32dale69f971257a 10.128.0.6:6379
slots: (0 slots) slave
replicates 708e9901616c408e9feb3ef6c087d2360698ed74
S: 7e3a53581e1ab2ba06e9f307d601fb6ef39f77d1 10.128.0.7:6379
slots: (0 slots) slave
replicates 23d934a7e95b2fea9eca5730b0fea8e429dc88af
M: 1f42079c4c9961b6bceec8e2d9f521f82313eab6 10.128.0.4:6379
slots:[10923-16383] (5461 slots) master
1 additional replica(s)
M: 23d934a7e95b2fea9eca5730b0fea8e429dc88af 10.128.0.3:6379
slots:[5461-10922] (5462 slots) master
1 additional replica(s)
S: 49de2b2b6e054fb868ab23806852680d8cf793fe 10.128.0.5:6379
slots: (0 slots) slave
replicates 1f42079c4c9961b6bceec8e2d9f521f82313eab6
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

- To make sure that the cluster were created I used the following commands:

```
redis-cli -c -p 6379
    set testkey "hello"
    get testkey
```

- c which acts like a flag to use cluster mode in redis-cli with the port being set as the default port where the Redis instance connects to. Then it stores the key-value pair testkey "hello" in the Redis database, where I then retrieve the value for the key testkey. The purpose for this is to ensure that I can interact with Redis in a clustered set up and that my key value operations are working properly by the Redis nodes in the cluster.

- Then I ran the command:

```
cluster nodes
```

- This command outputs a detailed report about the nodes in the Redis cluster, providing the id of each node along with the Ip address/port of each node, the role of each node meaning if its a master or replace , the slots handled by each node, and the status of each node where it can be connected or disconnected. In other words it gives me the cluster configuration report.

- This was the output; We can see which nodes are the master and which as the slaves/replicas.

```
vmop232@redis-node-1:~$ redis-cli -c -p 6379
127.0.0.1:6379> set testkey "hello"
OK
127.0.0.1:6379> get testkey
"hello"
127.0.0.1:6379> cluster nodes
f7cff51cfb5367ab68fae8ba32dale69f971257a 10.128.0.6:6379@16379 slave 708e9901616c408e9feb3ef6c087d2360698ed74 0
1745192455921 1 connected
7e3a53581e1ab2ba06e9f307d601fb6ef39f77d1 10.128.0.7:6379@16379 slave 23d934a7e95b2fea9eca5730b0fea8e429dc88af 0
1745192456000 2 connected
1f42079c4c9961b6bceec8e2d9f521f82313eab6 10.128.0.4:6379@16379 master - 0 1745192453000 3 connected 10923-16383
708e9901616c408e9feb3ef6c087d2360698ed74 10.128.0.2:6379@16379 myself,master - 0 1745192452000 1 connected 0-546
0
23d934a7e95b2fea9eca5730b0fea8e429dc88af 10.128.0.3:6379@16379 master - 0 1745192456926 2 connected 5461-10922
49de2b2b6e054fb868ab23806852680d8cf793fe 10.128.0.5:6379@16379 slave 1f42079c4c9961b6bceec8e2d9f521f82313eab6 0
1745192453000 3 connected
127.0.0.1:6379> █
```

- Next I ran the command:

```
cluster info
```

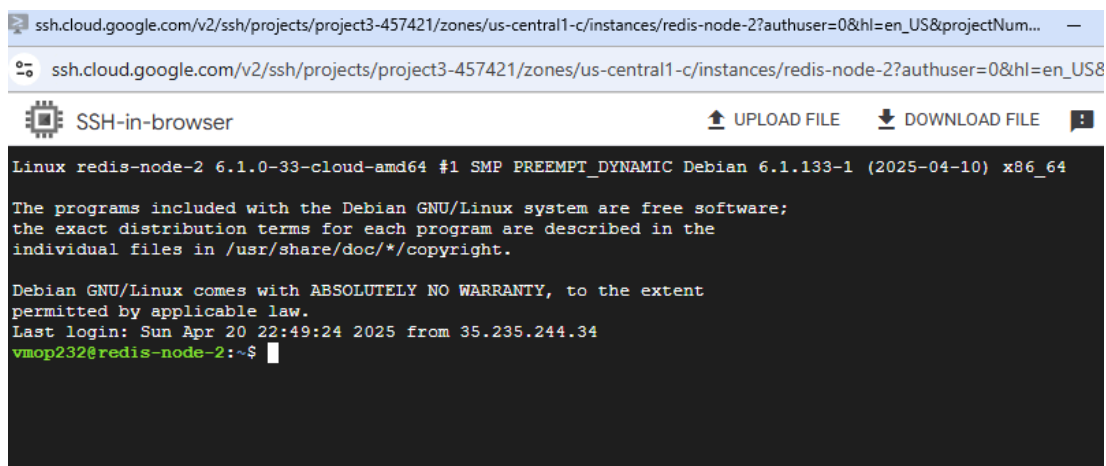
- The purpose of this command is to provide health status about the Redis Cluster, including whether it is functioning properly and how many nodes and slots are assigned.

