Google Cloud Dataproc Integration in the Flight Delay Prediction Project

**Repository:** <https://github.com/MalikChettihIA/Emiasd-FlightProject>

# 1. Introduction

This section details the comprehensive integration of Google Cloud Dataproc into the Flight Delay Prediction project. This distributed machine learning system aims to predict flight delays, and our goal was to transform the original local Spark application into an automated, scalable, and cloud-native solution. The following content will demonstrate this transformation, covering all facets of the DevOps lifecycle: development, integration, deployment, and operation.

The project illustrates best practices in MLOps (Machine Learning Operations) applied to a cloud environment, emphasizing reproducibility, observability, and resource efficiency.

# 2. Context of the Flight Delay Prediction Project

## 2.1 Business Objective

The project aims to predict flight delays by analyzing flight and weather data. The application processes:

* **142,000+ flights** with temporal data
* **44 features** after feature engineering
* **Datasets of several GB** after transformation

## 2.2 Technical Architecture and Key Components for Data Processing and Machine Learning

### DataProc: Key Components and Services for Data Processing

The **Google Cloud Dataproc** solution is a managed computing platform designed for distributed data processing and machine learning (ML) experiments. It relies on a set of fundamental technological components.

* **Apache Spark 3.5.3:** This is the distributed data processing engine par excellence. Spark is renowned for its speed (thanks to in-memory execution) and versatility, allowing for batch processing, real-time *streaming*, as well as interactive queries. Version 3.5.3 was specifically selected to target a precise machine, thus ensuring perfect alignment with the one used by LAMSADE.
* **Scala 2.12:** This programming language is often favored for developing Spark applications. Scala combines object-oriented and functional paradigms, offering great conciseness and performance for writing distributed data processing code. The choice of this version ensures uniformity with the LAMSADE environment. Scala, which combines object-oriented and functional programming paradigms, was selected for its conciseness and performance in distributed data processing.

### ML Monitoring and Management Tool

**MLflow:** This tool is of paramount importance for monitoring and managing the lifecycle of machine learning experiments. MLflow notably offers the following functionalities:

* **Tracking:** Allows recording of parameters, metrics, and artifacts associated with models.
* **Projects:** Ensures the encapsulation of ML code in a format guaranteeing reproducibility.
* **Models:** Facilitates the management and deployment of machine learning models.
* **Model Registry:** This central element centralizes the referencing, version management, and transition of models to a production environment.

**Platform and Storage Service**

* **Google Cloud Dataproc:** It is the managed computing platform solution provided by Google Cloud. Dataproc simplifies the deployment, management, and scaling of Apache Spark, Hadoop, Flink, and other Big Data tools clusters. It allows users to focus on data analysis rather than infrastructure administration.
* **Google Cloud Storage (GCS):** This service acts as the storage service for data and artifacts. GCS is a highly available, durable, and scalable object storage service, ideal for:
  + Storing raw and processed datasets used by Spark.
  + Backing up models and artifacts generated by MLflow.
  + Serving as a high-performance distributed file system decoupled from Dataproc clusters.

## 2.3 ML Pipeline

1. **Data Pipeline:** Data ingestion and cleaning
2. **Feature Extraction:** Feature generation
3. **Training:** Random Forest model training
4. **Evaluation:** Performance metrics and logging

