Scaling Machine Learning with Spark:

Land Cover Classification on FRACTAL Dataset

Course:

Big Data

Submitted by:
Bruna Candido
Ethel Ogallo

Date:
16/11/2024

Table of Contents

1. Introduction

In the field of Remote Sensing and Machine Learning, researchers are always dealing with a huge amount of data especially when working with land cover classification, a broad set of variables can be employed as input features to distinguish among soil types. Moreover, when talking about supervised machine learning algorithms, it is crucial to have a considerable amount of labelled data, so the model can properly learn how to classify it. The FRACTAL Dataset, published by (Gaydon et al., 2024) is an example of dataset developed for that. Therefore, as other big data datasets, it is not possible to use it entirely to train a machine learning model on a single regular computer.

With cloud computing, this process can be efficiently performed in the cloud, instead of running locally. However to do that, it is necessary to employ tools that have been developed for this purpose, such as Apache Spark, which is a large-scale analytics engine for data processing, PySpark, an interface for Apache Spark in Python, and MLflow, an open-source platform for managing the machine learning lifecycle (Polak, 2023). A processing platform is also necessary, such as Amazon Elastic MapReduce (AWS EMR). Additionally, the large dataset must be stored somewhere, and for that, a good choice is the Amazon Simple Storage Service (S3). With AWS EMR it allows us to separate storage from compute, enabling clusters to be created and terminated without losing data. It also provides scalable, durable, and low-cost storage that integrates natively with Spark.

Besides the challenge of running the codes itself, another challenge commonly faced by researchers is making their codes and research reproducible.

In our context, the purpose of this study was to:

- Choose, implement and evaluate a Machine Learning (ML) algorithm for land cover classification on the FRACTAL dataset.
- Process and prepare the FRACTAL dataset using PySpark or MLlib using the distributed computing technology available including S3 and Amazon EMR.
- Experiment with the scalability of the distributed computing pipeline by using different data samples, spark and cluster configurations.
- Ensure reproducibility through well documented code and providing all the specific parameter details required to reproduce the workflow.

2. Data and Tools

2.1 Dataset

FRACTAL: An Ultra-Large-Scale Aerial Lidar Dataset for 3D Semantic Segmentation of Diverse Landscapes(Gaydon et al., 2024) is a large-scale aerial Lidar dataset. It was designed for 3D segmentations of diverse landscapes for land monitoring. It's a dense point cloud comprehending 250km^2 in France. The dataset includes seven semantic soil classes: *ground, vegetation, building, water, bridge, permanent structure,* and *other*. The whole dataset size is about 180GB. The dataset was stored at S3 bucket in the parquet format, and it was divided in three different folders, i.e. train (80%), test (10%) and validation (10%).

2.2 Tools

Given the size of the dataset, the code had to be scaled so the model could be trained, since the data would not fit in a single machine's memory. Therefore, tools that make the scalability possible were used:

Apache Spark:
Apache Spark is an open-source distributed computing framework, being widely used for large-data processing. It provides in-memory computation, which significantly speeds up data analysis. Between other purposes, it supports the implementation of ML algorithms. It is implemented in different APIs, for Python, Java and Scala (Polak, 2023).

MLlib:
MLlib is a Machine Learning Library from Apache Spark. It is a facilitator for ML problems, designed to provide scalability and easy integration with other tools. It was developed for preprocessing, training models and making predictions on data (*MLlib | Apache Spark*, 2025).

AWS infrastructure:
The AWS infrastructure has several AWS services that support different processes for big data.

