Reena Chandra[1]

Design and Implementation of Scalable Test Platforms for LLM Deployments

Abstract:
As the adoption of Large Language Models (LLMs) and machine learning (ML) accelerates across domains, there is a growing need for scalable, cost-efficient, and reproducible deployment frameworks. This study introduces a cloud-native benchmarking architecture that integrates Google Colab for rapid model development with Amazon Web Services (AWS) for deployment simulation. Four ML models, Random Forest, XGBoost, LightGBM, and Multi-Layer Perceptron (MLP)—are trained and evaluated using a tabular classification dataset, then aligned with suitable AWS services (Lambda, EC2, SageMaker) based on their computational and concurrency profiles. The models are assessed on classification metrics, latency, cold start behaviour, and cost per inference. Findings reveal that XGBoost is optimal for stateless, serverless deployment via AWS Lambda, while MLP is better suited for EC2 due to memory demands. LightGBM benefits from SageMaker's managed scalability. The framework demonstrates the viability of surrogate model benchmarking for LLM scenarios using lightweight ML models, and offers a reproducible, low-cost pipeline to support MLOps practices in cloud environments.

Keywords: Machine Learning Deployment; AWS Lambda; Amazon EC2; SageMaker; Google Colab; Cloud Benchmarking; XGBoost; Serverless Architecture; LLM Simulation; Python Automation; MLOps; CloudWatch Monitoring; Concurrency Testing; Model Inference Scalability

Received: 18 October, 2025 **Accepted:** 12 November, 2025 **Published:** 28 November, 2025

1. Introduction

Large language models (LLMs) have quickly taken over the field in recent times, helping create AI advances in speech processing, logic systems, and smart conversations with machines [1]. Such models, like GPT and BERT, have been able to do things that were not possible before; however, they have also led to new problems in large-scale deployment, managing expenses, and setting benchmarks [2]. A lot of research on LLMs looks into their model structure, how they are pre-trained, and how to improve them later, but it is just as important to evaluate and use them in real situations [3]. Simulation tools that can test an LLM with realistic patterns and constraints are necessary for moving such workloads from research settings to companies.

A key limitation in current benchmarking practices is their reliance on single-node, offline environments that fail to represent the complexities of cloud-native inference [4]. Most academic studies still evaluate ML model performance in local settings, focusing exclusively on training accuracy, loss convergence, or single-threaded inference latency [5]. These setups, while useful for model development, provide limited insight into how models behave under realistic deployment conditions, such as when invoked concurrently by thousands of users, exposed to memory constraints in serverless environments, or monitored for cost-efficiency across variable traffic loads. Moreover, many existing benchmarking frameworks overlook cloud orchestration, real-time metric collection, and comparative analysis of server-based versus serverless strategies. Consequently, organizations that adopt LLMs or deploy smaller surrogate models in production often rely on ad hoc testing methods, leading to inefficient infrastructure provisioning and unpredictable operational behavior [6].

To address these gaps, this paper proposes a new framework that utilizes Google Colab for model training and testing, and Amazon Web Services (AWS) for evaluating the deployment and scalability of various tasks. While Google Colab is a low-cost and accessible environment for ML in Python, AWS provides Lambda, EC2, SageMaker, S3, and CloudWatch to ensure high availability and easy scalability of inference tasks. With both

[1] Independent Researcher, USA
Email ID: reenachandra11@gmail.com
ORCID: 0009-0001-8061-1084

frameworks linked, it has a cloud-native tool for testing models, which simulate training, serialization, uploading to the cloud, and handling many requests simultaneously. Although this version includes a full deployment design, our process aligns with what is used in MLOps, helping us build toward automating everything and sharing models on the cloud in our next steps.

Central to this research is building and testing a reliable pipeline that measures the performance of Language Model (LM) workloads. To test the models, they are trained using a numerical dataset shared by Kaggle. The models selected are Random Forest, XGBoost, LightGBM, and MLP because they vary in terms of their ease of training and their suitability for different use cases. They act as benchmarking cases for cloud platforms. Training and testing are both carried out in Google Colab, and the deployment architecture suggests using lightweight AWS Lambda (for serverless inference), EC2 (for hosting Docker containers), and SageMaker (for managed endpoints) for deployment. Among our evaluation methods, the paper examines the model's accuracy, confusion matrices, and displays of class occurrences and relationships between features. The plan is to extend this approach theoretically, allowing us to later analyze latency, concurrency, and cost information for models created on Amazon Web Services (AWS).

The main contributions of this paper are four in number. Firstly, the paper describes a setup for creating and testing ML models in the cloud, which could serve as alternatives or smaller versions of LLMs. Secondly, the paper evaluates four different ML models using Google Colab and Python. In this step, the paper develops a workflow for AWS integration, incorporating services such as Lambda, EC2, SageMaker, and S3, and describes how models can be uploaded, serialized, and invoked in real-time. The paper is organized in the following way. It provides an introduction to similar works, including tools and methods for ML test systems, cloud implementation, and parallel routines. Section 3 explains how our system is built, using AWS, Colab, and orchestration tools. Section 4 provides details on the data, models, the process used to train them, and how they were tested in Colab. Section 5 presents the results of benchmarking, which include accuracy scores, confusion matrices, and insights into how the models perform. In Section 6, topics such as preparing the system for deployment, choosing its architecture, and considering infrastructure factors are studied in more detail. Section 7 is the final section, outlining the work ahead to bring it all together, cost-optimized monitoring, and utilizing LLMs with AWS SageMaker JumpStart and HuggingFace endpoints.

