UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

FINAL PROJECT

Nguyen Thien An - 2440048

CLOUD COMPUTING

Automation of Spark Deployment with Ansible and Terraform on AWS

Academic Year: 2024-2026

1. Context

This project focuses on automating the deployment of an Apache Spark environment cluster using Terraform and Ansible on Google Cloud Platform (GCP).
The overall architecture contains two layers:

- Infrastructure Layer (Terraform) – responsible for creating the cloud resources.
- Software & Configuration Layer (Ansible) – responsible for installing and configuring Spark and its runtime environment.

1.1 Infrastructure Layer (Terraform)

The entire cloud environment is implemented through Terraform. All components are deployed inside a dedicated Virtual Private Cloud (VPC) to maintain network isolation and secure intra-cluster communication. After running the terraform file, an (`inventory.ini`) file is created for the Ansible playbook.

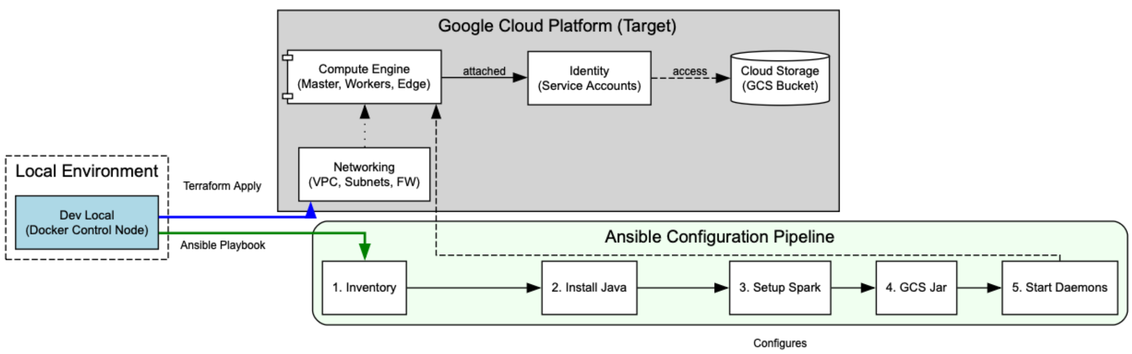

Figure 1.1: GCP Infrastructure Architecture

- **Network Architecture:** A custom VPC (`spark-vpc`) was established in the `us-central1` region with one subnet
  - Subnet: spark-subnet
  - CIDR: 10.0.1.0/24
  - Provides private internal IPs for all cluster nodes.
- **Compute Instances:** The cluster consists of some Google Compute Engine instances, all utilizing the `ubuntu-os-cloud/ubuntu-2204-lts` image:
  - **Spark Master (1):** An `e2-medium` instance responsible for resource negotiation and cluster management.
  - **Spark Workers (2-6):** Depending on the number, we have `e2-medium` instances that execute the actual data processing tasks.
  - **Edge Node (1):** An `e2-small` instance serving as the cluster gateway. This node is the only entry point for SSH access and job submission, isolating the cluster internals from external access.
- **Security (Firewall Rules):** Terraform provisions a set of stateful firewall rules to control traffic:
  - **allow-ssh**

    - Ingress on port 22
    - Source: `0.0.0.0/0`
    - Enables administrative access.
  - **allow-spark-ui**
    - Ports: 8080 (Master UI), 4040 (Application UI)
    - Allows external monitoring of Spark interfaces.
  - **allow-internal**
    - Allows all TCP/UDP traffic within `10.0.1.0/24`
    - Ensures unrestricted communication between Master and Worker nodes.

- **Storage & Identity:** Instead of deploying an HDFS cluster, the environment uses a Google Cloud Storage (GCS) bucket as the central data lake.

  To enable secure, credential-free access:

  - All VMs are assigned a Service Account.
  - The account is granted the `cloud-platform` OAuth scope.
  - Spark applications can seamlessly read/write data using the `gs://` protocol.