- **AWS management console** is a web-based application that contains and provides centralized access to all individual AWS service consoles (*Welcome to AWS Documentation*, 2025).
- **Amazon EC2 (Elastic Compute Cloud)** provides on-demand, scalable computing capabilities in the cloud (Aishwarya Anand & GD Goenka University, 2017, 2017; *Welcome to AWS Documentation*, 2025)
- **Amazon Simple Storage Service (S3)** is an object storage service that offers scalability, data availability, security, and performance (*Welcome to AWS Documentation*, 2025).
- **Amazon Elastic MapReduce (EMR)** is a managed cluster platform that simplifies running big data frameworks, such as Apache Hadoop and Apache Spark, on AWS to process and analyse vast amounts of data (*Welcome to AWS Documentation*, 2025)

3. Methodology

3.1 Overall Architecture

The overall architecture as shown in ***Figure 1*** is set up to enable execution of a large-scale machine learning workflow processed efficiently, scalable and parallelized.

The Amazon S3, the storage layer discussed in section 2, is where the FRACTAL dataset is stored as well as the processing, model and metric results stored.

The resource management layer is the core processing environment as it is responsible for scheduling the jobs for processing data (*Amazon EMR Architecture and Service Layers - Amazon EMR*, 2025). It includes the EMR cluster, the master node and worker nodes.

The master node hosts the Spark Driver, which is responsible for executing the SparkSession, maintaining the DAG scheduler and block manager. The driver works together with the cluster manager, i.e. Hadoop YARN (Yet Another Resource Negotiator), which has scheduling policies that govern how cluster resources are allocated among competing jobs (*Apache Hadoop 3.4.2 – Apache Hadoop YARN*, 2025). These policies impact job start times, resource contention, and overall cluster efficiency, influencing the performance patterns observed in our experiments on shared EMR clusters.

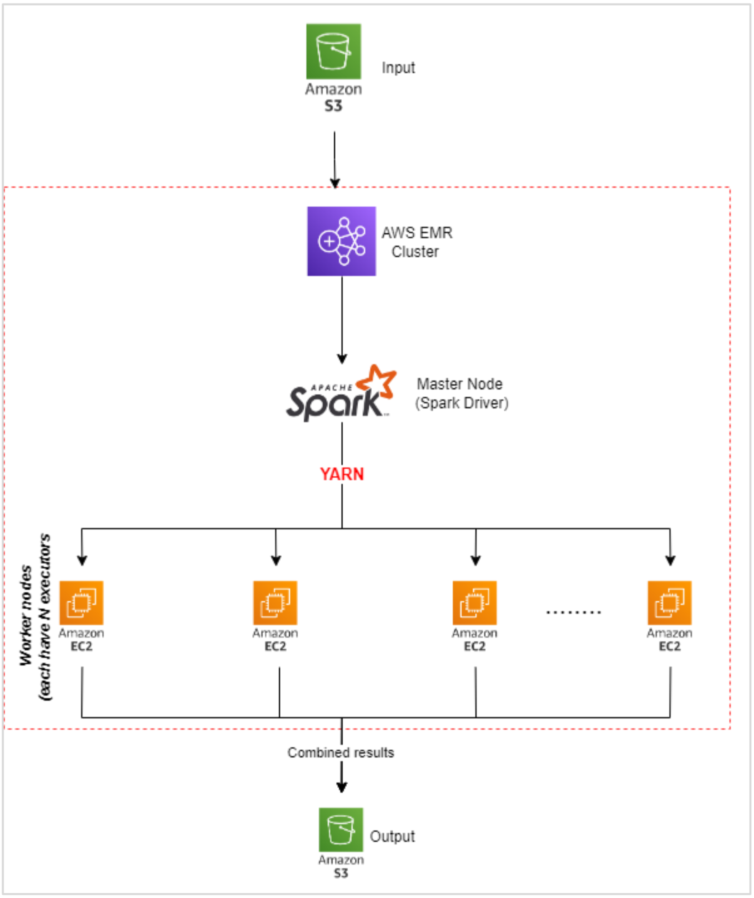

Figure 1 overall architecture of the cloud distributed system utilized

The worker nodes run one or more Spark Executors which executes the computational tasks in parallel and hold the in-memory data partitions used during processing (Polak, 2023). Spark ensures executors can recover from failures through its built-in fault-tolerance mechanisms, allowing tasks to be retried on other nodes and ensuring uninterrupted execution on the EMR cluster. (Polak, 2023; *RDD Programming Guide - Spark 4.0.1 Documentation*, 2025)

3.2 Spark Machine Learning Pipeline

The scalable Spark machine learning pipeline, as shown in Figure 2, was set up and implemented to support a multi-class land-cover classification task. The overall workflow consists of distributed data ingestion, preprocessing, feature engineering, and model training using Spark MLlib.

