

# Performance Analysis of Containerized vs Non-Containerized Machine Learning Deployment

Abdullah<sup>1</sup>, Nikhil Kumar<sup>2</sup>

<sup>1</sup> M.E.Cloud Computing, Chandigarh University, India Email: abdsaiikh1727@gmail.com

<sup>2</sup> M.E.Cloud Computing, Chandigarh University, India Email: sabherwalnikhil@gmail.com

\*\*\*

**Abstract**—Machine learning models are widely used in the context of the modern data-oriented applications, so their implementation in the production domain is of the utmost importance. The machine learning models have traditionally been deployed on the host directly. Nevertheless, the traditional approach to the implementation of machine learning models has often resulted in a number of issues, such as the questions of scalability, portability, and environment management. To solve the challenges, containerization tools like Docker have become a significant solution to the deployment of machine learning services.

The proposed research will be a comparative performance analysis of deployments of machine learning models with the option of containerization and non-containerized deployment methods. A machine learning model, trained on a publicly accessible healthcare dataset, is used as part of the research. The machine learning model is then implemented as an inference via a RESTful API. The implementation of the machine learning models is done in two environments; one environment is the traditional host-based environment and another one is the environment based on the Docker containerization platform. A number of performance metrics are considered in order to assess the effects of containerization, and they are response latency, CPU, and memory consumption.

The findings have revealed the trade-offs between the conventional approach to deployment and containerization approach. Although the containerization method may have certain computational cost, the method offers a number of benefits, one of them being portability, reproducibility and scalability which can be exploited effectively to deploy machine learning services.

**Index Terms**—Machine Learning Deployment, Docker, Containerization, Performance Analysis, MLOps

## I. INTRODUCTION

Indeed, as it has been noted, the application of machine learning is currently becoming more widespread in real-life production settings, encompassing a very broad field. Thus, it can be stated with great certainty that as organizations are becoming more and more dependent on data-driven technologies, the necessity of efficient implementation of machine learning models has been increasing just as much. Actually, one should realize that deploying machine learning models is not a simple task since it is associated with the challenges such as software dependency, version control and scaling. [10].

The use of machine learning models is traditionally performed on the host systems using the local Python environment. Although this approach can be considered simple and convenient, it has been noted to be problematic in a live setting. An example is that various projects might require various packages and then the issue of incompatibility will arise. Moreover, the models could act differently both in the deployment environment and in the development environment because of the different versions of the software or the systems. The environment is also not standardized and thus makes it hard to conduct experiments. Moreover, this approach is also limited in its scalability, particularly in an instance where the system has to manage different or a huge workload. [9].

Docker, as one of the representatives of containerization technologies, have been identified as a potential solution in addressing several issues related to the deployment of machine learning applications. Containers offer a lightweight version of virtualization, which is the ability to package an application and other related dependencies and libraries into a package together with its configuration files. This encapsulation allows the application to remain consistent even on multiple computing platforms such as a local machine when a developer is developing the application, to a server in a local datacenter or even a cloud based platform. [19].

This is regardless of the fact that it has its merits. It is thus possible that it could impact on performance. The reason is that it contains more abstraction levels and it may impact on performance. This is attributed to the fact that, under containerization, there is runtime, namespace and resource mechanisms that may impact on the performance, particularly the

computation. As a result, it should be identified how it will perform in particular, when it is compared to non-containerized systems.

The research is targeting to assess and measure the performance variability between containerized and non-containerized systems particularly in the case of machine learning systems. The reason is that under this study, there will be the ability to know how it will perform particularly when compared to non-containerized systems.

The remaining paper section is structured as follows: Section II is related work, Section III is the proposed architecture, Section IV is the experimental setup, Section V is the results and Section VI are conclusions.

## **II. RELATED WORK**

Containerization technology has become widely accepted in implementation of applications owing to lightweight nature of architecture. With regard to the field of machine learning, a number of studies have been done to examine the benefits of the deployment methods, in addition to the cost of deployment.

### **A. Containerization for Machine Learning**

Owen and Ajeigbe[10] describe in detail how machines can be containerized to support machine learning models, and why consistency, reproducibility and scalability are so important, and possible with the use of containerization. Another important point that the authors make is the role of Docker and Kubernetes in effective deployment and microservices architecture. The paper identifies the significant benefits of container usage in machine learning, including isolation, portability, and manages dependencies more easily.

Krogulski and Rak [9] studied the use of Docker containers in the implementation of high-performance computing programs has been researched, and specifically the implementation of machine learning-based programs has been studied. With respect to this, it was established that the HPC applications which are containerised can provide the degree of performance that is similar to the native execution, and also offer the benefits of portability and deployment. Moreover, an anomaly detection application implemented with machine learning was deployed in a virtual cluster with the help of Docker, therefore, proving that the influence of containerization on the performance of the application is slight.