- This was the output; showing a total of 6 known nodes with no slot errors.

```
127.0.0.1:6379> 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:560
cluster_stats_messages_pong_sent:540
cluster_stats_messages_sent:1100
cluster_stats_messages_ping_received:535
cluster_stats_messages_pong_received:560
cluster_stats_messages_meet_received:5
cluster_stats_messages_received:1100
total_cluster_links_buffer_limit_exceeded:0
127.0.0.1:6379> █
```

### **Task 3: Failover Configuration and Testing :**

- To begin task 3 I first SSH into another node:

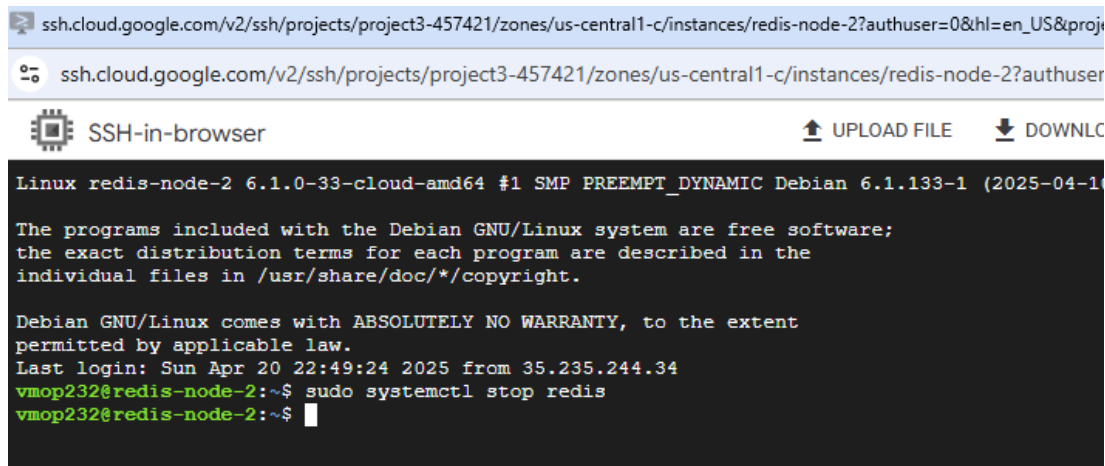


```
ssh.cloud.google.com/v2/ssh/projects/project3-457421/zones/us-central1-c/instances/redis-node-2?authuser=0&hl=en_US&projectNum... —
ssh.cloud.google.com/v2/ssh/projects/project3-457421/zones/us-central1-c/instances/redis-node-2?authuser=0&hl=en_US8
SSH-in-browser  UPLOAD FILE  DOWNLOAD FILE  !
Linux redis-node-2 6.1.0-33-cloud-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.133-1 (2025-04-10) x86_64
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Apr 20 22:49:24 2025 from 35.235.244.34
vmop232@redis-node-2:~$ █
```

- Next I ran the following command to stop Redis on the current node which is redis-node-2

```
sudo systemctl stop redis
```

- This is how it looked after running the command. It seems like nothing happened but the importance of this step to ensure that the replica was promoted to master. Simulating a fail over.