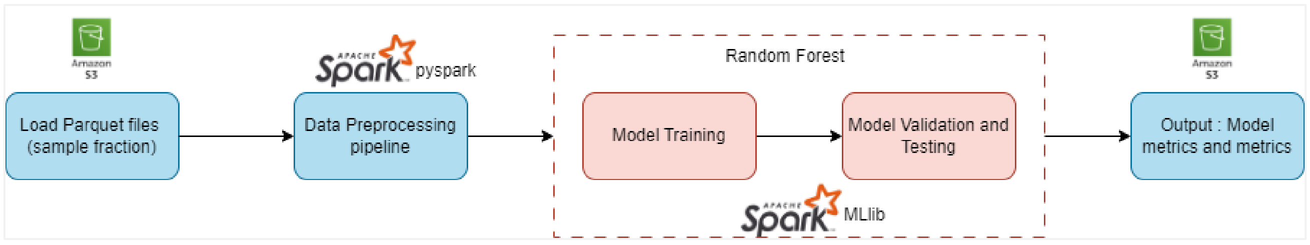
Amazon S3
Load Parquet files (sample fraction)
Spark pyspark
Data Preprocessing pipeline
Random Forest
Model Training
Model Validation and Testing
Spark MLlib
Amazon S3
Output : Model metrics and metrics

Figure 2 the spark machine learning pipeline workflow

We initially loaded the data stored in S3 in the parquet format for preprocessing and performed exploratory data analysis to understand it. The data was generally clean and therefore we did not check for missing data or outliers, we did however check for the distribution of the classification to inform the sampling process for the scaling experiment. To do this we first did label encoding which involved remapped the classification feature according to the semantic classes defined in (Gaydon et al., 2024) including the additional 'aesthetics' class to avoid adding nulls into the data. Table 1 summarizes the class distribution across the training, validation, and test sets:

Classification	Description	Train	Test	Val
1	Unclassified	0.56	0.67	0.53
2	Ground	38.97	40.49	39.1
3	Vegetation	56.98	54.09	56.93
4	Building	2.8	3.34	2.8
5	Water	0.52	1.2	0.49
6	Bridge	0.13	0.16	0.1
7	Permanent Structures	0.04	0.03	0.04
8	Filtered/Artifacts	0.01	0.03	0.01

Table 1 classification distribution in the full fractal dataset

The distribution shows that the data is well-balanced across classes, which informed efficient sampling strategies for scaling experiments and model choice and training process.

The full machine learning pipeline was then setup including data ingestion, feature engineering and feature selection. Initially we loaded the full dataset and then sampled a smaller fraction. This however led to slower execution time and I/O bottlenecks. We then implemented a file sampling method where we first list all the files in the directory and then randomly selects a sample fraction of the data (i.e. 1%, 3% and 5%). The selected sample files are then read into Spark DataFrames and then repartitioned to ensure parallelism is maintained as the data sample size increases.

Additional features such as z-norm (normalized height) and NDVI which would provide relevant information for land cover classification. This was done using spark pipeline and Transformer APIs to ensure reproducibility and scalability. For feature selection, the VectorAssembler transformer was applied to combine the specific selected features into a one vector column, because it is the required input format for MLlib.