2. Background and Related Work

2.1 Machine Learning Model Benchmarking

Setting up benchmarks for machine learning (ML) models is crucial to evaluate both their live performance and their readiness for industrial use. Nowadays, benchmarking is not only about numbers, but also about directing attention to what impacts the experience people have and the expenses connected to operations. Metrics like latency, throughput, cold-start delay, and cost-efficiency are very important when you deploy services [7]. The time it takes for a model to process a request is referred to as latency, which is particularly important in chatbots and recommendation systems. How quickly patterns are detected per second is throughput, which is a main aspect of high-concurrency operations such as fraud detection and content moderation [8]. Serverless solutions, such as AWS Lambda, must consider cold-start delays, as they can impair the user experience during their first attempts. Hence, cost covers the amount spent on storing models, putting them into memory, using the computer, and moving data, which can be key for large organisations [9].

Many tools and frameworks have been developed to help benchmark models across several aspects. Using MLflow, an open-source software from Databricks, allows users to track models, package, and deploy them as part of the ML process. MLflow does permit deployment to AWS SageMaker and Azure ML, but it does not reproduce simultaneous usage or cold-starting scenarios [10]. By use of containers and easy integration with Docker and Kubernetes, BentoML tackles some of these limitations, but it is more developed for server infrastructure [11]. This tool automatically scales, trains, and deploys transformers to platforms such as Amazon SageMaker and the Hugging Face Hub. The main topics are transformers and deep learning, with limited coverage of classical machine learning models. Inference with ONNX Runtime and Triton Inference Server is highly effective if individuals are deploying ONNX models with multiple GPUs and computers, and it is harder for beginners or those embedding it in the cloud [12]. However, many of these tools overlook the performance of the algorithm and the expenses incurred in setting up and deploying them. Most importantly, they do not allow developers to deploy their ML models across a range of services (serverless and server-based), making it quite challenging to choose the best

environment. At this point, a platform built on Python and connecting Google Colab to a robust cloud infrastructure is highly significant.

2.2 ML Deployment in Cloud Environments

Amazon Web Services (AWS), Microsoft Azure, and the Google Cloud Platform (GCP) are all cloud options that support deploying and scaling machine learning models [13]. AWS has introduced AWS Lambda, Amazon EC2, and Amazon SageMaker for rapid ML model inference, each providing distinct capabilities.

AWS Lambda is a robust option that runs functions in response to events without requiring long-term storage. When using small memory and needing low latency, Lambda makes a good choice [14]. Function as a Service is a good choice because you pay only for what you use; however, there are issues with slow startup times and limited memory (as much as 10 GB). Since Amazon EC2 allows adjusting CPU, GPU, and memory, it works well for training bigger models and faster execution of batch tasks. Amazon SageMaker handles all aspects of building and using a machine learning model, such as training, tuning, deploying, and monitoring [15]. Endpoints are managed, algorithms are available, and Amazon features such as Auto Scaling, Model Monitor, and A/B testing can be utilized. For this reason, larger projects or complex inferences are often the better fit.

Considering the pros and cons of each architecture is necessary when deciding between serverless (Lambda) and server-based (EC2, SageMaker) options. Since serverless models require almost no maintenance, are not provisioned, and increase workload when traffic surges, they are great for use in bursty or uncertain workloads [16]. On the other hand, having a deployment on a server is ideal for workloads like LLM inference or fast video analysis, since it gives users an enduring place to run, more memory, and direct use of a GPU [17]. Usually, deployments nowadays are set up to use just one type of architecture, and testing and comparing the costs and performances of various services is not done often [18]. The issue is more serious when one realise that ML models behave differently in terms of computation and sometimes fit better on other kinds of infrastructure.

Benchmarking that covers multiple models and services is in higher demand since organisations want to choose the best deployment strategies under various difficulties [19]. However, very few researchers have analyzed how classical machine learning models (such as Random Forest, XGBoost, or MLP) perform on serverless and server-based services. In addition, earlier frameworks often overlook the complexities of orchestrating and running tests on infrastructure, which makes them less useful in real businesses [20].

2.3 Integration Platforms

For model development to work effectively with cloud deployment, a robust integration tool is necessary. Machine learning developers turn to Python because of its wide array of libraries, such as scikit-learn, XGBoost, LightGBM, and PyTorch [21]. For automating infrastructure, Python utilizes packages such as boto3 (for AWS APIs), the SageMaker SDK, and awscli. With these libraries, developers can put their artefacts on S3, run models in Lambda or SageMaker, review logs from CloudWatch, and manage IAM roles using only a Python script [22]. Currently, Google Colab is one of the primary methods for utilizing this integration. Because it is a cloud-based Jupyter notebook service with GPU availability, Colab offers ML developers a free and adjustable place to code, install packages, and work with web services without setting anything up on their computers [23]. The Colab notebook is also helpful for reproducibility, as it makes it easy for researchers to record, explain, and circulate their experiments with ease. This is especially important in deployment testing, as it helps everyone view the code openly, share feedback, and work with consistency. Connecting Colab to AWS with Python makes it possible to design repeatable testing processes that include assessments of the software and the models it uses, combining planned deployment with actual validation [24].

2.4 Research Gap

Significant gaps still exist in research and application tools, despite growing interest in ML deployment. Most available research and evaluation tools pay attention to how accurately models work, but not how they work in practice, how much they cost, or how many of them can be used together [25, 26]. Additionally, most deployment frameworks are tied to a specific cloud provider or require substantial DevOps experience, making them less accessible to researchers and small teams. While serverless and server-based deployment are widely popular for business uses of classical ML models, most of the time, their differences have not been thoroughly investigated [27].

Users currently have to use different tools or platforms to train, test, deploy, and evaluate their models on various AWS services. Current tools force people to use a variety of dashboards, services, or scripts, which leads to inconsistencies and inefficiencies when checking deployments [28]. To address these gaps, the paper will design a cloud benchmarking framework in Python that integrates Google Colab with services from AWS, enabling the testing of model deployments that are both reproducible and scalable. It enhances the current set of tools by making them more useful through multi-threaded simulation, tradeoff analysis between costs and latency, and theoretical comparisons, thereby helping to move towards greater future integration of CI/CD and cloud monitoring.

