Integration of the Flight Delay Prediction Application on the LAMSADE Cluster

# 1. Introduction

This report presents the complete integration of the Flight Delay Prediction application on the Hadoop/Spark cluster of Paris-Dauphine University (LAMSADE). The objective is to demonstrate how we adapted a locally developed Spark application to run efficiently on a shared cluster, covering all aspects of the DevOps cycle: development, integration, deployment, and usage.

The main interest in automating our deployment cycle (CI/CD) in this Data Science/Machine Learning context is manifold:

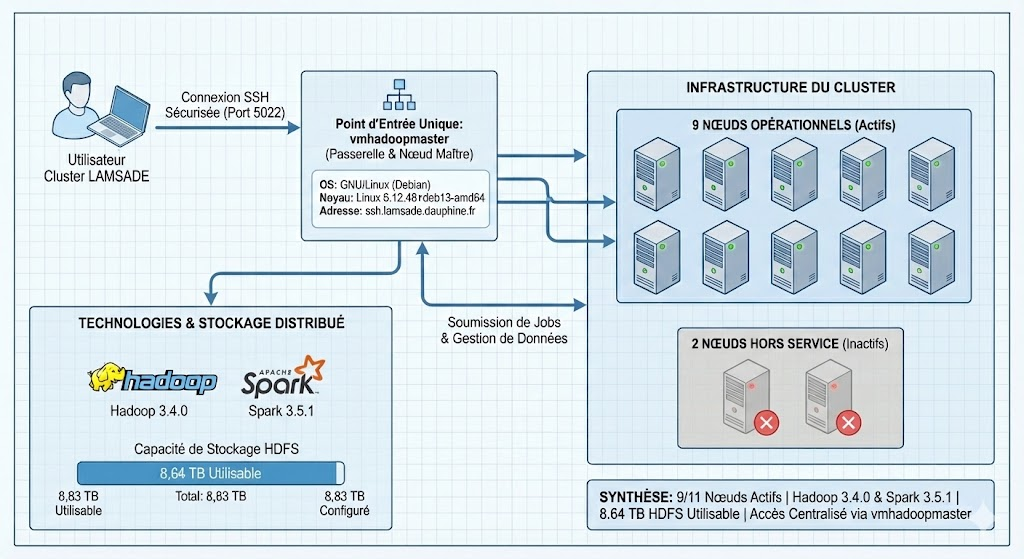
1. **Focus on Added Value (Analysis):** By eliminating manual and repetitive tasks of configuring and submitting Spark jobs, we can dedicate ourselves entirely to analyzing our results, improving our models, and designing our experiments, which is our core business.
2. **Scientific Rigor and Comparability:** Automation ensures that every ML experiment is executed in an environment and with a configuration that is strictly identical (reproducibility). This is fundamental for reliably comparing the performance of different models or hyperparameters we test.
3. **Reliability and Error Reduction:** Eliminating manual interventions drastically minimizes our risk of human errors (wrong dependency version, cluster configuration error, etc.), making our process more robust.
4. **Efficiency and Resource Savings:** Our approach allows us to optimize cluster usage (dynamic release or adjustment of resources after each execution), an essential point for our costly and shared environments.

In summary, for us, it's about **moving from a craft and time-consuming mode to an industrial, reproducible, and optimized process**, thereby accelerating our cycle of innovation and experimentation in Machine Learning.

# 2. LAMSADE Cluster Context

## 2.1 LAMSADE Infrastructure

The LAMSADE cluster constitutes a shared computing environment, whose technical configuration is detailed below:



**Cluster Summary:**

* **Operational Nodes:** Nine (9) nodes are currently active, out of a total of eleven (11) configured nodes (two nodes being temporarily out of service).
* **Technologies Implemented:** The cluster is based on the following versions of data processing frameworks:
  + Hadoop 3.4.0
  + Spark 3.5.1
* **HDFS Storage Capacity:** Usable storage space amounts to 8.64 Terabytes (TB), out of a total configured capacity of 8.83 TB.

**Access Methods and Architecture:**

Access to the LAMSADE cluster is centralized and secured, being exclusively established via the **SSH** protocol.

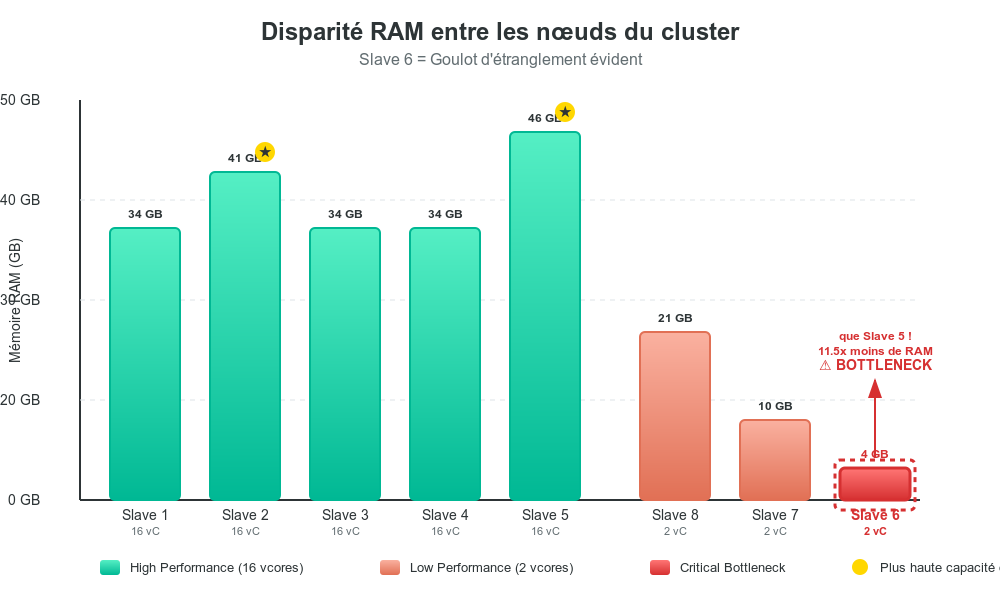
* **Single Entry Point:** The initial connection must be directed to the network address ssh.lamsade.dauphine.fr using port 5022.
* **Gateway and Master Node:** All computing and storage resources are accessible via the vmhadoopmaster machine.
  + This machine simultaneously performs the functions of **master node (in client mode deployment)** and **access gateway**.
  + It operates under a **GNU/Linux operating system (Debian distribution)**, as evidenced by the kernel version: Linux vmhadoopmaster 6.12.48+deb13-amd64. This technological foundation is known to guarantee robustness and stability within distributed computing environments.