The model classifier used in the pipeline was Random Forest, trained on the training dataset. We performed hyperparameter tuning on the number of trees and selected the value that achieved the highest validation accuracy. The final model was then refit with the new parameters on only the training set, rather than a combined train and validation set as is standard in ML workflows because of memory constraints and the size of the data. The final model then evaluated on the final test dataset giving the outputs of this workflow as the model, validation accuracy, best parameters and the test accuracy.

4. Technical choices

While conducting the scaling experiment, we had access to two EMR-based Spark cluster configurations: a 32-node cluster shared among 13 users, and an 8-node cluster shared between two users where both clusters used the same node types, consisting of one master node and worker nodes. The master node was an m5.4xlarge instance that has an Intel Xeon Platinum processor (8 physical cores / 16 vCPUs at 3.1 GHz) and 64 GB RAM. The worker nodes were r6i.2xlarge instances, each with 4 physical cores (8 vCPUs) at 3.1 GHz and 64 GB RAM.

The aggregated resources of each cluster are then as such:

- 8-node cluster (1 master + 7 workers)
  Total resources: 36 physical cores (72 vCPUs) and 512 GB RAM
  Effective for spark: 64 vCPUs and 460.8 GB RAM
- 32-node cluster (1 master + 31 workers):
  Total resources:132 physical cores (264 vCPUs) and 2,048 GB RAM
  Effective for spark: 248 vCPUs and 1,843.2 GB RAM

Because Spark, YARN, the operating system, and HDFS require their own overhead, each node must reserve at least 1 vCPU and approximately 10% of total memory for system processes. This significantly affects how we configure Spark executors.

The configurations below illustrate the specific trade-offs between CPU and memory allocation on each node, highlighting how these choices impact system performance and resource utilization:

- Using **64 executors** and **1 core per executor** would result in no vCPUs left for system processes and an **8GB executor memory** does not leave sufficient memory for YARN and HDFS.
- Configuring **8 executors** with **8 core per executor** also leaves no spare vCPUs for the OS, and **64GB executor memory** uses all available memory per node.

To balance CPU, memory, and system overhead, the configurations in Table 2 were selected as a baseline for all experiments.

Parameter Choice	**Justification**
Number of Executors	Selected as the primary variable to test scaling effects on runtime and parallelism.
Executor Cores	The number chosen to leave sufficient CPU capacity for YARN OS, and Spark overhead processes balancing task-level parallelism and resource overhead.
Executor Memory	Balanced to provide enough memory for processing without excessive spilling, while reserving memory for system-level operations.
Driver Memory	Allocated to support efficient DAG scheduling and job coordination without over-provisioning master node resources.

Data Partitioning Strategy	Designed to ensure even task distribution across the cluster i.e. avoid creating too many small partitions or too few large ones.

Table 2 shows the parameters chosen and rationale

5. Scaling Experiment

5.1 Experiment Setup

As mentioned in section 4, the experiments were conducted on two different cluster configurations. The first set of experiments used a 32-node cluster, while later experiments were repeated on a smaller 8-node cluster as that was the one available during scaling experiments. Both clusters used the same instance types described in Section 4 and used the same Spark configurations logic shown in Table 3.

Parameter Choice	**Value**
Data sample size	1%, 3% and 5%
Number of Executors	8, 16, 24, 30 and 32
Number of cores per executors	2
Executor Memory	8GB
Driver Memory	6GB
Data partitioning	Number of executors x number of executor cores x 4

Table 3 Spark configuration parameters

To investigate the impact of scaling, we varied only the number of executors, keeping the number of executor cores, driver and executor memory constant across all experiments. This was done because changing both the number of executors and the cores per executor would change several dimensions of parallelism at once, such as task concurrency, scheduling, memory pressure and CPU utilization, introducing multiple mixed effects. By fixing executor cores at 2, we ensured that even at the highest configuration (e.g., 30 executors × 2 cores = 60 vCPUs on the 8-node cluster), the cluster still had enough CPU capacity to handle YARN and OS processes. Maintaining memory per executor constant as well prevented memory pressure from skewing results. This design controlled for interactions between CPU and memory configurations, allowing us to directly attribute changes in runtime or efficiency to the level of parallelism and the number of executors. This approach provided a clearer understanding of how the system scales with increasing parallelism.