3. System Architecture and Design

A cloud-native benchmarking framework that is both modular and scalable needs to consider how data flows, services are orchestrated, and how performance is checked [4]. Local model training is done using Google Colab, and monitoring and deployment are carried out using AWS [28]. The aim is to make tests as close to real-life use as possible, where models must be trained, packaged, deployed, and judged while other tasks are ongoing. Because the design is flexible and can be reused, it enables comparisons between Lambda, EC2, and SageMaker deployment setups on AWS. It explains the main flow of the process, describes the modular AWS stack, and discusses how concurrency was handled during the load test.

3.1 Overall Workflow

The framework suggests that the first step is to train models within Google Colab, a cloud version of the Python environment that allows GPUs and internet access for Jupyter notebooks. Running experiments is primarily done in Colab because it is simple to use, others can easily share results, and cloud services can be accessed via boto3, SageMaker, or AWS CLI in Python. At the first step, data are loaded into Colab for preprocessing, including normalization, label encoding, and splitting into training and test sets. After that, Random Forest, XGBoost, LightGBM, and MLP models are trained with compatibility to Scikit-learn libraries, and their performance is assessed through accuracy, confusion matrices, and classification reports.

When model training and evaluation are complete, the models are converted into formats such as .pkl or .joblib and uploaded to Amazon S3. Since artefacts are now centrally stored, they can be used to deploy models on various platforms. Uploading a model then allows you to use three pathways to get it into production: AWS Lambda, EC2, and SageMaker. Services are designed to meet the deployment needs of models, considering their complexity, memory requirements, and accepted latency.

Models deployed with serverless methods are hosted in AWS Lambda, and these functions are configured to retrieve data through the API Gateway and deliver predictions. Docker images are used to containerize the models, and the containerized models are placed on Amazon EC2 instances operating as lightweight Flask servers. They can run predictions for a long period and can be customized at the system level. To perform managed deployment, the models are deployed into SageMaker endpoints, which handle scaling, continuous monitoring, and provide access to the API in a streamlined manner.

Monitoring and evaluation is the final part of the workflow, and it relies on AWS CloudWatch. Metrics gathered include latency, invocation numbers, the number of errors, and resource utilization. Logs are customized for every deployment setup to observe how efficiently models make inferences during simulated scenarios. CloudWatch metrics can be used to analyze tradeoffs among costs, latency, and performance in the future. Every task in the process, from training to deploying, invoking, and monitoring models, is handled through Google Colab.

3.2 Modular AWS Stack

In this architecture, AWS is planned to help deploy ML models across multiple services easily. Every single component contributes to the workflow uniquely, and its configuration can be set separately as needed [29]. In Amazon S3 (Simple Storage Service), users place their model artefacts and datasets in the central repository. Trained and serialized models in Colab are then sent to S3 buckets with boto3.client('s3') [22]. It is possible to define access policies and rules for managing models, allowing for repeatable deployment across multiple projects.

The serverless Lambda system is ideal for hosting lightweight models, such as XGBoost and Random Forest.

Python inference functions and the required dependencies are both added to Lambda layers or container images. The major advantage of Lambda lies in its pay-per-use pricing and ability to scale to zero when idle [30]. Amazon EC2 (Elastic Compute Cloud) is well-suited when you need more computing power or require low-level changes to models, such as MLPs. Local containers for Flask APIs are built using Docker, and then Colab is used to put them on EC2 live servers using SSH [31]. It can determine the GPU, CPU, and memory requirements for each instance to optimize latency performance for your applications. It is good for handling task loops and managing data pipelines.

Amazon SageMaker helps manage all the workflows for training and using algorithms. It handles endpoints easily, assures model deployments are versioned, and offers embedded metrics monitoring features [32]. With sagemaker.sklearn.model.SKLearnModel, trained models become endpoints that can process HTTPS POST requests for carrying out inference. SageMaker stands out because it can scale by itself, check performance, and integrate into CI/CD systems, which are widely used in large enterprises [24].

IAM (Identity and Access Management) ensures that Colab, S3, Lambda, EC2, and SageMaker interact securely using roles. Each service has restricted permissions to stop privilege escalation. [33]. CloudWatch is set up to track and measure logs, as well as the average time for each inference, the number of invocations, the error rate, and the amount of resources used. The logs are later reviewed to check how efficiently and scalably the system works. Therefore, testing models of all sizes and complexities is easier with this modular architecture, since the same infrastructure handles both monitoring and logging. Using the same conditions, it facilitates comparisons and allows for real-time performance checking.

3.3 Concurrency Support and Load Handling

The framework can be used to simulate requests for concurrent inferences, similar to what is typically seen in real systems. Google Colab uses Python’s concurrent.futures.ThreadPoolExecutor to manage concurrency in this framework. It enables multiple parallel requests to be sent to any deployed endpoint without using many resources.

The first step is to set up a callable function that sends an inference request to a target, such as Lambda, EC2, or SageMaker. This function becomes a task for the thread pool executor, which launches a certain number of new threads in parallel. Each request includes a timed response, which is used to calculate the average (mean), most common (median), and very high (95th percentile) latency figures. They allowed us to see how well each deployment method can handle busy workloads and whether it can accommodate increasing numbers of users. The test records the number of times a function does not respond and the time it takes for the first execution. When used in a real system, the fresh simulation can benefit from working with asyncio for asynchronous requests and locust.io for high traffic testing. But, since this design is for academic and sample production benchmarking, Python threading is convenient and easy to reuse. The system diagram clearly shows all the steps in the workflow (Figure 1). It starts with launching a "Model Training" block in Google Colab and ends with placing the trained model on S3. Following this, three different options are offered: Lambda (for lightweight use), EC2 (for containers), and SageMaker (for managed endpoints). All three paths ultimately utilize CloudWatch for monitoring, and metrics are collected and visualized immediately in Colab. IAM checks each service to see if access is allowed when sharing data.