**Usage for Researchers/Students:**

Once we have established the SSH session to **vmhadoopmaster**, we have the ability to interact with cluster services, submit our jobs (via available frameworks such as Hadoop and Spark), and manage our data, while benefiting from the security and authentication framework offered by the SSH connection.

## 2.2 Available Resources

Inspection reveals the following overall characteristics for the cluster. It is composed of **heterogeneous nodes**, detailed below:

****

**Heterogeneity Management and "Least Common Denominator"**

**Spark Task Optimization: Neutralizing Slow Nodes**

To ensure the stability and speed of our data processing (Spark tasks), we had to manage a problem: the overall performance of the cluster was limited by its weakest element (the "Least Common Denominator"), in this case, servers like vmhadoopslave6.

**The Problem: The Weak Link**

This server is a major bottleneck because it has very few resources: 2 cores and less than 4 GB of random access memory (RAM).

| **Slow Server** | **RAM (GB)** | **vCores** |
| --- | --- | --- |
| vmhadoopslave6 (example) | < 4 | 2 |

**The Technical Dilemma**

If we included this slow server in the general execution pool, we would be forced to significantly reduce the amount of resources allocated to each worker process (executor) so that the system could uniformly schedule tasks on *all* nodes (about 2 GB of RAM and 1 vCore per executor).

**The Negative Consequence**

This reduction would throttle the powerful servers. For example, on a 46 GB RAM node, about 90% of the memory would be unused, severely penalizing the general performance of the processing.

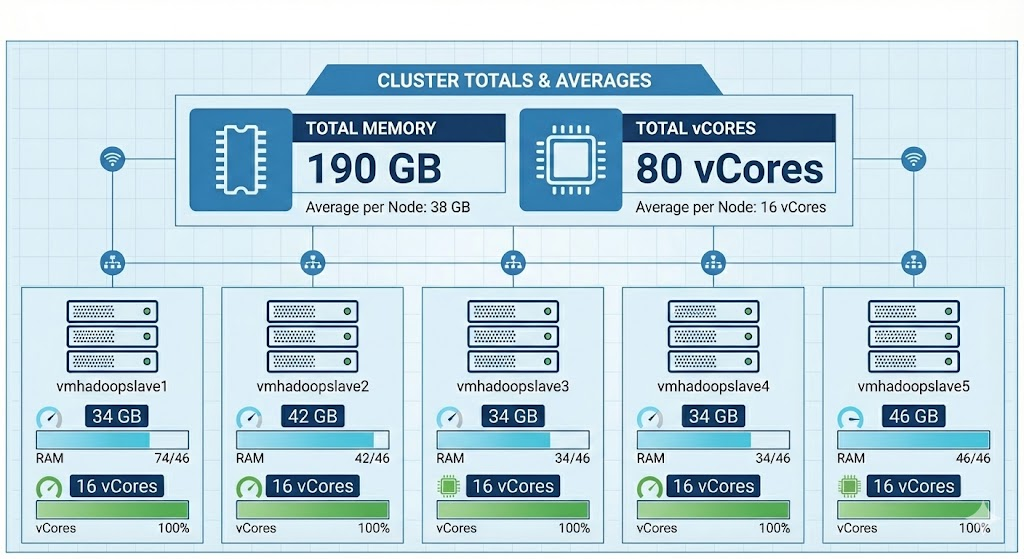
**Our Solution: Prioritizing Speed**

We chose to maximize performance by **voluntarily excluding this pool of slow servers** (which represents about 11% of the total processing capacity).

The strategy implemented involved increasing the memory dedicated to each worker process on the most powerful nodes, escalating it from 2 GB to 8 GB—a fourfold increase. While this means some servers are unused, it results in significantly faster and more stable execution for our memory-intensive Big Data application.

**Conclusion**

The cluster consists of five heterogeneous nodes. To maximize computational efficiency and optimize resource allocation, high-intensity tasks are strategically processed on the **five most performant nodes**. This targeted approach leverages the full capacity of the highest-specification nodes. Collectively, our five-node cluster (illustrated below) currently offers a **Total Memory of 190 GB** and **80 vCores**.



## 2.3 Specific Constraints

Here are the constraints of the environment:

* **Limited and Shared Resources:** The computing environment has restricted resources that are shared among users.
* **Restricted Privileges:** Users do not have administrative rights; therefore, operations must utilize user-specific paths.
* **Remote Access:** Connection to the environment is established via an SSH tunnel.
* **Storage Limitations:** HDFS has user-specific quotas, limiting available storage.

