UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

FINAL PROJECT

Nguyen Viet Tung - 2440049

CLOUD COMPUTING

Automation of Spark Deployment with Ansible and Terraform on AWS

Academic Year: 2024-2026

1. Architecture

The architecture of this project is designed to provide a repeatable, scalable, and automated Apache Spark environment on Amazon Web Services (AWS). It includes a combination of Infrastructure as Code (IaC) and configuration management tools to achieve this. The architecture is composed of two primary layers: the Infrastructure Layer managed by Terraform, and the Software Configuration Layer managed by Ansible.

1.1 Infrastructure Layer (Terraform)

The cloud infrastructure was provisioned entirely on AWS using Terraform that being developed on local. All resources reside within a single Virtual Private Cloud (VPC) to ensure network isolation.

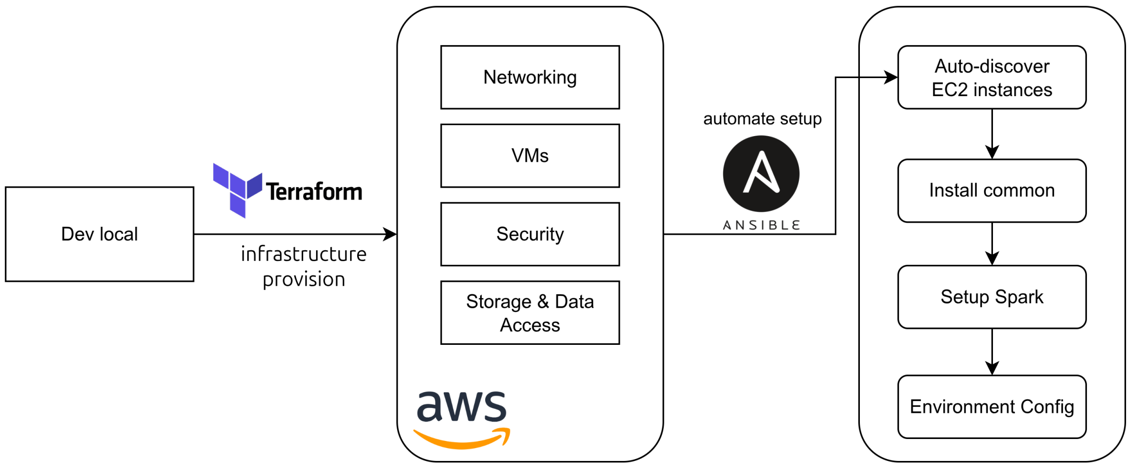

Figure 1.1: Infrastructure architecture

- Networking: A VPC (10.0.0.0/16) was established in the ap-southeast-1 region, containing a single public subnet (10.0.1.0/24). An Internet Gateway and a Route Table were configured to provide internet access to all instances for software downloads and SSH access.
- Compute Instances: The cluster consists of three distinct instance roles, all using the *ami-00d8fc944fb171e29* (Ubuntu 24.04) image: Spark Master (1): A single EC2 instance responsible for coordinating the cluster and managing worker nodes. Spark Workers (4): Four EC2 instances that execute Spark tasks. The number of workers is parameterized using a Terraform variable (worker_count) for easy scaling. Edge Node (1): A dedicated EC2 instance used as a client for submitting Spark jobs via spark-submit. This isolates the user environment from the master node.
- Security: A single AWS Security Group (spark-sg) governs all network traffic. It allows inbound SSH (port 22) from any IP for management and all internal traffic between nodes within the group itself. This enables seamless communication between the Spark master and workers.
- Storage & Data Access: An Amazon S3 bucket (usth-spark-project-data-tung-20251121) serves as the persistent data layer for both input files and job output. To provide secure, password-less access, an IAM Role with AmazonS3FullAccess policy was created and attached to all EC2 instances via an IAM Instance Profile. This is a security best practice that avoids storing credentials on the instances.