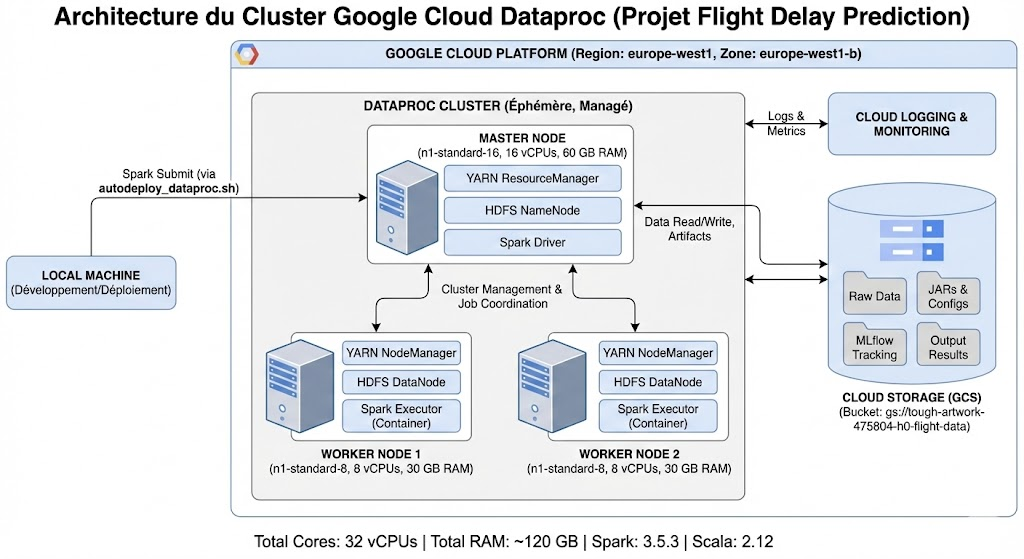
# 3. General System Architecture

## 3.1 Local Architecture (Development)

## 3.2 Cloud Architecture (Production)

The **Google Cloud Platform (GCP)** architecture set up for the *flight* project is robust, focused on availability, *scalability*, and performance. It is structured around four pillars:

* **Cloud Storage (GCS):** Used to store data, JARs, configurations, and outputs.
* **Dataproc Workflow:** A managed ephemeral cluster composed of one *Master Node* (n1-standard-16) and two *Worker Nodes* (n1-standard-8 × 2).
* **MLflow Tracking:** Whose tracking data is stored in GCS.
* **Cloud Monitoring & Cloud Logging:** Performance monitoring, log centralization, and operational alerts.



## 3.3 Data Flow

The chart below illustrates the flow.



The data flow describes the sequence of operations from the ingestion of raw data to the production of the prediction model and its results in the cloud.

1. **Data Source:** Raw datasets (flights, weather) are stored in a dedicated **Google Cloud Storage (GCS)** bucket.
2. **Processing (Dataproc):** The ephemeral Dataproc cluster is created and executes the Spark/Scala application. It reads the data from GCS, performs cleaning and *feature engineering*.
3. **ML Training:** The *Training* step uses Spark MLLib's distributed Random Forest model on the processed data.
4. **Tracking and Artifacts (MLflow):** Parameters, metrics, and the serialized model are recorded by **MLflow** and stored in a dedicated subdirectory of **GCS**.
5. **Results:** Predictions and final model artifacts are written to an output GCS bucket, ready for consumption or deployment.

This architecture was designed so that the data is not subject to any persistent storage on the Dataproc nodes. The cluster is assigned an exclusively computational role, which consequently optimizes both performance and cost management.

# 4. Code and Scripts Development

In this project, we opted for complete process automation via the command line (with gcloud and gsutil), prioritizing this approach over using the GCP web interface. This strategic decision presents several major advantages:

* **Guaranteed Reproducibility:** The use of a single execution script ensures the uniformity of results, regardless of the deployment environment.
* **Optimal Traceability:** Direct integration of commands into the source code facilitates monitoring (via versioning) and guarantees flawless traceability.
* **Efficiency Gain:** Cluster management is fast and automated, eliminating time-consuming manual operations.
* **Ideal for CI/CD:** This method is perfectly suited for deploying the infrastructure in our automated continuous integration pipelines.

This approach requires a preliminary environment preparation step, both for development and deployment.

## 4.0 Environment Prepared for Development and Deployment on Dataproc

For the deployment of the application on Dataproc, we carried out the following preparation steps on our workstation:

* **Setting up the Execution and Compilation Environment:**
  + We needed and installed the **Java Development Kit (JDK)**, version 8 or higher, for running JVM applications.
  + We configured the specific version of **Scala** required by the project (verified in build.sbt).
  + We installed and configured **SBT (Simple Build Tool)** to manage Scala dependencies and compile the project efficiently.
* **Configuration of Project Management and Version Tools:**
  + We set up **Git** (version control system) to clone the source code repository and ensure version management.
* **Configuration of Google Cloud Access:**
  + We needed and installed and configured the **Google Cloud SDK (gcloud CLI)** to interact with Google Cloud Platform (GCP).
  + We secured access by **authenticating** via the gcloud auth login command.
  + We defined the target GCP project as the default project with the command: gcloud config set project tough-artwork-475804-h0
* **Choice and Configuration of the Development Environment:**
  + We selected and configured a **Code Editor / IDE** (such as IntelliJ IDEA or VS Code) supporting Scala and SBT to facilitate code writing and testing.