# 3. System Architecture

The LAMSADE laboratory manages a strategic, on-premise high-performance computing (HPC) cluster located on our campus. This infrastructure, overseen by laboratory teams, guarantees us privileged access to intensive computing resources.

The major advantages of this internal hosting model are:

1. **Total Control**: Complete mastery for performance optimization adapted to specific needs.
2. **Security/Confidentiality**: Sensitive data remains secured within the environment.
3. **Low Latency**: Very fast internal network connection, crucial for large volumes of data.
4. **Adaptation**: Ability to evolve the cluster (hardware, storage) based on projects and funding.

This cluster provides LAMSADE with a robust, secure, and customized computing platform. It is a critical component for supporting scientific innovation within the laboratory.

## 3.1 Production Architecture

The LAMSADE production architecture is structured as follows:

### 

The "Emiasd-FlightProject" solution operates on the LAMSADE Hadoop/Spark cluster. This technical architecture diagram illustrates the project's operational environment.

In summary: The remote cluster is accessed securely from a **local workstation** using an **SSH tunnel**. This setup facilitates the execution of distributed calculations (Spark/MLlib) on the cluster. The results, including metrics and models, are automatically logged to an **MLflow server** located on the master node, enabling us to view them through a web interface on their workstation.

### 3.2.1 Environment Preparation on vmhadoopmaster

Unlike platforms such as GCP, Lamsade lacks a native storage service. Consequently, a separate solution must be configured to support the architecture, handling elements like JAR files, data, and logs.

* **Remote Workspace Tree**

The primary working directory for the project is located at /Emiasd-Flight-projet within the user's path on our workspace. This directory is accessible from the master node (vmhadoopmaster) and is designed to clearly separate the various application components.

| **Directory** | **Objective** | **Main Content** |
| --- | --- | --- |
| ~/Emiasd-Flight-projet/apps/ | Executable Storage | Compiled application JAR, MLflow JARs (mlflow-client, mlflow-spark). |
| ~/Emiasd-Flight-projet/config/ | Configuration Files | YAML configuration file (prodlamsade-config.yml) for Spark execution. |
| ~/Emiasd-Flight-projet/data/ | Local Raw and Intermediate Data | Temporary data files before HDFS upload, upload/cleanup scripts. |
| ~/Emiasd-Flight-projet/logs/ | Execution Logs | Detailed logs of each Spark *run*, symbolic links (latest.log). |
| ~/Emiasd-Flight-projet/scripts/ | Automation Scripts | Bash scripts, utility Python scripts. |
| ~/Emiasd-Flight-projet/.venv/ | Isolated Python Environment | Virtual environment containing all necessary Python dependencies (MLflow, PySpark, scikit-learn). |

* **Isolated Python Environment Preparation (venv)**

Given the shared environment and the lack of administrator rights, using a Python virtual environment (venv) is essential to manage dependencies in isolation.

The installation of the production architecture on the LAMSADE cluster focuses on creating a structured working tree and setting up an isolated Python environment, ensuring reproducibility and project organization.

The following steps detail the installation of the necessary environment, performed by SSH connection on the vmhadoopmaster virtual machine:

1. **Creation of the virtual environment (venv)**:
   * Execute the command to create the virtual environment.
2. **Activation of the virtual environment**:
   * Activate the created virtual environment to isolate project dependencies.
3. **Installation of dependencies (libraries)**:
   * Once the environment is activated, install the required libraries for executing Spark tasks (PySpark), tracking experiments (MLflow), as well as for data analysis and visualization.

Key Note: The activation of this venv, performed after its installation, was maintained throughout the project.

* **HDFS Storage Paths**

In addition to the local working tree, a dedicated storage space is necessary for massive data on the Hadoop Distributed File System (HDFS).

The required HDFS paths are:

* **Raw Data**: /students/p6emiasd2025/$USER/data/  
  Downloading data to these HDFS locations is done via option 5 of the LAMSADE deployment scripts (Production) and should only be done once.
* **Results**: /students/p6emiasd2025/$USER/output/

The output directory contains the following subdirectories:

| **Directory** | **Description** | **Typical Content** |
| --- | --- | --- |
| **Experience-lamsade-1h** | **MLflow/Tracking**: Data and artifacts of a *run* | Models (RandomForest), metrics (F1, AUC), PNG graphs. |
| **common** | **Intermediate/Shared Outputs**: Transformed data, pre-calculated *features*, reusable aggregated results. | *Feature-engineered* datasets, intermediate tables. |
| **spark-checkpoints** | **Spark Checkpointing**: Ensures fault tolerance and optimizes complex calculations (ML, DAG), allowing recovery after failure. | Directories/metadata generated by SparkContext.setCheckpointDir(). |