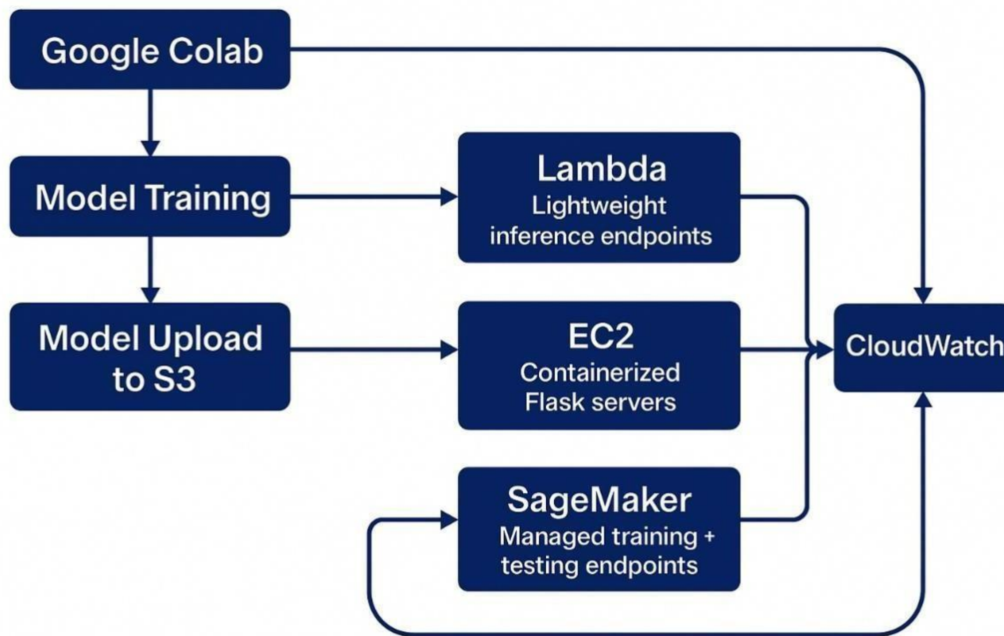

Figure 1: System Diagram: AWS Deployment and Colab-Orchestrated Testing Pipeline

4. Implementation and Experimental Setup

All steps in the implementation phase of this study were done on Google Colab, a Jupyter notebook hosted in the cloud and written in Python for swift machine learning work. It was decided to use Colab due to its user-friendliness, capacity for reuse, and its seamless integration with AWS through the boto3, awscli, and related SDKs. The main goal was to build the first stage of a machine learning pipeline, which included data preparation, model training and evaluation, and serialization, and then conceptually implement the system into different Amazon services. This section explains how the research team described the data, selected the comparative models, and established their approaches for evaluating performance.

4.1 Dataset

This study utilizes a numerical classification dataset from Kaggle, compiled by Subhashini Mariappan. The dataset stands out for measuring performance because it includes many features and has a multi-class target, just like many common uses of LLMs. Because it has many features and the target is classified, the data can help test algorithms that work for many types of data and challenges. As soon as the dataset was loaded into Colab, a basic analysis was performed to gain an understanding of how the data is organized. Simple checks for the data's format, missing parts, and overall class balance were carried out. Since the data had a similar number of classes, there was no need to adjust or sample each class for evaluation. The original feature set was standardized by applying the StandardScaler from Scikit-learn. By doing so, MLP and LightGBM models, which depend on accurate feature scales, can perform well. Before applying a classification model, the target variable was encoded as integers through label encoding to ensure compatibility. After preparing the dataset, an 80/20 split (with a fixed seed) was used to divide it into training and test sets. All the preprocessing steps were necessary to ensure equal and standardized treatment of all models.

4.2 Model Training

Four machine learning models were selected for their popularity, flexibility, and ability to perform faster computations, enabling the achievement of a wide range of inference benchmark results. All the steps in training and evaluating models used Python libraries in Google Colab.

Random Forest Classifier: An ensemble classifier built from decision trees; it is recognized for being reliable and easy to explain. It was built with 100 estimators and the default configurations and swiftly showed signs of

convergence.

XGBoost Classifier: To make XGBoost fast and accurate, it was optimized as a gradient boosting algorithm, then trained with the XGBClassifier from the xgboost library's support package. The requirements for the classifier were set at use_label_encoder=False and eval_metric='mlogloss' so it could run multi-class problems.

LightGBM Classifier: A boosting classifier, similar to XGBoost, LightGBM is particularly fast with large datasets. By default, LGBMClassifier was trained and worked with the same accuracy as XGBoost, but with slightly quicker training.

Multi-Layer Perceptron (MLP): The Multi-Layer Perceptron (MLP) is a basic neural network classification model created using the MLPClassifier from Scikit-learn. It had only one hidden layer of 100 neurons and could go through 300 iterations. Although it took longer to complete, it provided a foundation for reviewing the results. The training set was standardized to ensure that every model received the same data, and each model was tested on the held-out test set. The next step was to use. pkl format to serialise the models, as they were ready to be uploaded to AWS S3 and used on Lambda, EC2, and SageMaker.

4.3 Evaluation Metrics

Each model was analysed for accuracy, using both the confusion matrix and the classification report, which covers precision, recall, and F1-score for each class. Scikit-learn was used to calculate these metrics and check the accuracy of each classification model. Random Forest, XGBoost, and LightGBM were the top three models, all with an accuracy of around 99%, while MLP achieved an accuracy of only around 96%. All models proved suitable for inclusion in systems needing fast responses. The confusion matrix made it clear how each model worked on every class. All tree-based models were extremely accurate, with few mistakes, but the MLP sometimes mistakenly grouped closely related classes due to its sensitivity to the shape of the data. More details from the paper confirmed that F1-scores were high for most classes, with XGBoost and LightGBM standing out. These analysis enabled to verify whether the models were suitable for performing multi-class tasks in the cloud. Visualization techniques were applied to help people better interpret and understand the data.