1.2 Software & Configuration Layer (Ansible)

Ansible was used to automate the software provisioning of all instances. The execution was performed from a Docker container to ensure compatibility across different host operating systems.

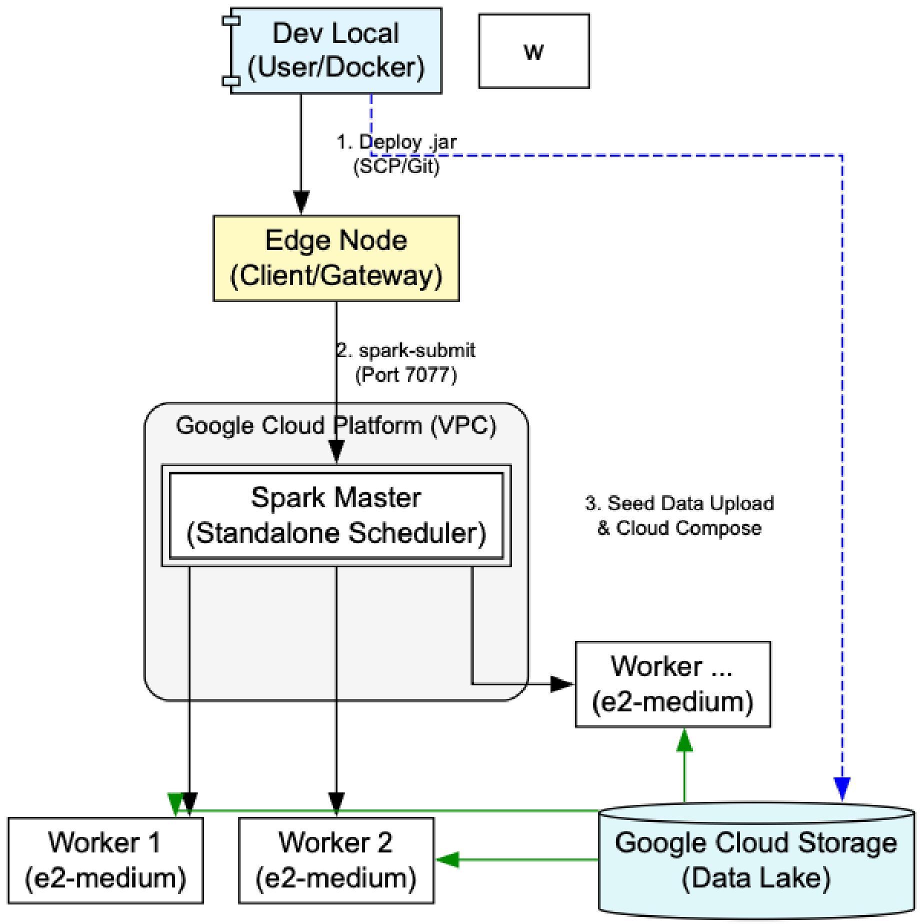

Figure 1.2: Software Configuration Architecture

- **Static Inventory:** `inventory.ini` file was is created for the mapping of Terraform output IPs to Ansible groups (`[master]`, `[workers]`, `[edge]`).
- **Common Configuration:** A playbook (`site.yml`) applied a baseline configuration to all nodes, including the installation of `openjdk-11-jdk`, system dependencies (`wget`, `tar`), and the extraction of the Spark 3.5.0 binaries to `/opt/spark`.
- **Cloud Connector Integration:** The automated download of the `gcs-connector-hadoop3.jar` is use to the Spark classpath, enabling the standalone cluster to interface directly with Google Cloud Storage.
- **Service Orchestration:**
  - On the Master node, the `start-master.sh` script was executed.
  - On Worker nodes, the `start-worker.sh` script was executed, configured to connect specifically to the Master's **Internal IP** (e.g., `spark://10.0.1.10:7077`). This usage of internal networking was vital for the stability of the distributed system.

