Automated Apache Spark Cluster Deployment on GCP

Using Terraform, Ansible, and Service Discovery for Cloud & BigData Course 2025

Le Minh Hoang - 2440051

November 22, 2025

1 Introduction

The project addresses the challenge of deploying and managing distributed big data infrastructure by automating the entire lifecycle of an Apache Spark cluster

1.1 Architecture Overview

The cluster architecture consists of three primary components:

- **Runner Node (Edge Node):** Hosts Code-Server for web-based development, service discovery API, and Ansible orchestration
- **Master Node:** Runs Spark master services, web UI, and history server
- **Worker Nodes:** Execute Spark jobs with automatic registration and scaling

Figure 1 illustrates the complete infrastructure topology deployed on GCP.

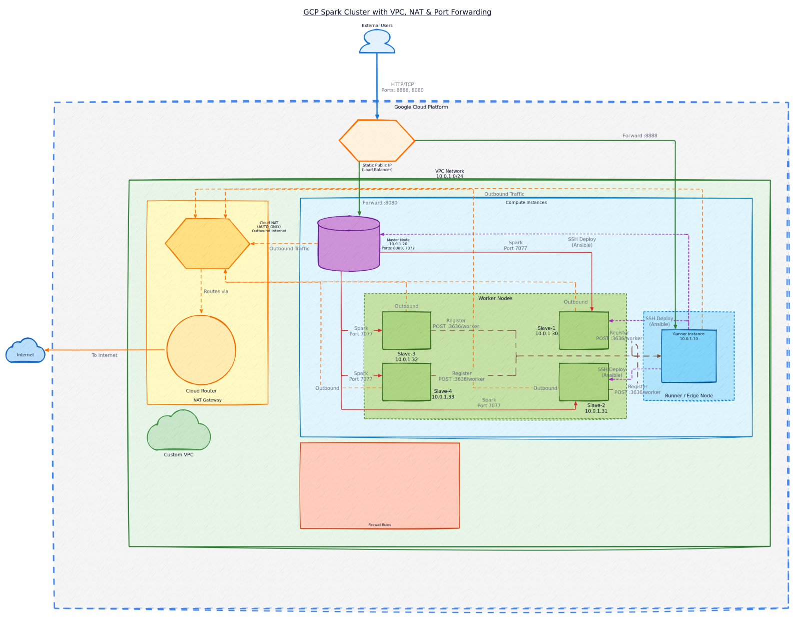

Figure 1: Apache Spark Cluster Architecture on GCP with VPC, NAT, and Port Forwarding

2 Infrastructure Implementation with Terraform

2.1 VPC and Network Configuration

The infrastructure is deployed within a custom Virtual Private Cloud (VPC) to ensure network isolation and security. The implementation uses a dedicated subnet with CIDR block `10.0.1.0/24`.

2.2 NAT Gateway for Outbound Connectivity

Since the cluster nodes do not have public IP addresses for security reasons, a Cloud NAT gateway provides outbound internet access for package installations and external communications. The Cloud Router manages the NAT configuration dynamically.

2.3 Firewall Rules and Security

The firewall configuration implements a defense-in-depth approach with the following rules:

- **Internal Communication:** All TCP/UDP traffic allowed within `10.0.1.0/24`
- **SSH Access:** Restricted to IAP (Identity-Aware Proxy) range `35.235.240.0/20`
- **External Services:** Port 8888 (Code-Server), 8080 (Spark Master UI), 8081 (Worker UI)

2.4 Port Forwarding Architecture

External access to internal services is achieved through Google Cloud Load Balancer forwarding rules. A single static public IP is shared by multiple services using port-based routing:

Table 1: Port Forwarding Configuration

Public Port	Target Node	Service
8888	Runner (10.0.1.10)	Code-Server IDE
8080	Master (10.0.1.20)	Spark Master UI

2.5 Compute Instances Provisioning

The cluster consists of predictable static IP assignments for reliable service discovery:

- **Runner Node:** `10.0.1.10` (e2-medium, Ubuntu 22.04)
- **Master Node:** `10.0.1.20` (e2-medium, Ubuntu 22.04)
- **Worker Nodes:** `10.0.1.3x` (n workers, e2-medium)

2.6 Automated SSH Key Distribution

SSH authentication is secured through key-based authentication with automatic distribution via GCP metadata service. The Terraform configuration embeds the SSH public key in instance metadata, while the private key is securely transferred to the runner node for Ansible operations.