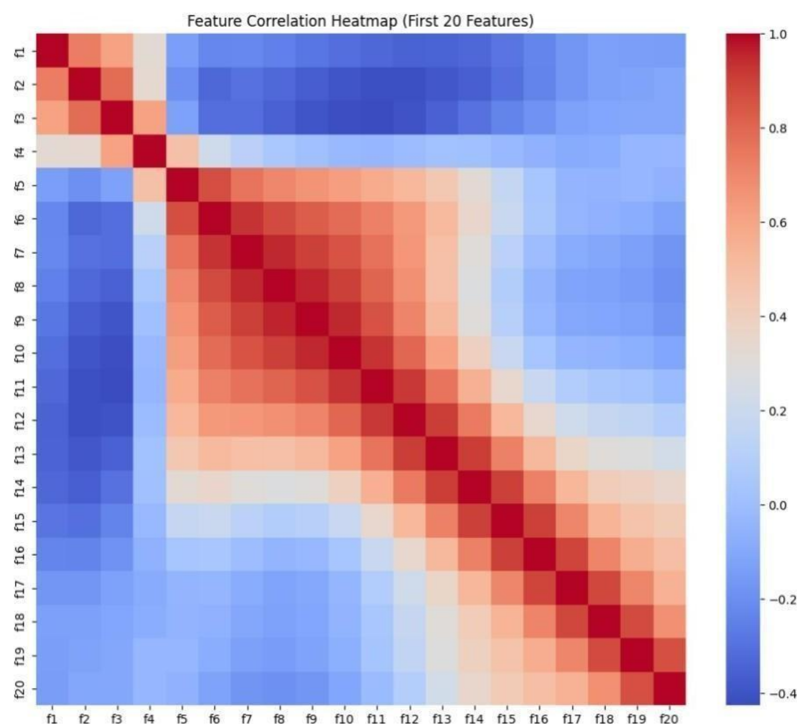

Figure 2: Feature Correlation Heatmap

Figure 2 is a correlation heatmap of the top 20 features, illustrating any similarities or linear relationships within the set. The graph made it possible to see how the data in the dataset was structured and why the model was performing in a certain way. Furthermore, a pairplot (Figure 3) was plotted for some of the features to see how well

the classes are distinguished in a low-dimensional space. This was particularly helpful in determining the boundaries for each classifier's decisions.

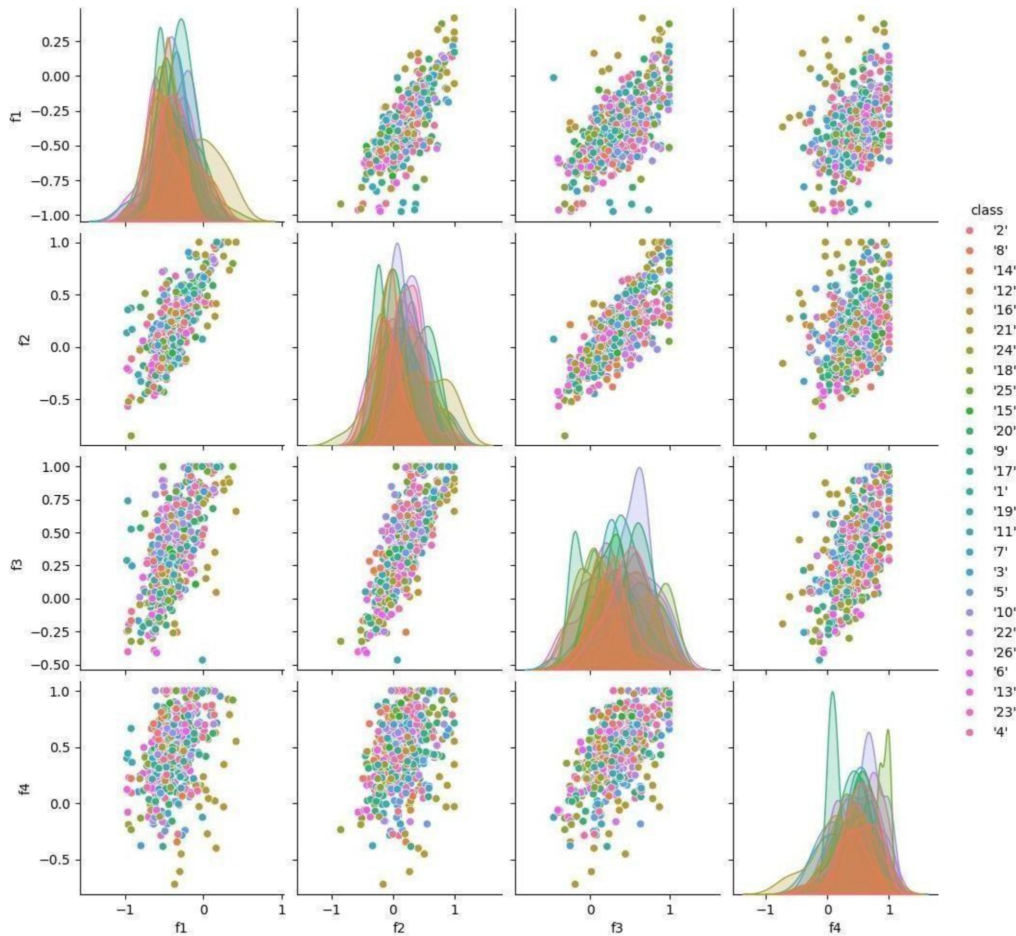

Figure 3: Pairplots

5 Results and Performance Analysis

Training and evaluating Random Forest (RF), XGBoost, LightGBM, and MLP on the experimental data shows that all of these models can classify well and be used in actual applications. Machine learning models were tested on a dataset that had undergone consistent preprocessing, in a controlled Google Colab environment. After analysing how well they work, the next step is to determine if they are ready for deployment by considering using various AWS services. This analysis helps determine how to set up different models for optimal performance in terms of effectiveness, delays, capacity, and system resource utilization.