## 4.1 Scala Code Structure

The code architecture is detailed either on the GitHub repository or in section XXX.

## 4.2 YAML Configuration

This YAML file is the **masterpiece of configuration** for the flight delay prediction application. It contains **all the parameters** required for executing the machine learning pipeline in the **production environment (here GCP)**.

**Fundamental Philosophy:** The experiment can be modified **without altering the code**; simply update this YAML file.

## 4.3 Build and Deployment Scripts

* autodeploy\_dataproc.sh: Main deployment script.
* configure\_dataproc.sh: Script for interactive configuration.

# 5. Integration with Google Cloud Dataproc

## 5.1 Choice of Dataproc

The choice of Dataproc is justified by several major assets:

* **Managed Service:** Eliminates the burden of infrastructure management.
* **Cost-Effectiveness:** Use of ephemeral clusters billed on a pay-per-use basis.
* **Optimal Integration:** Native compatibility with key tools like Spark, GCS, and MLflow.
* **Guaranteed Reproducibility:** Ensures that any evaluator can recreate the exact environment to validate the results.

## 5.2 GCP Project Configuration

This section aims to provide a comprehensive overview of the crucial parameters configuring the DataProc environment used for data processing. This information is essential for any maintenance operation, workload deployment, or troubleshooting.

### Project Identifiers and Location

The DataProc environment is hosted within a specifically dedicated Google Cloud Platform (GCP) project.

| **Parameter** | **Label (EN)** | **Value** | **Detailed Description** |
| --- | --- | --- | --- |
| **ID Project** | Project ID | tough-artwork-475804-h0 | Unique and global identifier for the GCP project. All interactions with APIs or resources must reference this ID. |
| **Region** | Region | europe-west1 | The main region where DataProc computing resources are deployed. |
| **Zone** | Zone | europe-west1-b | The specific availability zone within the region. |

### 

### Data Storage Infrastructure (Google Cloud Storage - GCS)

The DataProc cluster uses a GCS *bucket* as the persistent storage layer for input data, intermediate results, job logs, and final results.

| **Parameter** | **GCS Reference** | **Value** | **Role in the Architecture** |
| --- | --- | --- | --- |
| **GCS Location** | GCS Bucket URI | gs://tough-artwork-475804-h0-flight-data | This *bucket* is the main repository for flight data (raw and processed). It is virtually mounted on the DataProc cluster for low-latency access (Spark/Hadoop) and serves as the default location for cluster logs/events history. |

Precise knowledge of these parameters is vital for:

1. **Job Submission:** Ensuring that gcloud dataproc jobs submit commands correctly specify the --region (europe-west1) and the --project (tough-artwork-475804-h0).
2. **Access Management:** Verifying that the service accounts associated with the cluster have the necessary IAM permissions to read/write to the *bucket* gs://tough-artwork-475804-h0-flight-data.
3. **Monitoring:** Configuring dashboards and alerts in Cloud Monitoring by targeting the resources in this region and project.

## 5.3 MLflow Integration

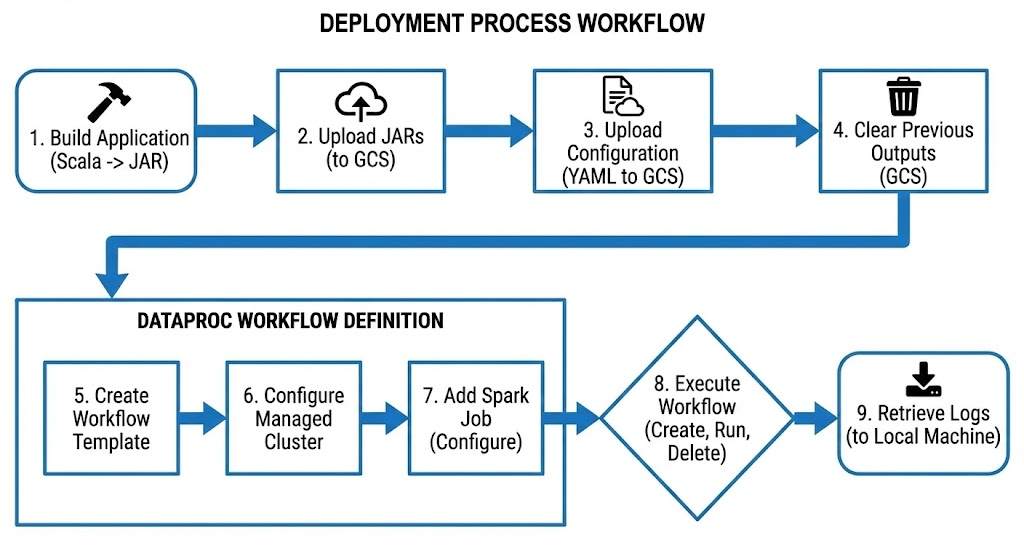
MLflow tracking is configured to use GCS:

## 5.4 Dependency Management

* MLflow JARs uploaded to GCS
* Configuration via --jars and --files in spark-submit

# 6. Automated Deployment

The diagram below illustrates the integration of Google Cloud Platform services and the data flow for executing the flight delay prediction pipeline via Dataproc.



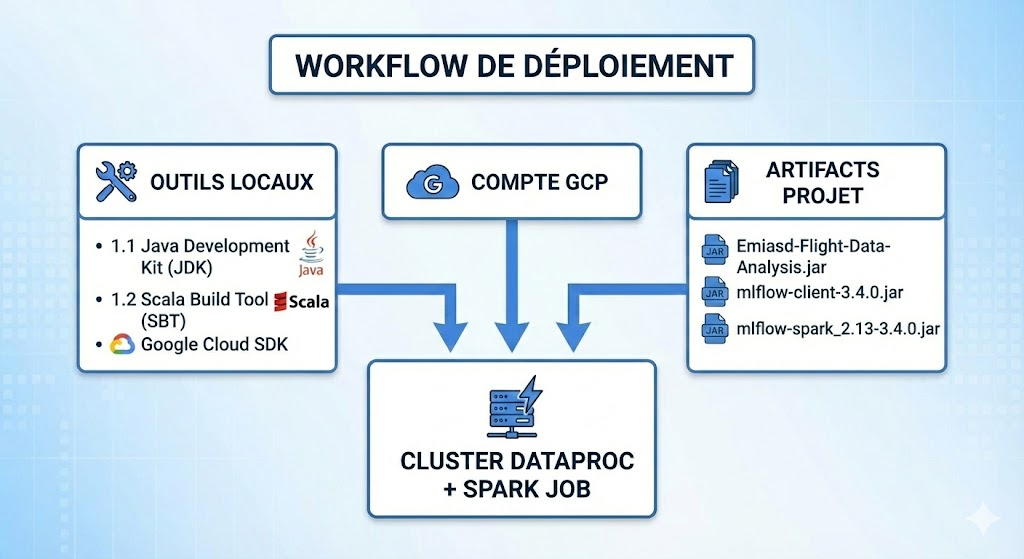
## 6.1 Main Script: autodeploy\_dataproc.sh

The automated script:

1. **Build:** sbt clean package
2. **Upload:** JARs and configs to GCS
3. **Workflow Creation:** Idempotent Dataproc Template
4. **Cluster Configuration:** Managed ephemeral cluster
5. **Job Submission:** Spark job with parameters
6. **Monitoring:** Extraction of the operation ID

## 6.2 Interactive Configuration: configure\_dataproc.sh

**DataProc Key Steps:**



* **GCP Authentication:** Connection to Google Cloud Platform.
* **Configuration:** Setting up the project and the storage *bucket*.
* **Resources:** Parameterizing the *cluster*.

## 6.3 Environment Variables

The .env.dataproc file contains:

GCP\_PROJECT\_ID=tough-artwork-475804-h0

GCS\_BUCKET=gs://tough-artwork-475804-h0-flight-data

WORKFLOW\_NAME=flight-delay-workflow

# ... other parameters

# 7. Usage and Monitoring

## 7.1 Launching the Deployment

**1. Initial Configuration:** Execute the configuration script:

./configure\_dataproc.sh

**2. Automated Deployment: With default tasks**

****./autodeploy\_dataproc.sh

**3. With specific tasks (e.g., *data-pipeline* and *feature-extraction*)**

****./autodeploy\_dataproc.sh "data-pipeline,feature-extraction"

## 7.2 Real-Time Monitoring

* **GCP Console:** <https://console.cloud.google.com/dataproc>
* **Workflow Progress:** Operation tracking via operation ID
* **Logs:** Via Cloud Logging or YARN UI