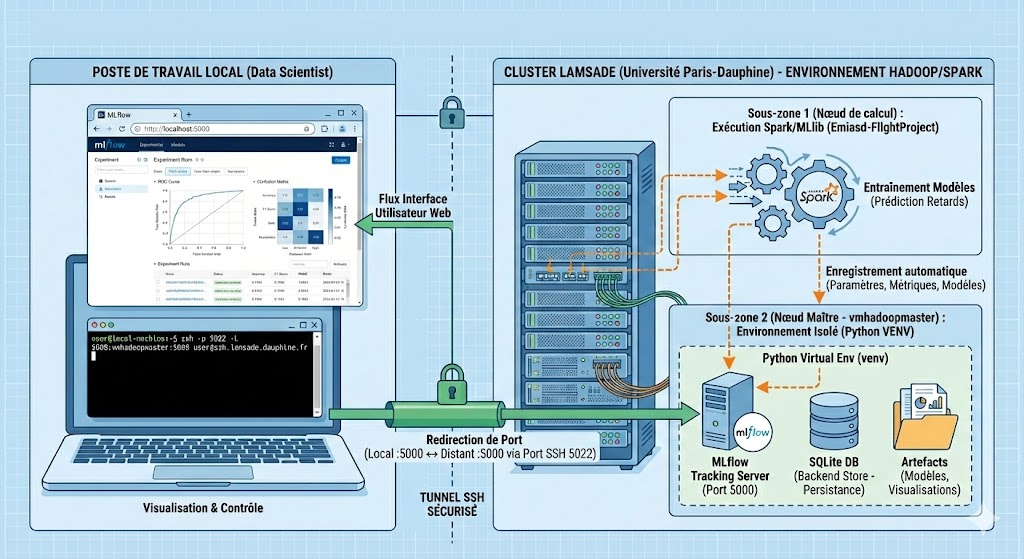
* **Setting up MLflow**

The Emiasd-FlightProject (flight delay prediction) relies critically on integrating MLflow with the LAMSADE cluster. This integration is vital for ensuring the traceability, reproducibility, and strict monitoring of ML experiments, including the logging of hyperparameters, metrics, models, and artifacts.

A prerequisite for this deployment is obtaining the necessary JAR files: **mlflow-client-3.4.0.jar** and **mlflow-spark\_2.13-3.4.0.jar**. These files are transmitted to the cluster using option 4 of the LAMSADE deployment scripts.

* **Architecture and Security**

It is installed in the isolated environment. Access to the MLflow interface is secured by an SSH tunnel, redirecting port 5000 of the LAMSADE server to local port 5000, ensuring confidentiality and integrity.



**Technical Configuration**

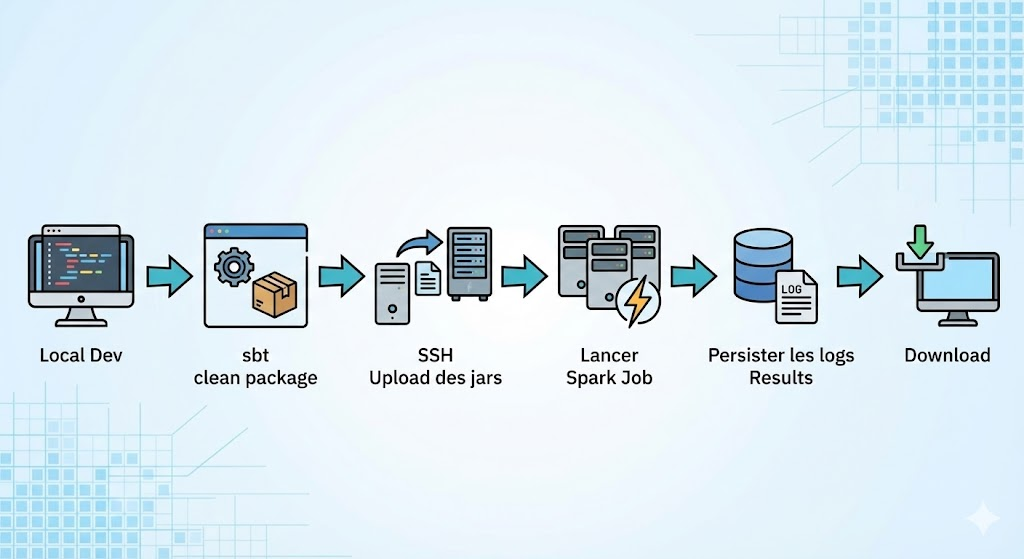
1. **Isolated Environment (venv) and Dependencies:** Installation of mlflow, visualization libraries (matplotlib, seaborn) to generate critical artifacts (ROC curves, confusion matrices), and project-specific libraries (pyspark, scikit-learn).
2. **Starting the MLflow Server:** The server is launched to persist data. The --backend-store-uri sqlite:///mlflow1.db option ensures the storage of metadata (runs, metrics, artifact paths). The server listens on port 5000 (--host 0.0.0.0) and automatically records Spark/MLlib experiments.
3. **SSH Tunnel for Secure Remote Access:** A *Port Forwarding* mechanism via SSH (-L 5000:vmhadoopmaster:5000) is used with private key authentication (-i /hbalamou.key) to access the interface without public exposure.

**Exploitation**

The MLflow interface is accessible locally via **http://localhost:5000**. It allows viewing the complete history of experiments, examining the details of the *runs* (e.g., RandomForest, Gradient Boosting), comparing model performances (F1-score, AUC), and downloading models and visualizations (artifacts)

# 3. Data Flow

The following image is a flow diagram illustrating the key steps in the life cycle of our Spark application, from development to result retrieval.Simplified process description:



1. **Local Dev**: Initial application development on the local machine.
2. **sbt clean package**: Compilation and packaging of the application (JAR creation) via sbt.
3. **SSH Upload of jars**: Secure deployment of JARs to the remote environment (server/cluster) via SSH.
4. **Launch Spark Job**: Execution of distributed data processing on the Spark cluster.
5. **Persist logs Results**: Saving of processing results and execution logs in a database.
6. **Download**: Retrieval of results or logs by the user on their local machine.

This workflow summarizes the typical cycle of development, deployment, and execution of Big Data applications with Spark.