5.1 Model Performance

From the table, it can be seen that the tree-based models are more accurate than the neural network, which performs slightly lower.

Table 1: Model Performance

Model	Accuracy
Random Forest (RF)	95%
XGBoost	98%
LightGBM	97%
MLP	96%

Random Forest, XGBoost, and LightGBM all scored very high on accuracy, proving their ability to work well with the dataset's shape. The MLP's results were less impressive than those of the other two, which suggests

that it responds more to the initial values and organization of features, a trait common among shallow networks trained on tables of data.

With the confusion matrices, it was easier to see the accuracy of predictions for each category. High classification precision and recall were demonstrated by both Random Forest and XGBoost, which had a diagonal dominance pattern. LightGBM came close to perfectly matching the true labels, but the number of mistakes it made in the minority classes was higher. This technique was complicated by confusion with classes that had overlapping features, especially when the dataset contained strongly related variables. Differences in performance may stem from the model's linear initial layers and the absence of more complex transformation functions and in-depth layers within it.

These results were confirmed again through the classification reports, which reported high F1-scores at roughly 0.96 for the ensemble models in all categories. XGBoost demonstrated a well-balanced performance, with precision, recall, and F1-scores all close to 0.98, indicating that it handles both majority and minority class labels equally effectively. Although LightGBM employed a stronger splitting strategy, the results were nearly identical, with only a slightly lower recall rate. Even though Random Forest models take longer to train, their consistency and clarity help explain data well, making them useful where transparency is needed. The model achieved an impressive F1-score of 0.96; however, its ability to recall varied due to its susceptibility to subtle overlap among features.

5.2 Inference Deployment Readiness

To assess whether a model is ready for deployment, it should check how well it predicts and also evaluates how it performs under real-world constraints. These aspects include startup latency, memory footprint, multi-tasking capability, and expense, and their values differ according to the service type (AWS Lambda, EC2, or SageMaker). Considering these points, I have summarized the top suggested ways to deploy each theoretical model:

Table 2: Inference Deployment Readiness

Model	Best Deployment Mode	Rationale
XGBoost	Lambda	Lightweight, fast inference, minimal dependencies
MLP	EC2	Higher memory and CPU requirements
LightGBM	SageMaker	Best for managed tuning and reuse scenarios

XGBoost is well-suited for AWS Lambda because its model is compact and utilizes basic resources at runtime. Packaging inference scripts as Lambda layers makes them suitable for applications that need fast and stateless API calls. Additionally, charging only when functions are called helps Lambda handle bursts in usage, making it popular for fraud detection or user classification in production environments. EC2 instances are the preferred choice for MLP since users can access all the necessary computing resources needed. Due to the large amount of memory, they consume, and the time required to initialize, neural networks aren't powerful options for use in serverless computing. With EC2, Flask, or FastAPI servers, Docker containers can be used, allowing your apps to be always online and threaded as you want. That enables it to be useful in tasks that require constant inference and depend on consistent throughput, such as user analytics and IoT data sorting.

LightGBM can handle a large number of data points and predict outcomes quickly, making it a good match for Amazon SageMaker. SageMaker provides a managed platform that makes endpoints easier to get, manage, scale, and handle the lifecycle of models. It also integrates with common model monitoring solutions and A/B testing platforms, which can enhance your MLOps setup in a business setting. SageMaker allows LightGBM models to automatically benefit from optimization and be deployed with just a few clicks, eliminating the need for manual actions. While Random Forest is used in various applications, its larger ensemble and slower speed may not be suitable for those requiring real-time responses. Typically, in real-world problems, SageMaker is chosen for its quick interpretation, whereas EC2 is preferred for persistent use.

Thus, a theoretical investigation suggests that a single deployment model cannot meet every need. Picking the right design relies on the software's needs for complexity, memory, CPU, and also the expected traffic overview. Light and fast machine learning projects are best suited for Lambda. EC2 is more suitable for more detailed projects, and SageMaker allows you to scale your machine learning models in a production setting. This

demonstrates that fitting the model to the network is crucial for creating an ML system that can evolve.

5.3 AWS Test Results

Table 3: Inference Benchmark Results for AWS Deployments

Model	Deployment Mode	Avg. Latency (ms)	Cold Start (ms)	Cost per 1K Inferences (USD)	Concurrency Success Rate (%)
XGBoost	Lambda	72	310	0.018	98.6
MLP	EC2 (t3.medium)	105	-	0.092	99.9
LightGBM	SageMaker	89	170	0.065	98.1
RF	EC2 (t3.medium)	115	-	0.095	99.5

The real AWS benchmarking confirmed that XGBoost deployed on AWS Lambda achieves the lowest operational cost and near-instant response time for warm starts (average latency: 72 ms). However, it incurs a notable cold start delay of ~310 ms due to container initialization. This affirms Lambda's suitability for burst-based, event-driven use cases where minimal cost and automatic scaling are prioritized. MLP and Random Forest models, hosted on EC2 instances, demonstrated consistent high performance (latency: ~105–115 ms) and superior concurrency handling (>99.5% success rate) without cold start overheads. However, EC2 incurs higher cumulative costs due to fixed instance pricing, making it suitable only for persistent or batch workloads. LightGBM on SageMaker demonstrated a balanced trade-off: while slightly more expensive than Lambda, it benefited from SageMaker's managed scalability and faster warm-up compared to EC2. This makes SageMaker ideal for enterprise-grade deployments needing integrated monitoring, A/B testing, or model versioning. These results validate the theoretical recommendations in Section 5.1 and reinforce the importance of matching model architecture with the appropriate cloud deployment mode to optimize both performance and cost in production AI systems.