For model performance, we measured validation accuracy, test accuracy, and the effect of varying the number of trees in the Random Forest classifier. For execution analysis, we collected detailed Spark task metrics, including task duration, executor runtime and CPU time, scheduler delay, JVM garbage collection time, records and bytes read, shuffle bytes written/read, number of tasks, peak execution memory, and presence of spills (disk or memory). These metrics allowed us to assess both the quality of the model and the efficiency of the distributed processing. The measurements were collected directly from Spark task-level metrics, obtained through the task metrics API.

The experiments were submitted using spark-submit in both cluster and client deploy modes. Cluster mode runs the Spark driver within the YARN Application Master on the cluster, allowing the client to disconnect while the job continues running, making it preferred for production and long-running tasks. Client mode runs the driver on the client machine, suitable for debugging and interactive use but dependent on client connectivity (*Running Spark on*

YARN - Spark 4.0.1 Documentation, 2025). Majority of our experiments were done on the cluster mode.

5.2 Results and Discussion

Figure 3 illustrates the execution time curves for each sample fraction (1%, 3%, or 5%), with each line representing a different input size and execution time plotted against the number of executors. The figure shows a comparison of how execution time changes as the cluster parallelism increases. The graph shows that larger sample sizes increase execution time due to the need for more data processing.

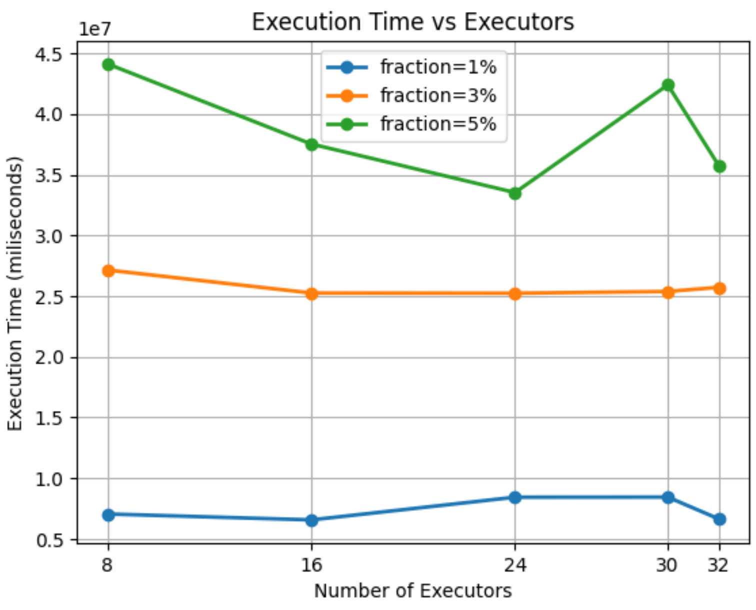

Figure 3 shows the execution time in seconds for number of executors given the different sample sizes

For the smallest sample size (1%), the execution time decreases initially as the number of executors increases, from 8 to 16 executors. The execution time then increases for 24 and remains constant up to 30 executors before decreasing again at 32 executors. This drop can be explained by cluster differences where the 30-executor experiment was performed on an 8-node cluster, while the others used the 32-node cluster as explained in the experiment setup. Compared to the 32-node cluster, the 8-node cluster had limited resources which may have caused higher overhead.

In the 3% sample fraction, execution time decreases from 8 to 16 executors and remains constant until 32 executors. For this sample, all experiments except the 30-executor experiment were done in the 32-node cluster, however we see that the parallelism is utilized well, and execution time stabilizes.

With the 5% sample fraction, execution time generally decreases with more executors, but a sharp increase is observed at 30 executors, again this is due to experiments being run on the smaller 8-node cluster. Experiments on the 32-node cluster show the expected decrease in execution time with increased parallelism.