# 4. Code and Scripts Development

## 4.1 Scala Code Structure

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

## 4.2 YAML Configuration

* The YAML file is the central element of the configuration for the flight delay prediction application. It brings together all the parameters necessary for the execution of the machine learning pipeline in the production environment, in this case, the LAMSADE cluster.
* **Key Principle:** This YAML file allows modifying experiments without needing to touch the code; a simple update of the file is sufficient.

## 4.3 Deployment Scripts

Two levels of automation developed:

* **Bash Scripts**: lamsade\_fulldeployement.sh (interactive menu)
* **GitHub Actions**: Automated CI/CD

# 5. Automated Deployment

## 5.1 Interactive Bash Script (lamsade\_fulldeployement.sh)

Executing the script opens a complete interactive menu, offering the following seven options:

1. **Full deployment**: Compile + deploy + run
2. **Compile only**: SBT clean package
3. **Deploy JAR**: Upload to cluster
4. **Deploy config**: Upload configuration
5. **Upload data to HDFS**: Data to HDFS
6. **Run on cluster**: Spark execution
7. **Download logs**: Log retrieval

In practice, options 2, 3, and 4 are the most frequently used, generally in combination with the *submit* execution on the server.

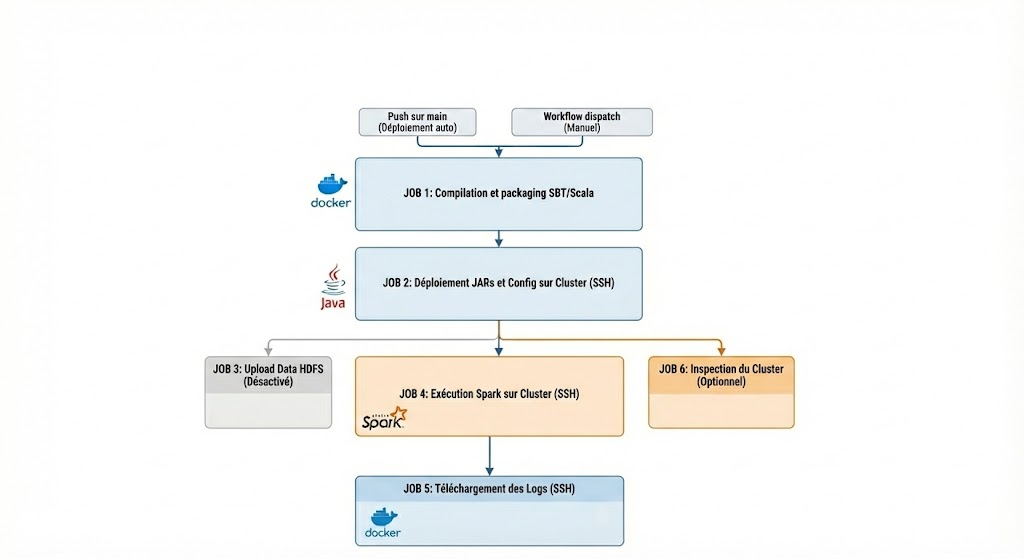
## 5.2 GitHub Actions CI/CD

The integration of **GitHub Actions** has been adopted for several major benefits:

* **Simplified Integration and Deployment (CI/CD):**The solution is **natively integrated with GitHub**, which greatly simplifies the setup of the continuous integration and deployment (CI/CD) chain. This eliminates the need for external configurations for triggers and artifacts.
* **Centralized and Versioned Workflow:**The workflow is defined in YAML files, versioned directly with the source code. This approach ensures that automation is always synchronized with the current state of the repository.
* **Reliability and Isolated Environment:**\*\*Task execution is ensured in a **dedicated, clean, and reliable environment** (the GitHub runners). This minimizes potential errors related to the user's working environment configuration.
* **Traceability and Detailed History:** Each execution is automatically **logged**, thus providing complete **traceability** (execution logs, status) far superior to using a simple script.

This diagram illustrates the workflow of an automated CI/CD pipeline for an Apache Spark application:

****



The pipeline is initiated automatically by a *Push on Main* (code update) or manually via *Workflow Dispatch*. It then proceeds sequentially:

1. **Build & Package (Job 1):** Compilation and packaging of the Scala/SBT code in a Docker container.
2. **Deployment (Job 2):** Transfer of artifacts (JARs and configuration files) to the remote cluster via SSH.
3. **Execution (Job 4):** Launch of the main Spark job on the cluster.
4. **Logs (Job 5):** Retrieval of execution logs once processing is complete.

# 6. Usage and Monitoring

## 6.1 Launching the Deployment

The complete deployment of the application on Lamsade can be carried out in several ways, offering flexibility and different levels of automation and control.

**1. Execution via Local Interactive Script**

The most direct approach, often preferred for local development or testing, or when the operator wants direct control over the process, is the use of a dedicated automation script.

* **Execution command:** ./work/scripts/lamsade\_fulldeployement.sh

**2. Automation via GitHub Actions (CI/CD)**

For robust, reliable, and reproducible continuous integration and deployment (CI/CD), the use of a cloud-based or server-based automation platform is essential.

* **Technology:** GitHub Actions.
* **Trigger Method:** Via the web interface with workflow\_dispatch (manual triggering) or automatically during a push to the configured branch, depending on the desired action.
* **Description:**
  + **Dedicated Workflow:** A specific workflow file is configured in the .github/workflows/ directory. This workflow encapsulates the steps of the complete "Lamsade" deployment, similar to or more sophisticated than those of the local script.
  + **Manual Triggering (workflow\_dispatch):** The use of workflow\_dispatch allows a user with appropriate rights to **manually trigger** the execution of the workflow directly from the GitHub web interface (under the "Actions" tab).