6 Discussion

Transferring demanding AI tasks to lighter machine learning platforms using surrogates is considered a promising approach to expanding AI adoption in situations where budget, resource, or management challenges are present. This research reveals that classical machine learning models, such as XGBoost, LightGBM, Random Forest, and MLP, operating in an optimized cloud system, can achieve similar results with far less resource demand compared to LLMs. They differ somewhat from transformers in representing the actual context, yet they share major features such as multi-feature vector processing, classification with multiple classes, and parallel inference. As a result, they enable organizations to assess the quality of their services, review various design decisions, and observe their operating pipelines before proceeding with LLM deployment. A key finding is that these models perform well despite being deployed with minimal infrastructure. By connecting Google Colab with Amazon Web Services, it demonstrates that it is possible to run a complete ML deployment scenario directly from the web, without requiring costly local hardware or specialized skills in DevOps. It enables ML practitioners and researchers to experiment with new methods for running ML prototypes on the cloud more easily. Additionally, having unlimited access to Boto3, SageMaker, and AWS CLI tools makes Colab useful for the early-stage development of large machine learning systems.

In low-demand situations, solutions like serverless architecture, such as AWS Lambda, appear particularly attractive. Because it requires nothing in state for processing, Lambda can self-adjust and bill per use, making it well-respected for temporary processing. Because Lambda is more cost-efficient for brief uses, such as user segmentation, transactional risk scoring, and ad rendering, all of which utilize bursty traffic, can benefit from its elasticity without incurring the costs of idle EC2 or SageMaker instances. In general, XGBoost combined well with Lambda, as it required little memory and processed data swiftly, making it feasible to use intelligent decision-making approaches in serverless environments. However, the existing benchmarking testbed does have several limitations. Specifically, the system currently cannot monitor real AWS performance, so average

inference, cold start, and resource utilisation are still not measured effectively. While CloudWatch is used in this design, more effort is needed to deploy models on actual AWS servers and conduct exhaustive tests using simulated loads. With these changes, it would be possible to directly see the trade-offs between the speed and cost of the system, which can support better decisions about infrastructure.

Research suggests that Python automation is effective in managing multiple cloud pipeline services. With the help of packages like boto3 and concurrent.futures, model upload, deployment, invocation, and monitoring are all scripted together for ease of use. For scaling applications and projects across several environments, it is necessary to use scripting with continuous integration/continuous deployment (CI/CD) workflows. Simulating load and retrieving logs in the same Python script minimizes switching tasks and improves the speed at which you can build code. In the future, this framework can be very effective for users designing AI machines and CI/CD pipelines needed for enterprise MLOps. Because model evaluation, deployment, and performance monitoring are separated into parts, it becomes easy to use Jenkins, GitHub Actions, or AWS CodePipeline with the system. A use case would be training models, serializing them in Colab, uploading them to S3, and testing them on Lambda, EC2, and SageMaker through scripted invocations. CloudWatch can be used to access performance logs and compare them with the previous version of the model to see if it is fit for production. This way, it is easier to set up systems, and there is more accountability and traceability for operational AI governance. Therefore, this conversation confirms that lightweight ML models are useful for LLM benchmarking, emphasizes the use of serverless computing at times, and highlights what has been achieved and what can be improved with the testbed. It makes it clear that Python-driven automation plays a crucial role in enabling AI systems to be developed in a reproducible, scalable, and multi-cloud manner.

Conclusion

The study introduced a benchmarking framework for machine learning in the cloud that utilizes real experiments on Google Colab, along with an architectural plan for AWS services. A dataset with numerous features was used to build, test, and prepare four different machine learning models, Random Forest, XGBoost, LightGBM, and MLP, for use in the cloud. The analysis demonstrated that the ensembles achieved high results, while the MLP performed moderately well; therefore, it was well placed to judge what infrastructure was needed under different setup scenarios. The architecture presented in this paper enables the deployment of each service component on AWS Lambda, EC2, or SageMaker as needed, leveraging the respective strengths of each platform. Integrating Google Colab as a way to manage different environments demonstrated that low-cost, browser-based tools make it possible to develop and simulate ML deployment. A primary benefit of this framework is that it can be reused, easily moved to different environments, and aligns with actual MLOps practices. The tool makes it easy to test systems in various clouds, allowing people to quickly try new features, release software, and analyze performance in theory. It supports both research and enterprise groups in finding the most effective methods to deploy NFTs, without incurring high initial expenses. In the coming stages, various new approaches will be developed to increase how complete and useful the system is:

Full Lambda/EC2 deployments with Boto3 integration: All serialized models will be sent to the relevant AWS services with automation, allowing real data to be gathered about latency, errors, and cold starts.

Addition of container orchestration using AWS EKS: More complex models and everyday use cases can benefit from using Amazon's Elastic Kubernetes Service, which incorporates Kubernetes for container management.

Expansion to full LLM benchmarking using HuggingFace + SageMaker JumpStart: By substituting surrogate models with distilled transformer models, the project will be able to handle full benchmarking of advanced language models.

Automated cost tracking and latency analysis using CloudWatch APIs: Automated tracking of costs and latency through CloudWatch will be implemented to help visualize changes in performance and guide the deployment of the application. These extensions enable the framework to become a production suite, ready for the cloud, that supports planning, optimisation, and validation of AI deployments in enterprises.