The screenshot shows a terminal window titled 'SSH-in-browser' with a URL bar at the top. The terminal output shows the user 'vmop232' logging into 'redis-node-2'. After the login banner, the user enters the command 'sudo systemctl stop redis'. The prompt returns to the user, indicating the command was executed successfully.

```
Linux redis-node-2 6.1.0-33-cloud-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.133-1 (2025-04-11)

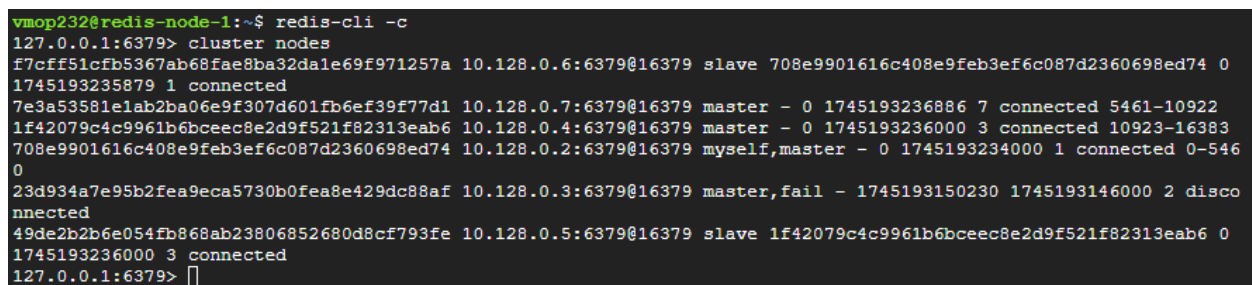
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Apr 20 22:49:24 2025 from 35.235.244.34
vmop232@redis-node-2:~$ sudo systemctl stop redis
vmop232@redis-node-2:~$
```

- To really see the failover test I SSH back into another node and ran the following command of cluster nodes to see the status of the nodes after stopping Redis in one of the master nodes.

```
redis-cli -c
cluster nodes
```

- This was the output; let's explain the difference between this image of cluster nodes and the previous image of cluster nodes.



The screenshot shows a terminal window where the user 'vmop232' is on 'redis-node-1'. They run the command 'redis-cli -c cluster nodes'. The output lists the status of all nodes in the cluster. Node 1 (127.0.0.1) is a master. Node 2 (10.128.0.6) is a slave. Node 3 (10.128.0.7) is a master. Node 4 (10.128.0.4) is a master. Node 5 (10.128.0.2) is a master. Node 6 (10.128.0.3) is a master in a 'fail' state. Node 7 (10.128.0.5) is a slave.

```
vmop232@redis-node-1:~$ redis-cli -c
127.0.0.1:6379> cluster nodes
f7cfff51c5367ab68fae8ba32dale69f971257a 10.128.0.6:6379@16379 slave 708e9901616c408e9feb3ef6c087d2360698ed74 0
1745193235879 1 connected
7e3a53581e1ab2ba06e9f307d601fb6ef39f77d1 10.128.0.7:6379@16379 master - 0 1745193236886 7 connected 5461-10922
1f42079c4c9961b6bceec8e2d9f521f82313eab6 10.128.0.4:6379@16379 master - 0 1745193236000 3 connected 10923-16383
708e9901616c408e9feb3ef6c087d2360698ed74 10.128.0.2:6379@16379 myself,master - 0 1745193234000 1 connected 0-546
0
23d934a7e95b2fea9eca5730b0fea8e429dc88af 10.128.0.3:6379@16379 master,fail - 1745193150230 1745193146000 2 disco
nnected
49de2b2b6e054fb868ab23806852680d8cf793fe 10.128.0.5:6379@16379 slave 1f42079c4c9961b6bceec8e2d9f521f82313eab6 0
1745193236000 3 connected
127.0.0.1:6379>
```

• Lets compare the previous cluster node from the before the failover test to after:

### • Before:

Now keep in mind the Internal IP and name of the node where Redis was stopped was called redis-node-2 with an IP of **10.128.0.3**; In the before failover test image this node was a master for slots 5461-10922. Keep an eye at the IP **10.128.0.7:6379** as well which is a slave/replica node as shown in the image below:

```
vmop232@redis-node-1:~$ redis-cli -c -p 6379
127.0.0.1:6379> set testkey "hello"
OK
127.0.0.1:6379> get testkey
"hello"
127.0.0.1:6379> cluster nodes
f7cfff51cfb5367ab68fae8ba32dale69f971257a 10.128.0.6:6379@16379 slave 708e9901616c408e9feb3ef6c087d2360698ed74 0
1745192455921 1 connected
7e3a53581e1ab2ba06e9f307d601fb6ef39f77d1 10.128.0.7:6379@16379 slave 23d934a7e95b2fea9eca5730b0fea8e429dc88af 0
1745192456000 2 connected
1f42079c4c9961b6bceec8e2d9f521f82313eab6 10.128.0.4:6379@16379 master - 0 1745192453000 3 connected 10923-16383
708e9901616c408e9feb3ef6c087d2360698ed74 10.128.0.2:6379@16379 myself,master - 0 1745192452000 1 connected 0-5460
0
23d934a7e95b2fea9eca5730b0fea8e429dc88af 10.128.0.3:6379@16379 master - 0 1745192456926 2 connected 5461-10922
49de2b2b6e054fb868ab23806852680d8cf793fe 10.128.0.5:6379@16379 slave 1f42079c4c9961b6bceec8e2d9f521f82313eab6 0
1745192453000 3 connected
127.0.0.1:6379>
```

### • After:

Now in the afterimage of the failover test you can see the Redis cluster detected the failure and promoted its replica to master where **10.128.0.7:6379** is now a master, owning slot range of 5461-10922; and **10.128.0.3** is now showing up as **master, fail** as shown in the image below:

```
Linux redis-node-1 6.1.0-33-cloud-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.133-1 (2025-04-10) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Apr 20 22:30:59 2025 from 35.235.244.34
vmop232@redis-node-1:~$ redis-cli -c
127.0.0.1:6379> cluster nodes
f7cfff51cfb5367ab68fae8ba32dale69f971257a 10.128.0.6:6379@16379 slave 708e9901616c408e9feb3ef6c087d2360698ed74 0 1745193235879
1 connected
7e3a53581e1ab2ba06e9f307d601fb6ef39f77d1 10.128.0.7:6379@16379 master - 0 1745193236886 7 connected 5461-10922
1f42079c4c9961b6bceec8e2d9f521f82313eab6 10.128.0.4:6379@16379 master - 0 1745193236000 3 connected 10923-16383
708e9901616c408e9feb3ef6c087d2360698ed74 10.128.0.2:6379@16379 myself,master - 0 1745193234000 1 connected 0-5460
23d934a7e95b2fea9eca5730b0fea8e429dc88af 10.128.0.3:6379@16379 master,fail - 1745193150230 1745193146000 2 disconnected
49de2b2b6e054fb868ab23806852680d8cf793fe 10.128.0.5:6379@16379 slave 1f42079c4c9961b6bceec8e2d9f521f82313eab6 0 1745193236000
3 connected
127.0.0.1:6379>
```

- To check Check that the cluster remains operational for clients I can the following command:

```
set postfailover "still works"
get postfailover
```

- This command is similar to one previously used where set postfailover "still works" will store the value "still works" with the key postfailover. And get postfailover gets the value for the key postfailover, which should return "still works" to confirm that the Redis cluster properly handled the failover.