1.2 Software & Configuration Layer (Ansible)

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

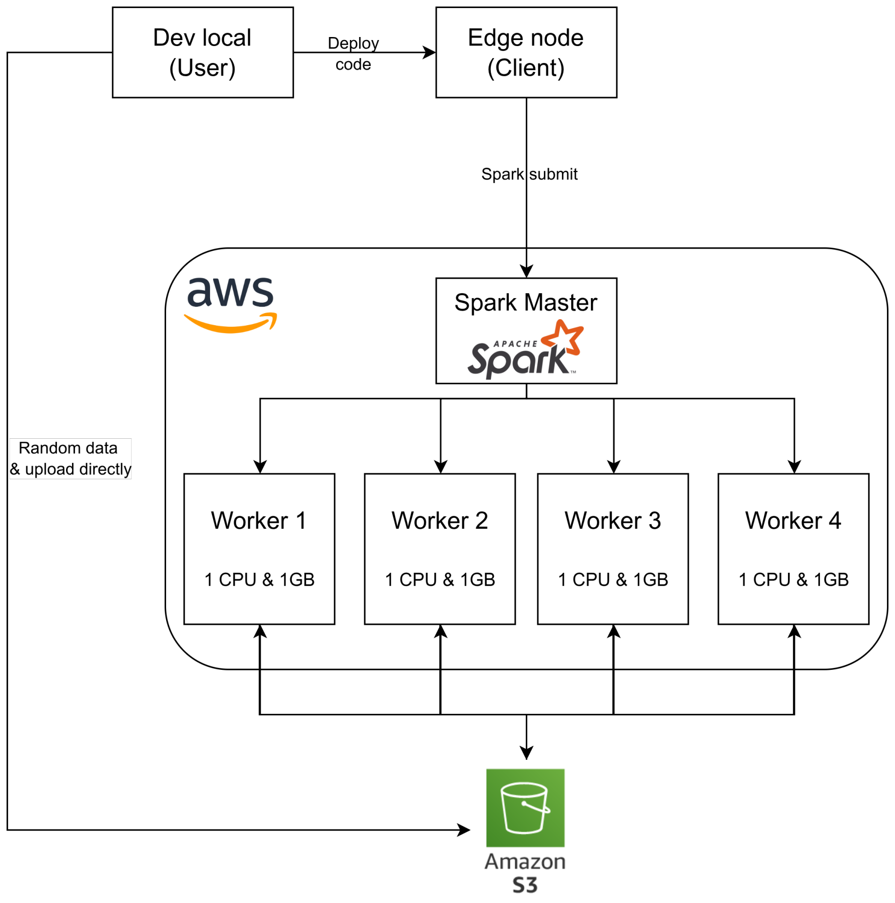

Figure 1.2: Framework & Configure architecture

- Dynamic Inventory: Ansible uses the aws_ec2 dynamic inventory plugin to automatically discover and group the running EC2 instances based on their assigned tags (spark-master, spark-worker, spark-edge-node).
- Common Role: A baseline configuration is applied to all instances, which includes updating the package cache and installing openjdk-11-jre-headless and scala. It also downloads and unarchives the Spark 3.5.0 distribution to /opt/spark.
- Spark Service Management: systemd service files are templated and deployed for both the master and worker nodes. This ensures that the Spark daemons start automatically on boot and are managed as proper system services. A key learning was to set the service Type to forking to correctly manage Spark's background processes. Additionally, here I choose to install Spark ***standalone*** to reduce the heavily workload of Hadoop Cluster, as my VMs are only t3.small . Furthermore, we are more interest on Spark itself (through Yarn is also exceptionally important).
- Environment Configuration: A spark-env.sh file is templated to configure the Spark environment, notably setting SPARK_WORKER_MEMORY and enabling automatic cleanup of old application data from the worker directories to prevent disk space exhaustion.

2. Methodology

1. Infrastructure as Code (IaC) with Terraform: The entire AWS infrastructure was defined in Terraform configuration files. This approach allows for the entire cluster to be created, modified, or destroyed with simple commands (terraform apply, terraform destroy). The deploy.sh script encapsulates this, first running terraform apply to build or update the cloud resources.