## 6.2 Real-Time Monitoring

Real-time monitoring of a Spark application submitted to the LAMSADE cluster via spark-submit is performed using the **YARN ResourceManager** web interface (port 8088). YARN is essential for managing resource allocation (CPU, memory) and for tracking the status of jobs, containers, logs, and overall cluster metrics.

To access it securely from a local workstation, an **SSH tunnel** is necessary, mapping local port 8088 to vmhadoopmaster's port 8088.

**Access:**

Once the tunnel is established, the interface is accessible via http://localhost:8088. It provides the status of Spark applications (e.g., flight prediction): list of jobs, resource usage, detailed container logs (debugging), and performance metrics. This monitoring complements the MLflow tool by offering a system view of the cluster.

* **YARN UI**:<http://vmhadoopmaster:8088>
* Spark UI ;
* **Real-time logs**: locally on the cluster via tail -F ~/workspace/logs/latest.log

## 6.3 Result Retrieval

**Methods for Downloading Logs**

| **Method** | **Command/Instructions** | **Notes** |
| --- | --- | --- |
| **Via Script** | ./work/scripts/lamsade\_fulldeployement.sh (Option 7: Download logs) | Uses a local full deployment script. |
| **Via Ansible** | ansible-playbook download-logs.yml | Executes an Ansible playbook specifically for log downloading. |
| **Via Direct SCP** | scp -P 5022 user@ssh.lamsade.dauphine.fr:~/workspace/logs/\*.log ./logs/ | Direct secure copy, requiring the specified port and user credentials to retrieve all .log files to the local ./logs/ directory. |

## 6.4 Application Management

Application management is accessible via the command line. Below is the list of available commands.

* **Monitoring Commands**

| **Command** | **Description** |
| --- | --- |
| yarn top | Displays real-time resources (YARN equivalent of htop) |
| yarn node -list | Lists cluster nodes |
| yarn node -list -showDetails | Lists nodes with details (memory, vcores) |
| yarn node -list -all | Lists all nodes (including inactive ones) |
| yarn node -status <node\_id> | Displays the detailed status of a specific node |

* **Application Management**

| **Command** | **Description** |
| --- | --- |
| yarn application -list | Lists currently running applications |
| yarn application -list -appStates RUNNING | Lists only RUNNING applications |
| yarn application -status <application\_id> | Displays the status of an application |
| yarn application -kill <application\_id> | Stops/kills an application |

* **Log Consultation**

| **Command** | **Description** |
| --- | --- |
| yarn logs -applicationId <application\_id> | Retrieves complete logs of an application |
| yarn logs -applicationId <app\_id> > app\_logs.txt | Saves logs to a file |

* **\*\* View shuffle metrics \*\***

| **Command** | **Description** |
| --- | --- |
| yarn logs -applicationId <application\_id> > app\_logs.txt | Retrieves complete application logs and saves them to a local file for analysis. |
| `grep -i "shuffle" app\_logs.txt | head -30` |  |
| grep -i "shuffle.\*bytes|shuffle.\*records" app\_logs.txt | Specifically searches for detailed *shuffle* metrics (number of bytes and records transferred), essential for network I/O optimization. |

| **yarn logs command** | **Description** |
| --- | --- |
| yarn logs -applicationId application\_xxx | grep -i "error|exception" | Search for errors |
| yarn logs -applicationId application\_xxx | grep -i "OutOfMemory" | Check for OutOfMemory |
| yarn logs -applicationId application\_xxx | grep -i "failed" | View failed tasks |

* **\*\* Advanced Log Consultation \*\***

| **Command** | **Description** |
| --- | --- |
| yarn logs -applicationId application\_1234567890123\_0001 | grep -i "shuffle" | Extract shuffle metrics |
| yarn logs -applicationId application\_1234567890123\_0001 | grep -E "Shuffle Read|Shuffle Write" | View shuffle read and write |

# 7. Spark Configurations for LAMSADE

## 7.1 Comparison of Spark Configuration Architectures: Compact (Thin), Voluminous (Fat), and Optimized

The efficiency, robustness, and optimal resource utilization of a Spark environment are closely dependent on rigorous task configuration. This analysis compares three distinct configuration approaches:

* Compact (Thin)
* Voluminous (Fat)
* Optimized

to evaluate their impact on overall performance.The main objective of this study is to demonstrate the relevance of choosing the **OPTIMIZED** configuration, identified as the most performing and stable, for the LAMSADE production environment.The Three Configuration Strategies

1. **Compact (Thin):** Favors maximum parallelism (multiplication of small *executors*), but leads to a degradation of network efficiency and Input/Output (I/O) operations.
2. **Voluminous (Fat):** Aims to reduce network *overhead*, but introduces critical issues related to *Garbage Collection* (GC) and system instability.
3. **Optimized (Optimized):** Relies on industry standards and best practices (Cloudera/Databricks) to establish an ideal compromise between I/O efficiency, memory management, and network performance.

* Detailed Analysis of Configurations on the LAMSADE Cluster

The experimentation relies on the five most performing nodes of the LAMSADE cluster. To ensure uniformity of results, the allocated resources are aligned with those of the most constrained node: **34 GB of RAM and 16 vCores (i.e., 33 GB of memory and 15 vCores available after deducting operating system resources).**

Here are the four key rules for cluster resource sizing and management, aimed at performance/stability optimization:

1. **Reserve the essential (Guaranteed Stability):** Systematically allocate **1 GB of RAM and 1 CPU core** per node for the OS and YARN to prevent saturation and ensure stability.
2. **Size uniformly (Global Allocation):** Adapt the size of executors (memory and cores) to the resources of the **weakest node** to guarantee container allocation on all nodes by YARN.
3. **Optimize cores (Equilibrium Point):** Use **5 cores per executor** to maximize HDFS throughput while controlling I/O contention and Garbage Collection.
4. **Control memory (Overshoot Prevention):** Ensure that the sum of *heap* and *overhead* of executors **does not exceed the available RAM** of the most limited node, even if a slight overshoot is sometimes tolerated.