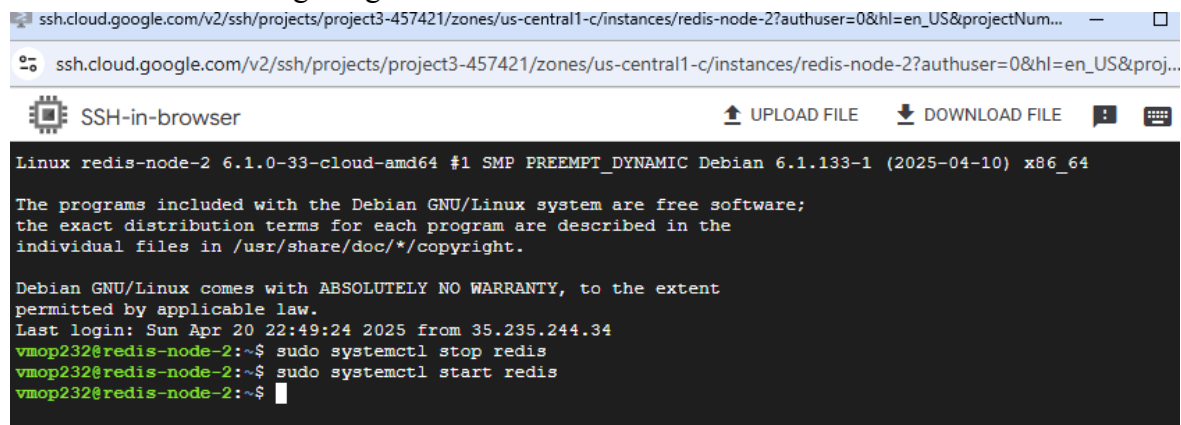
- After the running command get postfailover "still works" appears confirming the failover was handled. As shown in the image:

```
vmop232@redis-node-1:~$ redis-cli -c
127.0.0.1:6379> cluster nodes
f7cff51cfb5367ab68fae8ba32dale69f971257a 10.128.0.6:6379@16379 slave 708e9901616c408e9feb3ef6c087d2360698ed74 0 1745193235879
1 connected
7e3a53581e1ab2ba06e9f307d601fb6ef39f77d1 10.128.0.7:6379@16379 master - 0 1745193236886 7 connected 5461-10922
1f42079c4c9961b6bceec8e2d9f521f82313eab6 10.128.0.4:6379@16379 master - 0 1745193236000 3 connected 10923-16383
708e9901616c408e9feb3ef6c087d2360698ed74 10.128.0.2:6379@16379 myself,master - 0 1745193234000 1 connected 0-5460
23d934a7e95b2fea9eca5730b0fea8e429dc88af 10.128.0.3:6379@16379 master,fail - 1745193150230 1745193146000 2 disconnected
49de2b2b6e054fb868ab23806852680d8cf793fe 10.128.0.5:6379@16379 slave 1f42079c4c9961b6bceec8e2d9f521f82313eab6 0 1745193236000
3 connected
127.0.0.1:6379> set postfailover "still works"
-> Redirected to slot [14029] located at 10.128.0.4:6379
OK
10.128.0.4:6379> get postfailover
"still works"
10.128.0.4:6379> █
```

- The final step of task 3 is to restart the stopped node and verify it rejoins the cluster as a replica/slave. To restart the stop node I ran the command:

```
sudo systemctl start redis
```

- As shown in following image:



```
ssh.cloud.google.com/v2/ssh/projects/project3-457421/zones/us-central1-c/instances/redis-node-2?authuser=0&hl=en_US&projectNum...
ssh.cloud.google.com/v2/ssh/projects/project3-457421/zones/us-central1-c/instances/redis-node-2?authuser=0&hl=en_US&proj...
SSH-in-browser
Linux redis-node-2 6.1.0-33-cloud-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.133-1 (2025-04-10) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Apr 20 22:49:24 2025 from 35.235.244.34
vmop232@redis-node-2:~$ sudo systemctl stop redis
vmop232@redis-node-2:~$ sudo systemctl start redis
vmop232@redis-node-2:~$ █
```



- I then ran the command:`cluster nodes` in another node to confirm that the rejoin was a success. After running the command you can see that the IP of the stopped Redis master node which was **10.128.0.3** has rejoined and is now a slave with the previous replica node now staying as the master.

```
10.128.0.4:6379> cluster nodes
1f42079c4c9961b6bceec8e2d9f521f82313eab6 10.128.0.4:6379@16379 myself,master - 0 1745193853000 3 connected 10923-16383
708e9901616c408e9feb3ef6c087d2360698ed74 10.128.0.2:6379@16379 master - 0 1745193857173 1 connected 0-5460
f7cff51cfb5367ab68fae8ba32da1e69f971257a 10.128.0.6:6379@16379 slave 708e9901616c408e9feb3ef6c087d2360698ed74 0 1745193856000
1 connected
49de2b2b6e054fb868ab23806852680d8cf793fe 10.128.0.5:6379@16379 slave 1f42079c4c9961b6bceec8e2d9f521f82313eab6 0 1745193855000
3 connected
7e3a53581e1ab2ba06e9f307d601fb6ef39f77d1 10.128.0.7:6379@16379 master - 0 1745193855162 7 connected 5461-10922
23d934a7e95b2fea9eca5730b0fea8e429dc88af 10.128.0.3:6379@16379 slave 7e3a53581e1ab2ba06e9f307d601fb6ef39f77d1 0 1745193856168
7 connected
10.128.0.4:6379> █
```

#### Task 4: Key-Value Workload Generator Implementation:

- For task 4 I needed to develop a simple workload generator program focusing on simulating key-value access patterns. Ensuring that the generator can connect to the Redis cluster. Implement logic primarily performing high-frequency, basic key operations SET, GET, targeting random keys to distribute the load across shards. While also incorporating concurrency to simulate multiple clients accessing keys simultaneously. Finally adding basic functionality to measure latency per key operation and calculate overall throughput mainly aiming for key operations per second.

