UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

FINAL PROJECT

DAO Xuan Quy

CLOUD & BIG DATA

Automation of Spark Deployment with Ansible and Terraform on Google Cloud Service

---

Academic Year: 2024-2026

1. Architecutre and Methodology

I designed this project to automate Spark deployments on GCS using a two-step process. First, Terraform handles the Infrastructure as Code (IaC), and second, Ansible manages the software configuration. This approach ensures the environment is both scalable and easily repeatable.

1.1 Infrastructure Layer (Terraform)

Development of the Infrastructure as Code (IaC) took place locally using Terraform, which was then used to provision the environment on Google Cloud. For security, the architecture confines all resources to a single Virtual Private Cloud (VPC).

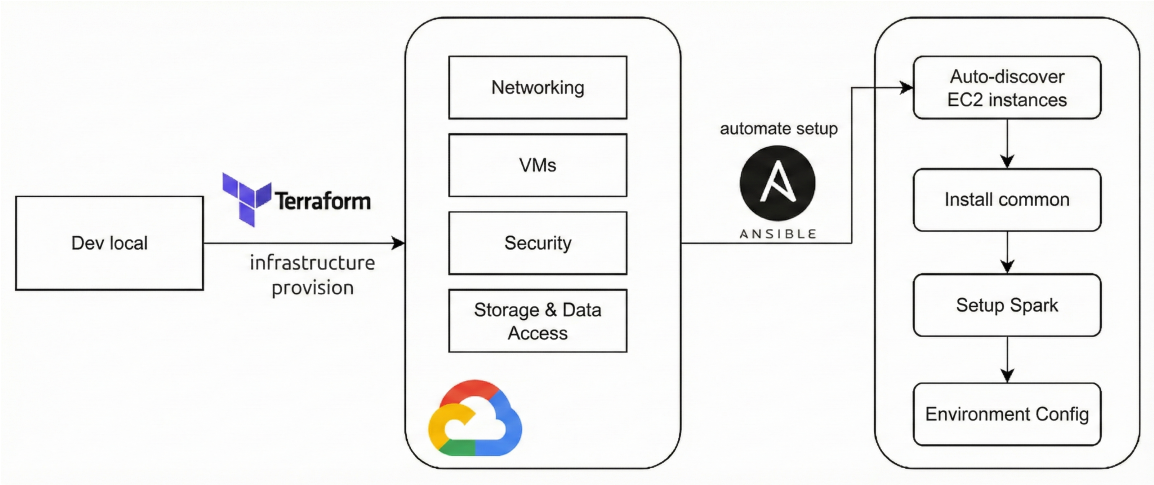

Figure 1.1: Infrastructure architecture

Table 1.1: Google Cloud Services Utilized in Terraform Configuration

Service	Terraform Resource(s)	Purpose in Project
Compute Engine	`google_compute_instance`	Provisioning Spark Master, Workers, and Edge nodes.
VPC	`google_compute_network` `google_compute_firewall`	Handling internal cluster communication and external SSH/UI access.
IAM	`google_service_account` `google_project_iam_member`	Managing identities for VMs to access GCS and Cloud Logging.
Cloud Storage	`google_storage_bucket`	Persisting Spark data and datasets (separate from boot disks).
Cloud Logging	*(Referenced via IAM role)*	Enabling workers to write system and application logs.

1.2 Software & Configuration Layer (Ansible)

Ansible was used to automate the configuration of all provisioned EC2 instances, ensuring consistency and repeatability.

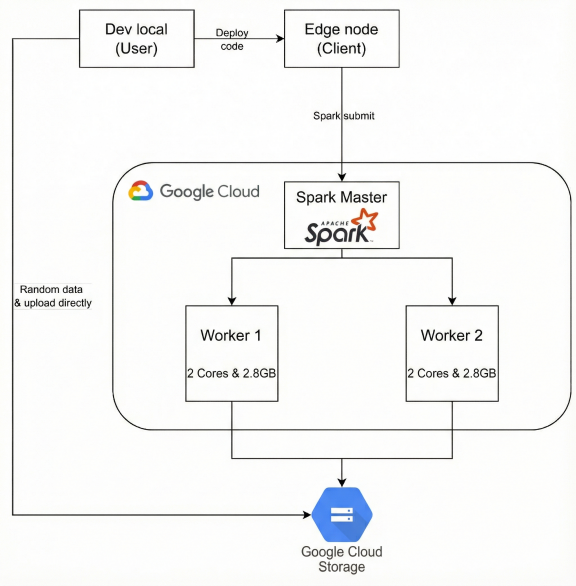

Figure 1.2: Framework & Configure architecture

The deployment of the Spark cluster is managed through a multi-stage Ansible playbook. The process is divided into common initialization steps applied to the entire cluster, followed by role-specific configurations for the Master, Worker, and Edge nodes.

1.2.1 Phase 1: Common Component Setup

The first play targets the `spark_cluster` group (comprising all nodes). Executed with root privileges, this phase ensures consistency and security across the infrastructure.