In all experiments, the task count grew with both sample fraction and number of executors, as partitions were scaled according to the formula discussed in section 5.1. Increasing the number of tasks utilizes more of the cluster's CPU, but having too many tasks, especially on clusters with fewer resources, can result in additional overhead. This effect is most visible in configurations like the 30-executor experiments on the 8-node cluster.

The trend in execution time we observe in figure 3 can be explained by how Spark resource management with different configurations. Increasing the number of executors increases parallelism, but up to a point, which initially reduces execution time because more tasks can run in parallel. However, having too many tasks or too many executors for a small cluster can decrease the performance due to a high demand of CPU and memory resources leading to increased overhead.

To measure the scalability of the distributed system and the parallel efficiency, we compute speedup, defined as:

$$\boldsymbol{Speedup(n) = \frac{T_{base}}{T(n)}}$$

where T_{base} is the execution time of the baseline executors and n is the number of executors. In our case, the baseline corresponds to using 8 executors, since this was the smallest executor configuration used in the experiments.

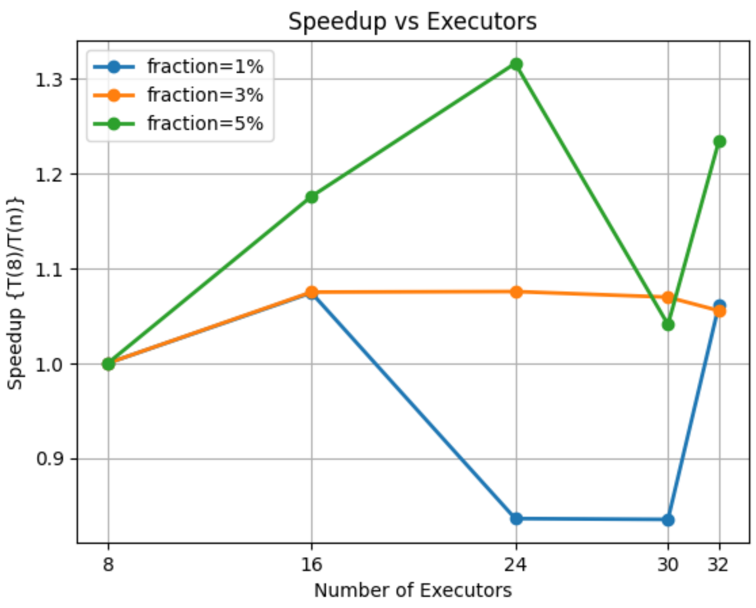

Figure 4 shows the speed up curves for each sample size

Figure 4 shows the speedup curves for each sample size where there is an initial increase in speedup as more executors are added in all the fraction sizes. The flattening or decrease in speedup for intermediate executor counts is seen on the 30-executor experiment for all sample sizes because they were computed on the 8-node cluster which can be due to limited resources as compared to the 32-node cluster. On the other hand, the 32-node cluster experiments and the 8-node cluster experiments with a smaller number of executors, maintain a nearly linear speedup curve indicating better parallelism with larger fractions achieving better utilization of cluster resources.

The execution time and speedup results show the importance of careful selection of executor count, cores per executor, and data partitioning in Spark to achieve efficient distributed processing and good parallel performance. The experiments make clear that Spark's performance rises or falls based on how well its settings fit the cluster resources. Adding more executors generally speeds up computations, but only if the cluster has enough nodes, memory, and CPU power to support the increased workload. The changes, especially with 30 executors on the smaller 8-node cluster, demonstrate the disadvantages of pushing a cluster beyond its limits. When resources are stretched too thin, scheduling delays, shuffle overhead, and memory issues slow everything down. On the other hand, larger clusters deliver steady performance gains when executor settings match available resources.