- I first ran the following commands:

```
sudo apt install python3-venv -y
python3 -m venv redis-env
source redis-env/bin/activate
pip install redis-py-cluster
```

- This command Installs venv if missing and the creates a virtual environment. Then it activate the virtual environment, finally installing redis-py-cluster. The purpose for this commands is to set up a Python virtual environment within an isolated environment.

- To confirm installation I ran the commands:

```
python3 -m venv redis-env
source redis-env/bin/activate
pip install redis-py-cluster
```



- This was the output:

```
vmop232@redis-node-1:~$ python3 -m venv redis-env
vmop232@redis-node-1:~$ source redis-env/bin/activate
(redis-env) vmop232@redis-node-1:~$ pip install redis-py-cluster
Collecting redis-py-cluster
  Downloading redis_py_cluster-2.1.3-py2.py3-none-any.whl (42 kB)
----- 42.6/42.6 kB 1.9 MB/s eta 0:00:00
Collecting redis<4.0.0,>=3.0.0
  Downloading redis-3.5.3-py2.py3-none-any.whl (72 kB)
----- 72.1/72.1 kB 6.2 MB/s eta 0:00:00
Installing collected packages: redis, redis-py-cluster
Successfully installed redis-3.5.3 redis-py-cluster-2.1.3
(redis-env) vmop232@redis-node-1:~$ pip list
Package            Version
-----
pip                23.0.1
redis              3.5.3
redis-py-cluster   2.1.3
setuptools         66.1.1
(redis-env) vmop232@redis-node-1:~$
```

- I then ran the command to create the workload generator script.

```
nano workload_generator.py
```

- The code for workload generator can be found here:

```
https://github.com/lu8e/8.git
```

- To begin the work generator program I imported threading to allow me to work with threads to simulate multiple clients accessing Redis at the same time. I then use the Redis Python client for Redis cluster so I can interact with a Redis cluster that spans multiple nodes.

- The startup nodes contains a list of dictionaries where each dictionary contains the IP address and port of a Redis node in the cluster. This represents the starting points when it comes to connecting to the Redis Cluster.

```
1  import threading
2  import time
3  import random
4  import string
5  from rediscluster import RedisCluster
6
7  # Redis Cluster connection info
8  startup_nodes = [
9      {"host": "10.128.0.2", "port": "6379"},
10     {"host": "10.128.0.3", "port": "6379"},
11     {"host": "10.128.0.4", "port": "6379"},
12     {"host": "10.128.0.5", "port": "6379"},
13     {"host": "10.128.0.6", "port": "6379"},
14     {"host": "10.128.0.7", "port": "6379"},
15 ]
16
```

- I then had to connect to the Redis cluster using RedisCluster specifying the Redis node to connect to list of IP address and port of a Redis node in the cluster, and finally making sure that the responses from Redis are automatically decoded from bytes to strings.

```
# Connect to Redis Cluster
rc = RedisCluster(startup_nodes=startup_nodes, decode_responses=True)
```

- Next is the global variables where I keep track of the total number of operations performed from both set and get. While also keeping track of the total latency for the Redis operations. Also using threading lock to synchronize access to shared data between operation count and the total latency which is crucial to prevent threads from changing variables at the same time.

```
20 # Global counters
21 operation_count = 0
22 latency_total = 0
23 lock = threading.Lock()
```

- I then made a function to generate random strings to simulate the Redis cluster to make sure that the workload is distributed across the entire Redis cluster creating more realistic performance and stress test by testing how Redis performs under high demand with random data.

```
25
26 def random_string(length=8):
27     return ''.join(random.choices(string.ascii_letters + string.digits, k=length))
28
```