2. Automated Configuration with Ansible: After infrastructure provisioning, the deploy.sh script calls an Ansible playbook (playbook.yml). Ansible connects to the newly created instances (discovered via the dynamic inventory) and executes a series of roles to:

3. Install common dependencies on all nodes.

4. Configure the master node to run the spark-master service.

5. Configure all worker nodes to run the spark-worker service, pointing them to the master's private IP address.

6. Job Submission and Testing: The WordCount application, written in Java and packaged with Maven, was used as the benchmark application. A run_benchmark.sh script was developed to automate the testing process. This script is copied to the edge node and, when executed, runs the spark-submit command against a series of input files of varying sizes stored in S3.

7. Iterative Debugging and Enhancement: The initial deployment was not without issues. The methodology allowed for rapid debugging and enhancement:

    - Memory Errors: Initial runs with larger files resulted in java.lang.OutOfMemoryError. This was resolved by explicitly setting –executor-memory in the spark-submit command to allocate sufficient heap space.
    - Disk Space Errors: The cluster later failed with No space left on device errors. This was traced to the small default EBS volume size and the accumulation of temporary files in the Spark work directory. This was solved by adding a root_block_device block in Terraform to provision larger (20GB) disks and configuring automatic cleanup in spark-env.sh.
    - Instance Type Limitations: An attempt to scale up to t3.medium instances failed due to AWS Free Tier account limitations, demonstrating a real-world constraint. The configuration was reverted to t3.small.

3. Benchmarking & Concluding

3.1 Benchmarking

The primary goal of the benchmarking was to validate the cluster's functionality and observe its performance characteristics under varying loads. The WordCount application was run against text files of increasing size on a cluster of one t3.small master and four t3.small workers, with different command of submission to limit the worker node.

The execution time measured in the WordCount.java application captures the full job lifecycle: reading from S3, the map phase (flatMap, mapToPair), the shuffle, the reduce phase (reduceByKey), and writing the output back to S3.

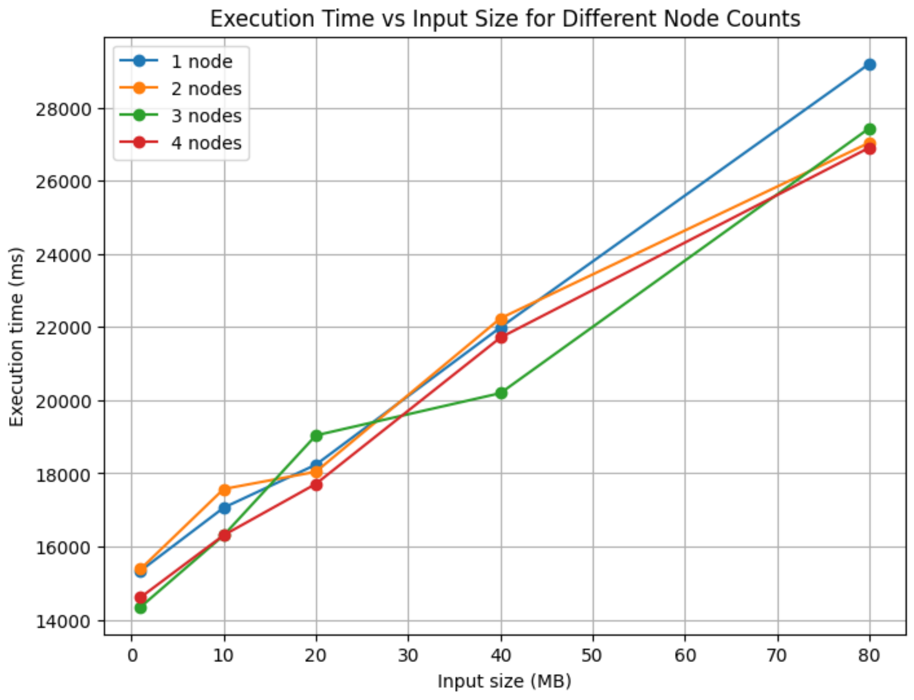

Figure 3.1: Comparison between different nodes and file size

3.2 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 nodes, but critically about providing each executor with sufficient memory (–executor-memory) and disk space.