## 7.3 DataProc Monitoring Interfaces via SSH Tunnel

Access and Supervision of the DataProc Cluster

To ensure effective supervision of our cluster, we have set up SSH tunnels to directly access the following monitoring interfaces:

| **Interface** | **URL (via SSH tunnel)** | **Main Role** |
| --- | --- | --- |
| **YARN** | `http://flight-temp-cluster-utcztjemtipca-m:8088` | Monitoring of running jobs. |
| **Spark History** | `http://flight-temp-cluster-utcztjemtipca-m:18080` | Detailed history of completed Spark applications. |
| **HDFS NameNode** | `http://flight-temp-cluster-utcztjemtipca-m:9870` | Display of the distributed file system status. |

1. YARN ResourceManager

* **Function:** This is the core of cluster resource management, acting as the main scheduler for YARN (Yet Another Resource Negotiator).
* **Interest for supervision:** Essential interface for **monitoring active applications**. It allows **visualization of resource allocation** (CPU, memory), checking the status of NodeManagers, and **performing crucial diagnostics** in case of resource assignment issues for different jobs (Spark, MapReduce).

1. Spark History Server

* **Function:** This server is dedicated to the persistence and presentation of the complete history of Spark applications after their completion.
* **Interest for supervision:** Essential for **post-execution performance analysis**. It offers a detailed view of **Spark operations** (stages, tasks, *shuffles*), facilitating the identification of bottlenecks, the evaluation of data distribution, and the **determination of necessary optimizations** for future processing.

1. HDFS NameNode

* **Function:** Control interface for the NameNode, the central component of HDFS (Hadoop Distributed File System) responsible for namespace management and regulating data access.
* **Interest for supervision:** Key role in **verifying file system integrity**. It allows monitoring the overall **storage space** and its usage, ensuring the proper functioning of DataNodes, **inspecting file blocks**, and **diagnosing anomalies** related to replication or data access.

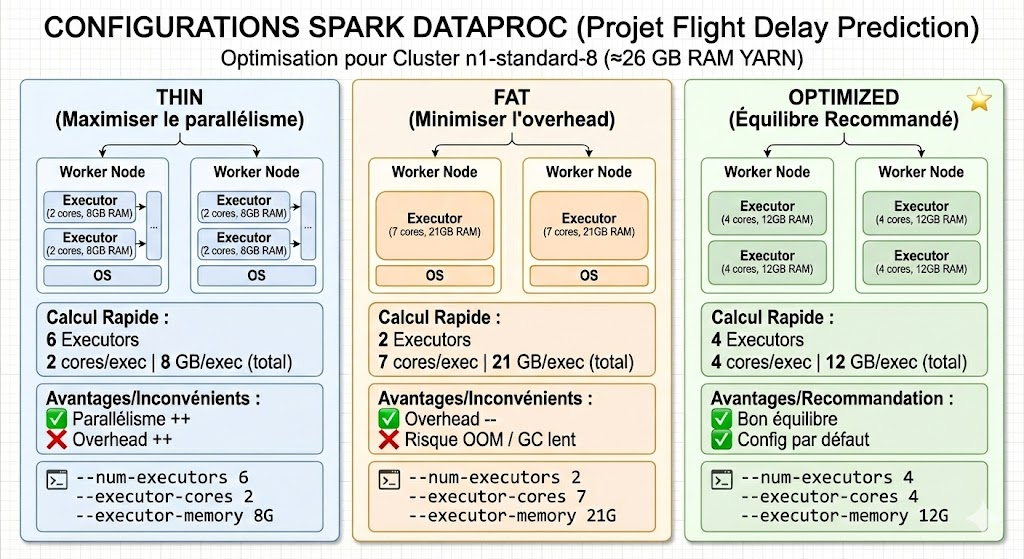
## 7.4 Results Retrieval

Results are saved in Google Cloud Storage (GCS). Once the pipeline is complete, the data is downloaded locally.