- The worker function then simulates a Redis client by constantly performing SET and GET operations on a Redis Cluster. It also generates random keys and values, with the purpose of measuring the time it takes to execute both operations given the computed latency in milliseconds, as well as updating the global counters to find both the total number of operations and latency. The function runs in an infinite loop to really see the test failure in real time and to prevent any concurrency problems the global counters are updated using a threading lock to avoid the shared data being manipulated by multiple threads at the same time.

```
30 def worker():
31     global operation_count, latency_total
32     while True:
33         key = random_string(8)
34         value = random_string(16)
35
36         start_time = time.time()
37         rc.set(key, value)
38         rc.get(key)
39         latency = (time.time() - start_time) * 1000 # milliseconds
40
41         with lock:
42             operation_count += 2 # one SET + one GET
43             latency_total += latency
44
```

- The monitor function displays reports of the performance of the Redis Cluster by finding and printing the number of operations per second and the average latency. Every 5 seconds, it will get the total number of operations and the total latency, to find the average latency, and then reset the counters. The output report, shows the operations per second and the average latency in milliseconds.

```

46  def monitor():
47      global operation_count, latency_total
48      while True:
49          time.sleep(5) # Report every 5 seconds
50          with lock:
51              ops = operation_count
52              avg_latency = latency_total / operation_count if operation_count else 0
53              operation_count = 0
54              latency_total = 0
55              print(f"[5s Report] Ops/sec: {ops / 5:.2f}, Avg Latency: {avg_latency:.2f} ms")
56

```

### Task 5: Performance Evaluation (Key-Value Focus):

- For task 5 I ran the following command to execute the workload generator program.

```
python3 workload_generator.py
```

- After executing the program; I am provided with both a **throughput of 7300 and up** operations per second followed by an **average latency, of 1.34-1.37**.

```

(redis-env) vmop232@redis-node-1:~$ nano workload_generator.py
(redis-env) vmop232@redis-node-1:~$ python3 workload_generator.py
[5s Report] Ops/sec: 7385.60, Avg Latency: 1.35 ms
[5s Report] Ops/sec: 7362.00, Avg Latency: 1.35 ms
[5s Report] Ops/sec: 7368.00, Avg Latency: 1.35 ms
[5s Report] Ops/sec: 7381.60, Avg Latency: 1.34 ms
[5s Report] Ops/sec: 7353.60, Avg Latency: 1.35 ms
[5s Report] Ops/sec: 7261.60, Avg Latency: 1.37 ms
[5s Report] Ops/sec: 7317.60, Avg Latency: 1.36 ms
[5s Report] Ops/sec: 7306.80, Avg Latency: 1.36 ms
[5s Report] Ops/sec: 7355.60, Avg Latency: 1.35 ms
[5s Report] Ops/sec: 7432.00, Avg Latency: 1.34 ms
[5s Report] Ops/sec: 7376.40, Avg Latency: 1.35 ms
[5s Report] Ops/sec: 7247.20, Avg Latency: 1.37 ms
[5s Report] Ops/sec: 7241.20, Avg Latency: 1.37 ms
[5s Report] Ops/sec: 7328.40, Avg Latency: 1.35 ms
[5s Report] Ops/sec: 7327.20, Avg Latency: 1.35 ms
[5s Report] Ops/sec: 7483.60, Avg Latency: 1.33 ms
[5s Report] Ops/sec: 7574.00, Avg Latency: 1.31 ms
[5s Report] Ops/sec: 7358.80, Avg Latency: 1.35 ms
[5s Report] Ops/sec: 7332.80, Avg Latency: 1.35 ms
[5s Report] Ops/sec: 7428.40, Avg Latency: 1.34 ms

```

- The next step for task 5 is to run the workload generator program again, and during the run, trigger another failover event by stopping a master node.

- To perform this I SSH into another master node and run the previously used command to stop the Redis.

```
sudo systemctl stop redis
```

- But before stopping the redis I ran the command:

```
ps aux | grep redis
```

- This command will list the process that are currently running that deal with redis to make sure that redis is running;