This approach ensures that:

1. No manual SSH key copying is required
2. Keys are securely distributed without exposing them in logs
3. The runner node can execute Ansible playbooks on all cluster nodes

3 Configuration Management with Ansible

3.1 Common Playbook (common.yaml)

The common playbook establishes the foundation for all Spark nodes by installing Java (JDK 1.8), Hadoop 2.7.1, and Spark 2.4.3. Key responsibilities include:

- Creating the `spark` user and group for service isolation
- Downloading and extracting Java, Hadoop, and Spark distributions
- Configuring environment variables (`JAVA_HOME`, `SPARK_HOME`, `HADOOP_HOME`)
- Setting up Spark configuration files (`spark-env.sh`, `spark-defaults.conf`)
- Creating necessary directories for logs, work files, and PID files
- Disabling password authentication and enabling SSH key authentication

3.2 Master Playbook (master.yaml)

The master playbook configures the Spark master node with systemd services for automatic startup and management

Additionally, the master node hosts the Spark History Server for reviewing completed job statistics and performance metrics.

3.3 Worker Playbook (worker.yaml)

The worker playbook dynamically configures worker nodes to connect to the master using the master's IP address from the Ansible inventory. Workers are configured with systemd for automatic restart on failure, ensuring cluster resilience.

3.4 Runner Playbook (runner-playbook.yaml)

The runner node serves as the operational hub, hosting:

- **Code-Server:** Web-based VS Code IDE accessible at `<PUBLIC_IP>:8888`
- **Service Discovery API:** Go-based HTTP server listening on port 3636
- **Ansible:** For orchestrating configuration across the cluster

4 Dynamic Worker Scaling with Service Discovery

4.1 Service Discovery Mechanism

A custom Go application (`service-automate`) implements a RESTful API for automatic worker registration. When new worker instances boot, they perform an HTTP POST to `http://10.0.1.10:3636/worker`, triggering the following workflow:

1. **Registration:** Worker's IP address is extracted from the request
2. **Inventory Update:** The Ansible inventory file is updated with the new worker entry
3. **Immediate Response:** HTTP 200 OK is sent to prevent worker startup timeout
4. **Background Configuration:** Ansible playbooks execute asynchronously to configure the worker

This architecture enables horizontal scaling by simply launching new worker instances, which automatically join the cluster without manual intervention.

5 Deployment Procedure

The deployment follows a streamlined six-step process executed from Google Cloud Shell:

1. **Clone Repository:**

```
git clone https://github.com/hzhoanglee/cloud-devops-2025-course
cd cloud-devops-2025-course/terraform

```

2. **Initialize Terraform:**

```
terraform init
terraform apply -auto-approve

```

3. **Access Code-Server:** Navigate to `http://<PUBLIC_IP>:8888` (password: `DuEmDaCoGangChoTinhYeuNguQuen`)

4. **Configure Master Node:**

```
ansible-playbook ./common.yaml -i inventory.ini \
  --limit spark_master \
  --ssh-common-args='-o StrictHostKeyChecking=no'

ansible-playbook ./master.yaml -i inventory.ini \
  --limit spark_master \
  --ssh-common-args='-o StrictHostKeyChecking=no'

```

5. **Configure Worker Nodes:** Workers auto-register and configure via service discovery

6. **Verify Deployment:**

```
cd /opt/cloud-devops-2025-course/java
spark-submit --class WordCount \
  --master spark://10.0.1.20:7077 \
  app.jar filesample.txt

```

6 Testing and Performance Evaluation

6.1 WordCount Application Test

The WordCount application was executed to validate cluster functionality with varying executor configurations. The test processes a large text file of 200MB (`filesample.txt`) to count word frequencies.

Table 2: WordCount Performance Results

Executors	Execution Time (s)	Tasks
1	15s	WordCount
4	14s	WordCount

6.2 Cluster Validation

Post-deployment verification confirmed:

- **Master UI:** Accessible at `http://<PUBLIC_IP>:8080` showing all workers
- **SSH Connectivity:** All nodes reachable via key-based authentication
- **Service Status:** Spark master, workers, and history server running
- **Auto-scaling:** New workers successfully registered via service discovery

7 Conclusion

This project successfully demonstrates a production-ready Apache Spark cluster deployment using modern DevOps practices. The infrastructure-as-code approach ensures repeatability and version control.

Key achievements include:

- Fully automated deployment from infrastructure to application layer
- Secure network architecture with VPC isolation and NAT gateway
- Automated SSH key distribution eliminating manual security configuration
- Dynamic worker scaling without manual intervention
- Production-ready systemd service management for high availability