**8. Optimal Spark Executor Sizing: Comparative Study Thin/Fat/Optimized**

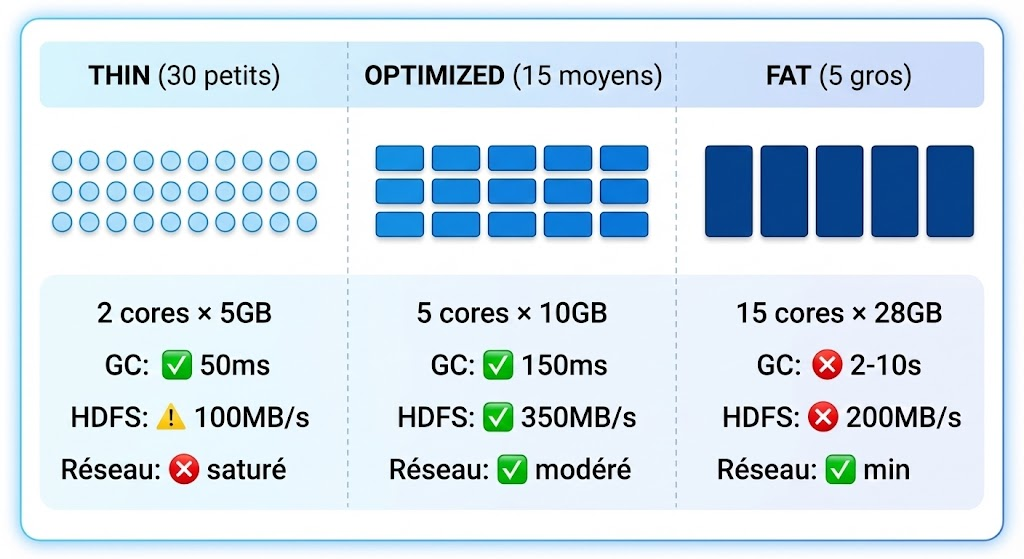
The objective of this section is to **determine the optimal configuration of Spark resources** (executors, cores, and memory) on the LAMSADE cluster. This is to ensure the **best possible performance** when executing flight delay prediction tasks.

We adjusted and compared different configurations (**THIN, FAT, and Optimized**) to evaluate their respective advantages and disadvantages, while considering the limits of available DataProc resources. The calculations take into account **YARN constraints** (notably the maximum memory allocation limit, and the **specifics of the n1-standard machine types** used.



**8.1 Review of Principles**

Here is a reorganization and reformulation of the text on executor configuration:



**Executor Configuration Strategies**

| **Strategy** | **Description** | **Advantages** | **Disadvantages/Consequences** |
| --- | --- | --- | --- |
| **THIN (Narrow)** | **Many small executors.** | Maximizes task execution parallelism. | Potential increased *overhead* related to managing a large number of instances. |
| **FAT (Large)** | **Few large executors.** | Minimizes network-related overhead. | Reduces the level of parallelism. |
| **Optimized** | **Balance** between the number and size of executors. | Achieves the best compromise between parallelism and overall processing efficiency. | Requires fine-tuning based on the specific workload. |

**8.2 THIN Configuration (Many small executors)**

This calculation is based on the following Dataproc cluster configuration:

* **Cluster:**
  + 1 Master node (machine type: n1-standard-16)
  + 2 Worker nodes (machine type: n1-standard-8)
* **Total Cluster Capacity:**
  + **Cores:** 16 (Master) + 2 × 8 (Workers) = 32 cores
  + **RAM:** Approximately 120 GB (with a YARN limit of about 26 GB per container)

**Spark/YARN Parameters (Calculator):**

* **executor-cores:** 2 (Conservative/small value)
* **Executors per Worker:** (8 Worker cores - 1 reserved core) ÷ 2 cores per executor ≈ 3.5. Rounded to **3 executors**.
* **Total Number of Executors:** 2 Workers × 3 executors/worker = **6 executors**
* **executor-memory:** The YARN limit is ~26 GB. After deducting overhead (≈ 26 GB ÷ 1.15), the available memory is about 20 GB. A more **conservative value of 8 GB** is retained.

**Spark-submit Command for THIN**

Here is the explanation of the choice of these parameters in the spark-submit command, generally used to optimize the execution of a Spark application on a YARN cluster (like DataProc):

**Basic Configuration (YARN/Deployment):**

* --master yarn: Specifies that the application should be executed on the YARN resource manager.
* --deploy-mode client: Indicates that the Spark Driver will be executed on a Worker Node of the cluster, not on the machine where the command is launched. This is the recommended mode for production applications.