- The following is my output, where I see the execution of the workload generator again with the throughput and latency being displayed properly before stopping the Redis in master node redis-node-3. Then after running the command to stop redis in redis-node-3, in real time I was able to see when the program was sending SET operations to the cluster when a node goes unresponsive causing an output of “rediscluster.exceptions.ClusterError: TTL exhausted” meaning the client exhausted the amount of times to find a working node and timed out.

```
(redis-env) vmop232@redis-node-1:~$ python3 workload_generator.py
[5s Report] Ops/sec: 7356.80, Avg Latency: 1.35 ms
[5s Report] Ops/sec: 7384.00, Avg Latency: 1.34 ms
[5s Report] Ops/sec: 7387.20, Avg Latency: 1.34 ms
[5s Report] Ops/sec: 7367.20, Avg Latency: 1.35 ms
[5s Report] Ops/sec: 7265.60, Avg Latency: 1.37 ms
[5s Report] Ops/sec: 7248.00, Avg Latency: 1.37 ms
[5s Report] Ops/sec: 7395.20, Avg Latency: 1.34 ms
[5s Report] Ops/sec: 7475.20, Avg Latency: 1.33 ms
[5s Report] Ops/sec: 7315.20, Avg Latency: 1.36 ms
[5s Report] Ops/sec: 7316.40, Avg Latency: 1.36 ms
[5s Report] Ops/sec: 4060.00, Avg Latency: 1.37 ms
Exception in thread Thread-6 (worker):
Traceback (most recent call last):
  File "/usr/lib/python3.11/threading.py", line 1038, in _bootstrap_inner
    self.run()
  File "/usr/lib/python3.11/threading.py", line 975, in run
    self._target(*self._args, **self._kwargs)
  File "/home/vmop232/workload_generator.py", line 35, in worker
    rc.set(key, value)
  File "/home/vmop232/redis-env/lib/python3.11/site-packages/redis/client.py", line 1801, in set
    return self.execute_command('SET', *pieces)
           ~~~~~^~~~~~
  File "/home/vmop232/redis-env/lib/python3.11/site-packages/rediscluster/client.py", line 555, in execute_command
    return self.execute_command(*args, **kwargs)
           ~~~~~^~~~~~
  File "/home/vmop232/redis-env/lib/python3.11/site-packages/rediscluster/client.py", line 721, in _execute_command
    raise ClusterError('TTL exhausted.')
rediscluster.exceptions.ClusterError: TTL exhausted.
Exception in thread Thread-1 (worker):
Traceback (most recent call last):
  File "/usr/lib/python3.11/threading.py", line 1038, in _bootstrap_inner
    self.run()
  File "/usr/lib/python3.11/threading.py", line 975, in run
    self._target(*self._args, **self._kwargs)

vmop232@redis-node-3:~$ ps aux | grep redis
redis        6760  0.6  1.2 110984 49324 ?        Ssl  Apr20   0:50 /usr/bin/redis-server 0.0.0.0:6379 [cluster]
vmop232     15860  0.0  0.0   3748  1944 pts/1    S+   01:08   0:00 grep redis
vmop232@redis-node-3:~$ sudo systemctl stop redis
vmop232@redis-node-3:~$
```

- I was then presented with an output of failed threads because of an exception and then shown how the cluster restabilized which took no more than 7 seconds with an operation/second of 0 due to the end of all workers. In my work load generator program there was no plan b to handle the failover; its job was to simply return with real time reports of my redis client and its operations.

[illegible]

## Task 6: Reporting and Finalization:

**1)System Architecture:** The system architecture being tested is a Redis Cluster deployment configured as a distributed key value store. In this environment clients connect with a focus on multiple Redis nodes having a key/slot distribution while also handling replicas.

- The workload was driven by a custom python program called workload generator, which uses multithreading to simulate clients performing SET operations.

**2)Workload Details:**The access pattern follows a workload of SET commands meaning they are constantly being made, then keys/values that were randomly generated and finally multiple threads to simulate real world scenarios.

**3) Results: Performance Metrics:** The Initial throughput before failover test was around 4060 ops/sec with an average latency of 1.37 ms per operation. After the failover throughput it went to 0 ops/sec after error conditions appeared.

### **4) Failover and Error Analysis:**

Observed Error was

```
rediscluster.exceptions.ClusterError: TTL exhausted
```

- The error was caused by the cluster exhausting the amount of times to find a working node and timed out.

### **5)Failover Behavior:**

The cluster did not recover from node issues during the workload run causing all threads to crash after the errors, resulting in 0 throughput as mentioned in task 5.

### **Source Code/Config Files/Final report:**

<https://github.com/lu8e/8.git>