1️⃣ Compact Configuration (Thin): Extreme Parallelism

| **Parameter** | **Value** | **Justification** |
| --- | --- | --- |
| **Executors** | 35 | 7 *executors* per node x 5 nodes |
| **Cores/exec** | 2 | Minimum operational requirement |
| **Memory/exec** | 5G | Derived from the 33 GB available per node (including 1 GB overhead) |
| **Total YARN/exec** | 6G | 5 GB (Memory) + 1 GB (Overhead) |

**Evaluation:** Although parallelism reaches its maximum (70 simultaneous tasks), the observed instability is critical. The **network overhead is prohibitive** (with a potential for 1,225 *shuffle* connections), and the low number of cores per *executor* (2) induces **low HDFS I/O throughput** (100 MB/s). This approach is deemed unviable for data-intensive processing (machine learning).

2️⃣ Voluminous Configuration (Fat): Minimizing Network Overhead

| **Parameter** | **Value** | **Justification** |
| --- | --- | --- |
| **Executors** | 5 | 1 *executor* per node x 5 nodes |
| **Cores/exec** | 15 | Use of all available cores per node |
| **Memory/exec** | 28G | Derived from the 33 GB available per node (including high overhead of 5 GB) |
| **Total YARN/exec** | 33G | 28 GB (Memory) + 5 GB (Overhead) |

**Evaluation:** Network overhead is minimized (only 25 *shuffle* connections), but the substantial size of the JVM (28 GB) causes **insoluble *Garbage Collection* (GC) problems**. GC pauses, lasting 2 to 10 seconds, cause *timeouts* and major instability. Moreover, the concentration of 15 cores on a single *executor* degrades HDFS I/O throughput. **Conclusion: Rejected** due to persistent instability related to GC.

3️⃣ Optimized Configuration (Optimized): The Balance (Recommended Practice) ⭐This methodology exploits the "sweet spot" of 5 cores per *executor*, a configuration recognized for offering the best HDFS I/O throughput while ensuring fast and efficient GC.

| **Parameter** | **Value** | **Justification** |
| --- | --- | --- |
| **Executors** | 15 | 3 *executors* per node x 5 nodes |
| **Cores/exec** | 5 | HDFS equilibrium point (350 MB/s) and efficient GC |
| **Memory/exec** | 10G | Derived from the 33 GB available per node (including 1.5 GB overhead) |
| **Total YARN/exec** | 11.5G | 10 GB (Memory) + 1.5 GB (Overhead) |

**Evaluation:** This configuration establishes an **optimal balance**.

* **Optimal HDFS Throughput**: The allocation of 5 cores per *executor* allows reaching 350 MB/s.
* **Efficient GC**: The 10 GB JVM ensures fast GC pauses (about 150 ms), guaranteeing system stability.
* **Balanced Parallelism**: Allows simultaneous execution of 75 tasks.

**Conclusion: Retained Configuration.** It maximizes I/O performance and memory stability, two essential factors for our data-intensive application.

Comparative Synthetic Table

| **Criterion** | **Compact (Thin)** | **Voluminous (Fat)** | **Optimized ⭐** |
| --- | --- | --- | --- |
| **Executors** | 35 | 5 | **15** |
| **Cores/exec** | 2 | 15 | **5** |
| **Memory/exec** | 5G | 28G | **10G** |
| **HDFS Throughput** | ⚠️ 100 MB/s | ❌ 200 MB/s | **✅ 350 MB/s** |
| **GC Pauses** | ✅ 50 ms | ❌ 2-10 s | **✅ 150 ms** |
| **Shuffle Connections** | ❌ 1,225 | ✅ 25 | **✅ 225** |
| **Stability** | ⚠️ Average | ⚠️ OOM/GC Risk | **✅ Excellent** |
| **Overall Performance** | ❌ Insufficient | ❌ Very Insufficient | **✅ Optimal** |

ConclusionThe *Compact* and *Voluminous* configurations were quickly ruled out due to the major bottlenecks they generate (network saturation and low I/O for the Compact; GC instability for the Voluminous).The **Optimized** strategy represents the most favorable pragmatic compromise, perfectly adapted to the heterogeneity constraints of the LAMSADE cluster. It guarantees maximum performance while maintaining critical stability, essential for massive data processing and machine learning operations.