2. Methodology

The methodology follows a DevOps-centric approach, utilizing a docker container to run cloud resources and configuration.

2.1 Infrastructure as Code (IaC) with Terraform

The entire GCP infrastructure was defined within Terraform.

- **Provisioning Workflow:** The deployment was executed from a Docker container.
- **Identity Management:** A critical step in the methodology was attaching specific Service Accounts to the Virtual Machines via Terraform. By enabling the `cloud-platform` access scope, the instances are granted implicit permission to interact with Google Cloud Storage without requiring manual API key management.
- Terraform creates the (`inventory.ini`) to specify the different groups for the ansible playbook.

2.2 Automated Configuration with Ansible

Ansible was used to convert the raw Ubuntu VMs into a functional Spark cluster.

- **Inventory Management:** (`inventory.ini`) is created after the terraform set up to map the dynamic Public IPs of the instances to their roles (`master`, `workers`, `edge`).
- **Library Injection:** A specific role was executed to download the `gcs-connector-hadoop3.jar` and inject it into the Spark classpath. This was essential for enabling the "Cloud-Native" architecture where Spark reads directly from object storage buckets (`gs://`).
- **Service Orchestration:** The playbook automated the startup of the `start-master.sh` and `start-worker.sh` daemons, ensuring the workers were correctly pointed to the Master's internal network address.

2.3 Data Generation Strategy (Server-Side)

To overcome the storage limitations of the `e2-small` Edge Node, we devised a server-side data generation strategy. Instead of generating an 8GB file, a 1GB data file was uploaded to a GCS bucket. We can duplicate and concat this file there but here, only the 1GB data file was used for testing.

2.4 Batch Processing vs Streaming

While the batch processing pipeline functioned within expected parameters, the real-time streaming implementation presented specific synchronization challenges when handling high-velocity data.

- **Ingestion Rate vs. Initialization Latency:** By piping a **1GB** dataset directly from Google Cloud Storage into the network socket using `gsutil cat | nc`, a critical issue was identified where the Producer (Data Pipe) began transmitting at maximum throughput before the Spark Streaming Consumer was fully initialized and ready to schedule micro-batches since it wasn't fully automated. This resulted in a "Denial of Service" condition.

- **Resolution Strategy:** The issue can be fix by, for now, manually synchronizing the deployment pipeline. By initializing the Spark Consumer job first and ensuring the Receiver was active before triggering the high-velocity data stream, the cluster successfully established back-pressure handling. This allowed the `e2-medium` workers to ingest and process the full 1GB stream in 1-second intervals.

3. Benchmarking & Concluding

3.1 Benchmarking Results

The primary goal of the benchmarking phase was to validate the cluster's distributed computing capabilities by observing its performance characteristics under different resource constraints. The `WordCount` application was executed against the dataset stored in Google Cloud Storage, utilizing a cluster configuration of one `e2-medium` Master and several `e2-medium` Workers.
The execution time measured captures the full job lifecycle: reading from GCS (Input), the map phase (`flatMap`, `mapToPair`), the shuffle, the reduce phase (`reduceByKey`), and writing the output back to the Cloud Storage bucket.
To test the parallelize speed up: Benchmark were run across the 2-5 worker nodes.

	2 Node	**3 Nodes**	**4 Nodes**	**5 Nodes**
Speed (ms)	50517	43588	32510	29787

Table 3.1: Speed vs Number of Nodes

3.2 Conclusion and Recommendations

This project demonstrated the complete automation of a scalable Apache Spark cluster on Google Cloud Platform. The integration of Terraform for infrastructure provisioning and Ansible for configuration management provided a robust, repeatable, and version-controlled deployment methodology.
Recommendations for Future Iterations: While the batch processing was efficient, the streaming tests revealed that the default heap memory of `e2-medium` instances is a limiting factor for high-velocity ingestion. Future deployments should explicitly tune the `spark.executor.memory` or vertically scale the worker nodes to `e2-standard-4` to handle larger micro-batches.