- **Base Dependencies:** The `common` role is executed to install prerequisite software (e.g., Java 8, basic system utilities).
- **Network Resolution:** The `/etc/hosts` file is updated using a Jinja2 template to ensure all nodes can resolve each other by hostname via the internal network.
- **Security Hardening:**
  - **SSH Hardening:** Password authentication is explicitly disabled in `/etc/ssh/sshd_config`.
  - **Root Restrictions:** Direct root login is disabled to prevent unauthorized privileged access.
  - **Service Restart:** The SSH daemon (`sshd`) is restarted to apply the security changes immediately.

1.2.2 Phase 2: Spark Role Configuration

Subsequent plays target specific node groups to configure the Apache Spark environment (Version 2.4.3)

1. **Spark Master Setup:**
   - **Target:** `master` group.

- **Action:** Executes the `spark` role with the parameter `spark_role: master`. This installs the Spark binaries and configures the node to operate as the cluster resource manager.

2. **Spark Worker Setup:**
   - **Target:** `workers` group.
   - **Action:** Executes the `spark` role with `spark_role: worker`.
   - **Cluster Registration:** The workers are configured with the `master_url` (port 7077) to allow them to register with the Master node automatically upon startup.
3. **Spark Client/Edge Setup:**
   - **Target:** `edge` group.
   - **Action:** Executes the `spark` role with `spark_role: client`.
   - **Purpose:** Configures the node for job submission (client mode) without starting background worker daemons.

1.2.3 Phase 3: Job submission and Testing

To validate the cluster's functionality and performance, an automated Bash script is utilized to orchestrate the workflow. The process begins by dynamically retrieving infrastructure metadata, such as the Edge node's public IP and the Google Cloud Storage (GCS) bucket name, directly from Terraform outputs. A sample dataset (`large_sample.txt`) is uploaded to the GCS bucket to simulate cloud-native data ingestion. The computational workload is triggered via an SSH connection to the Edge node, where the `spark-submit` command initiates the `JavaWordCount` application. The job is configured to utilize 2GB of executor memory and 4 cores, reading the input data directly from GCS and writing the processed results back to a dedicated output path in the bucket. Finally, the script performs verification by retrieving the generated output files using `gsutil` and displaying the top word frequency results to ensure data integrity.

1.2.4 Environment Variables

The playbook enforces specific versioning and paths across the cluster, as detailed in Table 1.2.

Table 1.2: Ansible Playbook Configuration Variables

Variable	Value
Spark Version	2.4.3
Hadoop Version	2.7
Java Home	`/usr/lib/jvm/jdk1.8.0_202`
Spark Home	`/opt/spark`
Master Port	7077

2. Benchmarking & Concluding

2.1 Benchmarking

The primary objective of the benchmarking phase was to verify the cluster's operational integrity and analyze its performance behavior under escalating workloads. To achieve this, the WordCount application was executed against datasets of increasing size, utilizing various submission parameters to constrain worker node resources.

The recorded execution time represents the total end-to-end latency of the job. This metric encompasses the entire processing pipeline, including data ingestion from cloud storage, the map and shuffle phases, the reduction operations, and the final persistence of results back to the storage layer.

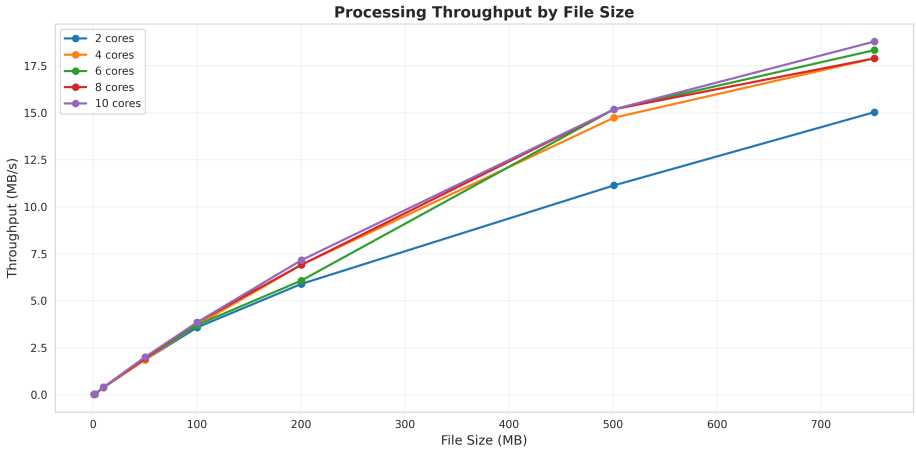

Figure 2.1: Comparison between different cores and file size

2.2 Conclusion and Recommendations

This project demonstrated the complete automation of a scalable Apache Spark cluster on AWS. The use of Terraform for infrastructure and Ansible for configuration provides a powerful, repeatable, and version-controlled methodology.

The benchmarking results clearly show that while the architecture is "beautiful", resource allocation is paramount. The OutOfMemoryError on the 100MB file highlights that the default or a naive memory configuration is insufficient for even moderately sized jobs. The key takeaway is that Spark performance is not just about the number of cores, but critically about providing each executor with sufficient memory (–executor-memory) and disk space.