Cloud and Big Data Project - USTH

Do Thanh Dat 2440059

November 22, 2025

I. Introduction

This project aims to automate the deployment of an Apache Spark cluster on Google Cloud Platform (GCP) using Terraform for infrastructure provisioning and Ansible for configuration management. The target environment is a small but realistic big data stack composed of a Spark Standalone cluster on top of HDFS, running a distributed `WordCount` application as a validation workload. While the infrastructure and software stack were successfully configured, the final stage, distributed job execution, was blocked by a networking limitation related to GCP firewall rules and Spark's dynamic RPC ports.

II. Architecture Overview

A. Deployment Diagram

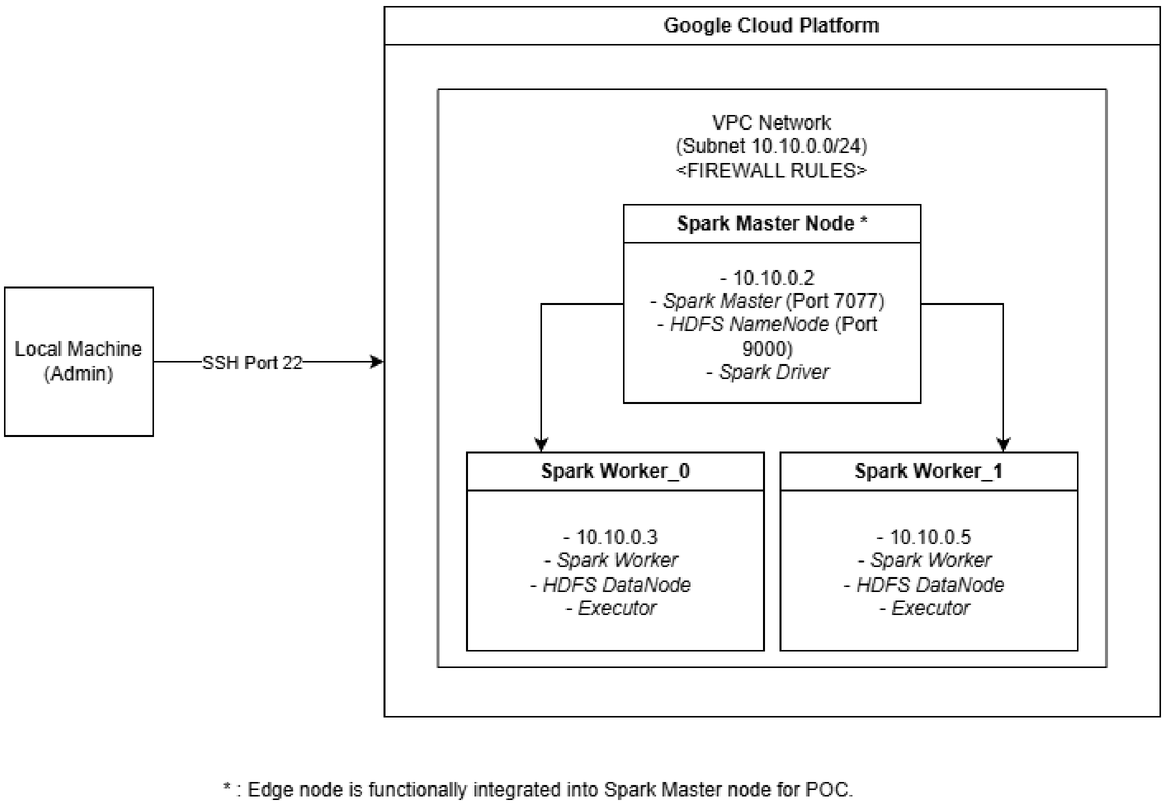

Figure 1: High-level architecture of the GCP-based Spark cluster.

B. Components and Roles

Table 1 describes the main components of the cluster.

Node	IP	Roles
`spark-master`	10.10.0.2	HDFS NameNode + DataNode, Spark Master, Driver / Edge
`spark-worker-0`	10.10.0.3	HDFS DataNode, Spark Worker
`spark-worker-1`	10.10.0.5	HDFS DataNode, Spark Worker

Table 1: Cluster components and responsibilities.

Note: In this project, the *edge node* described in the specification is functionally integrated into `spark-master` in order to simplify the proof-of-concept. In future work, this edge role can easily be moved into a dedicated VM with the same automation pipeline.

III. Methodology

A. Terraform

Terraform is used to interact with the Google Cloud API.

- A custom VPC and subnet with CIDR `10.10.0.0/24`.
- Three instances with network tags (e.g. `spark-node`) for firewall rules.
- Basic firewall rules for SSH (`tcp/22`), HDFS NameNode (`tcp/9000`) and Spark Master web UI (`tcp/8080`).

Running `terraform apply` reliably in all nodes. Destroying and recreating the cluster is a single command, which is important for experimentation and clean testing.