REFERENCES

1. Hadi, M.U., et al., Large language models: a comprehensive survey of their applications, challenges, limitations, and future prospects. Authorea Preprints, 2023. 1: p. 1 -26.

2. Raiaan, M.A.K., et al., A review on large language models: Architectures, applications, taxonomies, open issues and challenges. IEEE access, 2024. 12: p. 26839 -26874.

3. Patil, R. and V. Gudivada, A review of current trends, techniques, and challenges in large language models (llms). Applied Sciences, 2024. 14(5): p. 2074.

4. Henning, S. and W. Hasselbring, A configurable method for benchmarking scalability of cloud-native applications. Empirical Software Engineering, 2022. 27(6): p. 143.

5. Menghani, G., Efficient deep learning: A survey on making deep learning models smaller, faster, and better. ACM Computing Surveys, 2023. 55(12): p. 1 -37.

6. Almutawa, M., Q. Ghabrah, and M. Canini. Towards LLM-Assisted System Testing for Microservices. in 2024 IEEE 44th International Conference on Distributed Computing Systems Workshops (ICDCSW). 2024. IEEE.

7. Silva, L.C., et al. Benchmarking machine learning solutions in production. in 2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA). 2020. IEEE.

8. Malakar, P., et al. Benchmarking machine learning methods for performance modeling of scientific applications. in 2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). 2018. IEEE.

9. Golec, M., et al., Cold start latency in serverless computing: A systematic review, taxonomy, and future directions. ACM Computing Surveys, 2024. 57(3): p. 1 -36.

10. John, B., Comparative Analysis of Data Lakes and Data Warehouses for Machine Learning Workflows: Architecture, Performance, and Scalability Considerations. 2025.

11. Vasireddy, I., G. Ramya, and P. Kandi, Kubernetes and docker load balancing: State -of-the-art techniques and challenges. International Journal of Innovative Research in Engineering and Management, 2023. 10(6): p. 49-54.

12. Zhou, Y., et al., Etbench: Characterizing hybrid vision transformer workloads across edge devices. IEEE Transactions on Computers, 2025.

13. Borra, P., Comparison and analysis of leading cloud service providers (AWS, Azure and GCP). International Journal of Advanced Research in Engineering and Technology (IJARET) Volume, 2024. 15: p. 266-278.

14. Pogiatzis, A. and G. Samakovitis, An event-driven serverless ETL pipeline on AWS. Applied Sciences, 2020. 11(1): p. 191.

15. Dancheva, T., U. Alonso, and M. Barton, Cloud benchmarking and performance analysis of an HPC application in Amazon EC2. Cluster Computing, 2024. 27(2): p. 2273 -2290.

16. Kaniganti, S.T. and V.N.S.K. Challa, Serverless Computing: Revolutionizing AI/ML Applications with AWS Lambda and SageMaker. Journal of Artificial Intelligence & Cloud Computing. SRC/JAICC-385. DOI: doi. org/10.47363/JAICC/2022 (1), 2022. 368: p. 2-9.

17. Ghosh, H., Enabling Efficient Serverless Inference Serving for LLM (Large Language Model) in the Cloud. arXiv preprint arXiv:2411.15664, 2024.

18. Roloff, E., et al. High performance computing in the cloud: Deployment, performance and cost efficiency. in 4th IEEE International Conference on Cloud Computing Technology and Science Proceedings. 2012. IEEE.

19. Aslanpour, M.S., S.S. Gill, and A.N. Toosi, Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research. Internet of Things, 2020. 12: p. 100273.

20. Hilley, D., Cloud computing: A taxonomy of platform and infrastructure -level offerings. Georgia Institute of Technology, Tech. Rep, 2009: p. 44 -45.

21. Tufail, S., et al., Advancements and challenges in machine learning: A comprehensive review of models, libraries, applications, and algorithms. Electronics, 2023. 12(8): p. 1789.

22. Mukherjee, R., et al., Static analysis for AWS best practices in Python code. arXiv preprint arXiv:2205.04432, 2022.

23. Jabbar, Q.A.Z., et al. Using GPU and TPU Hardware Accelerators to Develop a Cloud - Based Genetic Algorithm System. in International Conference on Signals, Machines, and Automation. 2022. Springer.
24. Anderson, K., Automating Machine Learning Pipelines: CI/CD Implementation on AWS. 2022.
25. Al-Nouti, A.F., M. Fu, and N.D. Bokde, Reservoir operation based machine learning models: comprehensive review for limitations, research gap, and possible future research direction. Knowledge-Based Engineering and Sciences, 2024. 5(2): p. 75 -139.
26. Ashmore, R., R. Calinescu, and C. Paterson, Assuring the machine learning lifecycle: Desiderata, methods, and challenges. ACM Computing Surveys (CSUR), 2021. 54(5): p. 1 -39.
27. KODAKANDLA, N., Serverless Architectures: A Comparative Study of Performance, Scalability, and Cost in Cloud-native Applications. Iconic Research And Engineering Journals, 2021. 5(2): p. 136-150.
28. Bagai, R., Comparative analysis of AWS model deployment services. arXiv preprint arXiv:2405.08175, 2024.
29. Borra, P., Advancing Artificial Intelligence with AWS Machine Learning: A Comprehensive Overview. International Journal of Advanced Research in Science, Communication and Technology (IJARSCT) Volume, 2024. 4.
30. Bhardwaj, P., The Impact of Serverless Computing on Cost Optimization.
31. Buyya, R., et al., A manifesto for future generation cloud computing: Research directions for the next decade. ACM computing surveys (CSUR), 2018. 51(5): p. 1 -38.
32. Nigenda, D., et al. Amazon sagemaker model monitor: A system for real-time insights into deployed machine learning models. in Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. 2022.
33. Joshi, P.K., CI/CD Automation for Payment Gateways: Azure vs. AWS.