**Resource Allocation (Executors and Driver):**

* --num-executors 6: Requests 6 execution processes (Executors) for the application. The choice of the number of executors depends on the cluster size and the volume of data.
* --executor-cores 2: Allocates 2 CPU cores to *each* Executor. A good compromise between parallelism and contention.
* --executor-memory 8G: Allocates 8 GB of RAM to *each* Executor. Memory is crucial for RDD cache, shuffle, and calculations.
* --driver-memory 12G: Allocates 12 GB of RAM to the Driver. The Driver manages the application, stores the state, and can collect results; it may require more memory if many collect() actions are performed or if the execution plan is very large.
* --driver-cores 4: Allocates 4 CPU cores to the Driver.

**Advanced Memory and Parallelism Configuration:**

* --conf spark.yarn.executor.memoryOverhead=2G: This is additional memory (off-heap) allocated by YARN for internal JVM needs, metadata, and native code, *in addition* to --executor-memory. **Total memory per Executor (according to YARN) = 8G + 2G = 10G**. This is crucial to avoid YARN container shutdown errors (Container killed by YARN for exceeding memory limits).
* --conf spark.memory.offHeap.enabled=false: Disables the use of off-heap memory by Spark itself (distinct from YARN overhead).
* --conf spark.sql.shuffle.partitions=100: Defines the number of partitions used during *shuffle* operations (grouping, join, etc.) by Spark SQL. 100 is a common value; if the data is very large, this number should be increased to ensure that each partition is not too large (a target partition size of 100-200 MB is often recommended).
* --conf spark.default.parallelism=100: Defines the number of tasks that will be created for RDDs during transformation operations. Keeping this number aligned with spark.sql.shuffle.partitions (or a multiple of the total available cores: 6 executors \* 2 cores/executor = 12 cores; 100 is a much higher value to ensure good task granularity).

**Application Specification:**

* --class com.flightdelay.app.FlightDelayPredictionApp: Indicates the main class (containing the main method) of your Spark application.
* --jars gs://bucket/jars/\*.jar: Includes additional Java/Scala libraries required for the application execution, stored on Google Cloud Storage (GCS).
* --files gs://bucket/config/prod-config.yml: Makes a configuration file available to the application, which will be accessible by the Driver and the Executors.

prod "data-pipeline,feature-extraction,train": These are the arguments passed to the main method of the specified class. Here, it is probably an environment profile (prod) and a list of *pipelines* to execute.

**Advantages:** High parallelism, fast GC

**Disadvantages:** Network overhead, limited shuffle buffers

**8.3 FAT Configuration (Few large executors)**

**Calculation for Dataproc**

executor-cores = 7 (large, leaves 1 core for OS)

Executors per worker = 1 (1 large executor per worker)

Total executors = 2 workers × 1 = 2 executors

executor-memory = 26 GB max - 5 GB overhead = 21 GB

**Spark-submit Command for FAT**

****spark-submit \

--master yarn \

--deploy-mode cluster \

--num-executors 2 \

--executor-cores 7 \

--executor-memory 21G \

--driver-memory 12G \

--driver-cores 4 \

--conf spark.yarn.executor.memoryOverhead=5G \

--conf spark.memory.offHeap.enabled=true \

--conf spark.memory.offHeap.size=4G \

--conf spark.sql.shuffle.partitions=50 \

--conf spark.default.parallelism=50 \

--class com.flightdelay.app.FlightDelayPredictionApp \

--jars gs://bucket/jars/\*.jar \

--files gs://bucket/config/prod-config.yml \

prod "data-pipeline,feature-extraction,train"

**Advantages:** Minimal overhead, large shuffle buffers

**Disadvantages:** Slow GC, OOM risk, under-parallelism

**8.4 Optimized Configuration (Balance)**

**Calculation for Dataproc (Recommended)**

executor-cores = 4 (balanced)

Executors per worker = (8-1) ÷ 4 = 1.75 → 1-2 executors

Total executors = 2 workers × 2 = 4 executors (estimated)

executor-memory = 12 GB (safe for YARN)

**Spark-submit Command for Optimized**

****spark-submit \

--master yarn \

--deploy-mode cluster \

--num-executors 4 \

--executor-cores 4 \

--executor-memory 12G \

--driver-memory ...