B. Ansible

An Ansible playbook configures each node:

- Installs Java 17 and core utilities.
- Downloads and unpacks Hadoop and Spark into `/opt/hadoop-3.4.2` and `/opt/spark`.
- Adds environment variables (`JAVA_HOME`, `HADOOP_HOME`, `SPARK_HOME`) to `/etc/profile.d`.
- Templates cluster-specific configuration:
  - `core-site.xml`: `fs.defaultFS = hdfs://spark-master:9000`.
  - `hdfs-site.xml`: data directories on each node.
  - `spark-env.sh`: `SPARK_MASTER_HOST` and memory settings.
  - `workers`: lists `spark-worker-0` and `spark-worker-1`.
- Formats the HDFS NameNode and starts the HDFS daemons.
- Starts the Spark Master on `spark-master` and Spark Workers on both worker nodes.

C. WordCount

To validate the cluster, a Java `WordCount` application was packaged into a JAR file spark-wordcount-1.0-SNAPSHOT.jar. The validation workflow on `spark-master` is:

1. Copy a small text file to HDFS:

```
hdfs dfs -mkdir -p /input
hdfs dfs -put filesample.txt /input/filesample.txt
```

2. Submit the Spark job:

```
/opt/spark/bin/spark-submit \
  --class org.example.WordCount \
  --master spark://spark-master:7077 \
  spark-wordcount-1.0-SNAPSHOT.jar \
  hdfs://spark-master:9000/input/filesample.txt \
  hdfs://spark-master:9000/output-wc-1
```

3. Inspect `/output-wc-1/part-*` via HDFS after successful completion.

At this stage, the Spark Master web UI confirmed that executors were being allocated on both workers and that the application was registered. However, the job did not progress beyond the initial stage and never produced output files.

In the next chapter, we will dive into debugging.

IV. Error Analysis

A. Observed Behavior

In terminal, the Spark log repeatedly printed:

- `WARN TaskSchedulerImpl: Initial job has not accepted any resources; check your cluster UI to ensure that workers are registered and have sufficient resources.`
- Application remained in state `RUNNING` on the Master UI but without any completed tasks.

On each worker node, inspecting the executor logs under `$HOME/spark-work/app-...` revealed a consistent pattern. After starting, each executor printed:

- After 120 seconds:

```
WARN NettyRpcEnv: Ignored failure: java.io.IOException: Connecting to spark-master
    /10.10.0.2:40443 timed out (120000 ms)
```

The executor process then terminated with an exception, and the Master marked the executor as failed.

B. Potential Causes

The driver URL uses port `40443`, which is *not* the standard Spark Master port `7077`. In a Spark Standalone cluster, the Master process exposes two important endpoints:

- The cluster manager endpoint on `7077`.
- A dynamically allocated RPC port used by the driver to communicate with executors (here `40443`).

Inside a secure cloud environment, the VPC firewall rules often allow only a small set of TCP ports. In this project, only a limited set (e.g. `22`, `8080`, `9000` and `7077`) were explicitly opened. As a result, worker nodes could initiate a connection to the master on `7077` to register as workers, but executors could not complete the handshake on the driver's dynamically chosen RPC port.

C. Validation

Several facts confirm this hypothesis:

- The Spark Master UI showed workers as `ALIVE` and executors being launched, so basic node-to-node connectivity to port `7077` was functioning.
- The Java processes for executors were successfully created on the workers and ran long enough to attempt RPC communication.
- Every executor log failed with the *same* timeout to the same driver port (`40443`), strongly suggesting a network-level block rather than a misconfiguration of Spark or HDFS.
- No errors related to missing JARs, classpath issues, or HDFS access appeared in the logs.

D. Technical Conclusion

The software stack (Hadoop, Spark, Java and the WordCount application) is correctly installed and configured, and the automation pipeline reliably recreates this environment. The blocking issue is located at the networking layer: GCP firewall rules do not allow internal traffic to the driver's dynamically allocated RPC port, causing executors to time out and preventing the job from running to completion.

V. Future Work and Proposed Fixes

Although the networking issue was not resolved, the solution is conceptually straightforward and can be automated with Terraform.

A. Firewall Rule for Internal VPC Traffic

A potential fix is to explicitly allow internal TCP traffic within the VPC range used by the cluster, e.g.:

```
resource "google_compute_firewall" "allow-internal-spark" {
  name = "allow-internal-spark"
  network = google_compute_network.vpc_network.name

  allow {
    protocol = "tcp"
    ports = ["0-65535"]
  }

  source_ranges = ["10.10.0.0/24"]
  target_tags = ["spark-node"]
}
```

This rule keeps external access locked down while allowing Spark to use any port internally between master, driver and executors.

Alternatively, Spark can be configured with fixed ports (e.g. `spark.driver.port`, `spark.blockManager.port`), and only those ports can be opened, which is more restrictive but slightly more complex to maintain.

B. Dedicated Edge Node

In a production environment, it is advisable to separate the edge node from the master:

- **Edge node**: exposed to the internet via SSH or HTTP for job submission, without running master or worker roles.
- **Cluster nodes**: only accessible from within the VPC.

This current automation already supports role-based configuration via Ansible inventories and can be easily extended to add an edge node instance and associated firewall rules.

C. Summary

This project successfully demonstrates automated provisioning and configuration of a Spark + HDFS cluster on GCP using Terraform and Ansible. Although the WordCount job failed to complete due to an identified firewall restriction on Spark's dynamic RPC ports, I was able to point out potential causes and proposed the fix. With a small update to the Terraform firewall configuration, the same pipeline can be re-applied to obtain a fully operational distributed big data environment.