## 7.2 Optimized Configuration

## 7.3 Parameter Justification

Driver Resources

* **32G RAM, 8 cores**: Sufficient for coordination without overloading
* **maxResultSize=8g**: Handling voluminous results

Executor Resources

* **8G RAM/executor**: Adapted to 34-46 GB nodes
* **6 executors**: 3 per node on the 5 main ones (avoids weak slave6-8)
* **No cores specified**: Leaves YARN to decide (generally 1-2 per executor)

Spark Optimizations

* **400 partitions**: Adapted to available parallelism
* **Kryo serializer**: Optimal performance
* **Off-heap 2G**: Native memory for shuffle
* **Extended timeouts**: Stability on shared network

**8.4 Comparison with Theoretical Configurations**

| **Aspect** | **Configuration Used** | **Optimal Theoretical** | **Justification** |
| --- | --- | --- | --- |
| **Executors** | 6 | 15 (5×3) | Avoids weak nodes (slave6-8) |
| **Executor Memory** | 8G | 10G | Conservative for stability |
| **Driver Memory** | 32G | 16G | Surplus for security |
| **Partitions** | 400 | 300 | Adapted to resources |

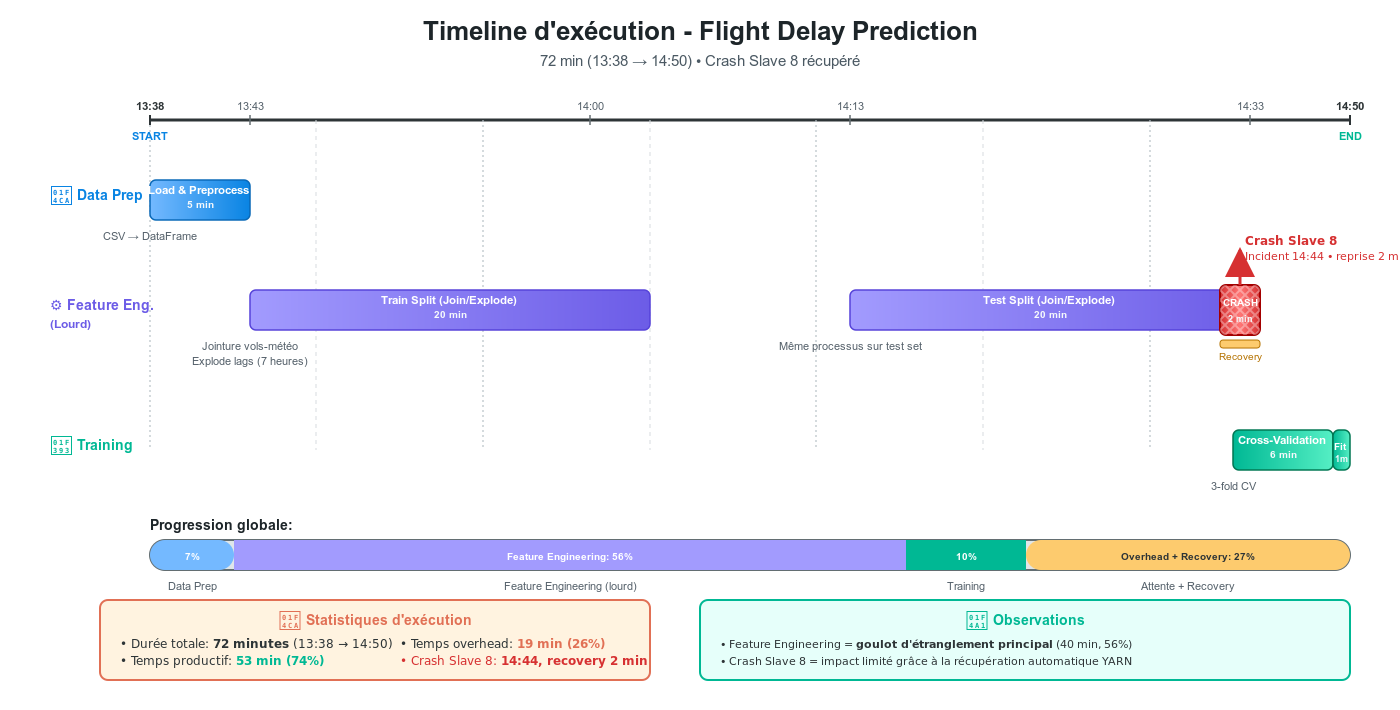
# 8. Results and Performance

## 8.1 Performance Metrics

* **Execution time**: 2-3 hours for complete pipeline
* **Cluster utilization**: 6 executors on 9 available nodes
* **Stability**: Successful executions without memory crash
* **Network**: Correct management of SSH latency

## 8.2 ML Metrics

* **Accuracy**: ~0.85 on cross-validation
* **Features**: 44 after PCA and feature engineering
* **Model**: Random Forest with optimized hyperparameters
* **Data**: 142,000+ flights processed



# 9. Conclusion and Perspectives

**Big Data Production Deployment on LAMSADE Cluster: A DevOps Success**

The integration of the project on the LAMSADE cluster has been a key factor, transforming the initiative into a production-ready Big Data application. The DevOps approach, characterized by multiple levels of automation (Bash, Ansible, GitHub Actions), ensures robust and fully reproducible deployment, even in a shared university environment.

## Major Technical Achievements

* **Complete Automation:** The continuous integration and deployment (CI/CD) chain is fully automated, from the initial *build* to the final deployment.
* **Resilience and Robustness:** The system integrates error and *timeout* management to ensure stability.
* **Efficient Supervision:** Implementation of a *monitoring* system with real-time logs and retrieval functionalities.
* **Resource Optimization:** The configuration has been specifically adjusted to maximize cluster resource utilization.

## Roadmap and Future Evolutions

* **Multi-cluster Extension:** Preparation for supporting heterogeneous environments (e.g., GCP Dataproc in addition to LAMSADE).
* **Container Orchestration:** Integration of Kubernetes for more advanced infrastructure management.
* **Extended CI/CD Pipeline:** Addition of automated tests directly on the cluster.
* **Advanced Monitoring:** Deployment of metrics with Prometheus and Grafana tools.

This project concretely illustrates the effectiveness of DevOps practices for the development and implementation of Big Data applications within research environments.