6. Challenges and Solutions

During the first sessions, one cluster was shared among all groups, which made the development slower. Because we had to wait for our colleagues' jobs to be finished given the cluster was only open during specific time slots per week, it was proving difficult to progress. One solution to mitigate this, was to submit jobs using cluster deploy mode, in which the driver runs on the master node instead of on the client. This improved runtime when the shared cluster was heavily loaded but also solved the issue of unstable internet connections because jobs continued running even if the client terminal disconnected, and all logs were stored directly on the cluster for later inspection. The other solution was during the last two sessions, each group had their own cluster, so we were able to perform all the runs we needed.

Another challenge faced was the memory availability. Because of the dataset was huge, we had to perform our training with less than 10% of the dataset. Because of this constrains, we decided to perform it using 1%, 3% and 5% of the total amount of training data.

One other challenge was that reading the dataset using spark.read.parquet () into a DataFrame and then sampling led to I/O overhead and longer execution time. To solve this, we instead list all the parquet files in the directory and then randomly sample the desired size, and this sample files is what is read into a DataFrame. This approach improved overall I/O time and reduced memory usage.

7. Conclusion

Overall, it is possible to say this project accomplished its purposes. A random forest model was implemented for land cover classification on the Fractal dataset. All the mandatory steps were followed, as using the S3 storage, as the AWS EMR, conducting the process using PySpark. The code delivered is reproducible and all the process can be run again without issues. In regarding of the different fractions of the data used, partitions of 1%, 3% and 5% of the total dataset were used in the model implementation, testing a different set of executors for each, 8, 16, 24, 30 and 32.

Through the execution time and speedup for each test, it was possible to see the importance of the selection of parameters, exemplified by the number of executors chosen. Different from what could be expected, the number of executors does not change the processing time linearly. The experiments showed that Spark's performance rises or falls based on how well its settings fit the cluster resources, and this highlights the importance of understanding how the cluster parameters work together, so a better performance with less resources can be achieved.

Besides that, the different versions of the cluster used showed how larger clusters deliver steady performance gains when executor settings match available resources.

As recommendations for future experiments, we would suggest training the data with a bigger amount of data and see how the parameters work. This is something we wanted to do, but due to time constrains we could not.

AI Disclaimer:
Artificial intelligence (AI) tools were used to improve the language, grammar, clarity, and structure of this report. However, they were all reviewed and edited by the authors before addition to final report. All ideas, analyses, and conclusions presented are originally of the authors.

8. References

Aishwarya Anand & GD Goenka University. (2017). Managing Infrastructure in Amazon using EC2, CloudWatch, EBS, IAM and CloudFront. *International Journal of Engineering Research And*, *V6*(03), IJERTV6IS030335. https://doi.org/10.17577/IJERTV6IS030335

Amazon EMR architecture and service layers—Amazon EMR. (2025, November 14). https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-overview-arch.html#emr-arch-resource-management

Apache Hadoop 3.4.2 – Apache Hadoop YARN. (Accessed 2025, November 15). https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html

Gaydon, C., Daab, M., & Roche, F. (2024). *FRACTAL: An Ultra-Large-Scale Aerial Lidar Dataset for 3D Semantic Segmentation of Diverse Landscapes* (No. arXiv:2405.04634). arXiv. https://doi.org/10.48550/arXiv.2405.04634

MLlib | Apache Spark. (Accessed 2025, November 15). https://spark.apache.org/mllib/

Polak, A. (2023). *Scaling machine learning with Spark: Distributed ML with MLlib, TensorFlow, and PyTorch*. O'Reilly.

RDD Programming Guide—Spark 4.0.1 Documentation. (Accessed 2025, November 16). https://spark.apache.org/docs/latest/rdd-programming-guide.html#fault-tolerance

Running Spark on YARN - Spark 4.0.1 Documentation. (Accessed 2025, November 16). https://spark.apache.org/docs/latest/running-on-yarn.html

Welcome to AWS Documentation. (Accessed 2025, November 15). https://docs.aws.amazon.com/