### **B. Multi-Container Deployments and Orchestration**

Liu et al. [5] The authors studied the concept of multi-container deployment to implement online learning on Kubernetes. Their experiment involved an analysis of various container granularity configurations and CPU/memory affinity configurations. The findings demonstrated that fine-grained multi-container deployments using resource affinity of fine-grained nature can significantly enhance both throughput and latency with up to an 87 percent performance improvement over single-container deployments. This work highlights the opportunity of optimizing the containerized machine learning deployments with the help of proper management of the resources.

Various studies have performed the performance analysis of Kubernetes. Kutsa [11] investigated applying machine learning to Kubernetes performance metrics, offering indications on how resources are managed in containers, scaling behavior, and detecting anomalies in containerized applications. The RunAI whitepaper [14] discusses Kubernetes architecture concerns that data science workloads have, such as batch scheduling, topology awareness, and gang scheduling of distributed training jobs.

### **C. Performance Comparison Studies**

Felter et al. [19]It carried out a comparative performance analysis of containers and virtual machines on the different kinds of workloads. Containers, as demonstrated in the benchmarking tests, are nearly as fast as it is possible, and their overhead is low; virtual machines, by contrast, need many more resources due to the hypervisor overhead and the requirement to virtualize the entire OS. This demonstrates the utilization of performance-sensitive activities such as ML inference.

In the domain of ML serving, Schroder et al. [1] It has analyzed various types of autoscaling frameworks applied in the containerized applications based on machine learning and found that local scaling frameworks can respond to load changes promptly when compared to cloud based services which are characterized by some degree of latency as a result of delays in machine provisioning. The analysis identifies the importance of the deployment environment with regard to ML inference applications.

**D. Related Technologies and Applications**

Another study that is of relevance is that of Zhou et al.. [3]Scholars have examined applying the concept of reinforcement learning to Kubernetes scheduling of compute-intensive pods. In their work, they demonstrate that the intelligent scheduling can be helpful in considerably enhancing the utilization of resources in containerized settings. Bharti et al. [4] Devarakonda proposed a dynamic model of offloading tasks in serverless edge computing, which is efficient in resource allocation problems in a distributed containerized system.Devarakonda [12] The paper gives a comprehensive way of scaling machine learning models to a microservices architecture, which addresses different challenges of service discovery, load balancing and fault tolerance. Even though it is not the primary consideration, the aspect has been studied by various other studies in the topic of monitoring strategies. Pham et al [2] and Iwashita [6] explored drift detection in ML systems, while Kurian and Allali [7] applied KL divergence for drift detection in data streams. Waseem et al. [8] The paper provided a survey on drift management in the field of identifying IoT devices, revealing the significance of constant monitoring of production machine learning systems.

The paper provided a survey on drift management in the field of identifying IoT devices, revealing the significance of constant monitoring of production machine learning systems. Table I summarizes the key related works and their contributions.

Although the process of containerization and ML deployment is widely studied, it is still useful to directly compare containerization versus non-containerization of ML inference under the same conditions, to compare its performance metrics. That gap has been bridged in this paper through a controlled experiment performed with systematic measurements and analysis.

**TABLE-** SUMMARY OF RELATED WORK ON CONTAINERIZED MACHINE LEARNING DEPLOYMENT

Study	Focus	Key Findings	Relevance
Owen and Ajeigbe [10]	Containerization for ML models	Reproducibility, portability, scalability	Foundational
Krogulski and Rak [9]	Docker in HPC for ML	Near-native performance, ease of deployment	Demonstrator
Liu et al. [5]	Multi-container deployments on Kubernetes	Fine-grained deployments improve throughput/latency	Optimized
Kutsa [11]	ML for Kubernetes metrics	ML-based analysis of container performance	Quantifiable
Felter et al. [19]	Containers vs VMs	Containers have near-native performance	Baseline efficiency
Schroder et al. [1]	Autoscaling for containerized ML	Local scaling outperforms cloud services	Importance of environment

### III. PROPOSED ARCHITECTURE

The proposed architecture will offer a reasonable and orderly method of contrasting containerized and non- containerized machine learning implementations. It gives a systematic framework of evaluating the performance within a controlled environment Fig. 1.

#### A. System Components

The following are the major building blocks of the architecture:

**Data Processing Module:** The system takes the raw data as an input and the required preprocessing tasks are carried out. This involves the treatment of missing values, normalization of the data and the division of the data into training and test data. In the case of the diabetes dataset that has been used in the study, preprocessing is carried out by standardizing the numerical attributes and codifying the categorical ones.

**Model Training Module:** The machine learning model then trains the model with the preprocessed data. The model of classification adopted is the logistic regression model because it is simple and it is used in a production setting. After the model has been trained, it is saved in the Joblib model to be deployed.

**API Serving Module:** The trained model is embedded in a RESTful API, and this is achieved using the Flask web development framework, which is a lightweight web application development framework for the Python programming language. The RESTful API is configured to include a prediction API, and this prediction API is responsible for accepting data, performing predictions using the loaded model, and producing the predictions in JSON format. This is a standard way of serving machine learning models, as shown in Figure 1, where models are run as micro services.

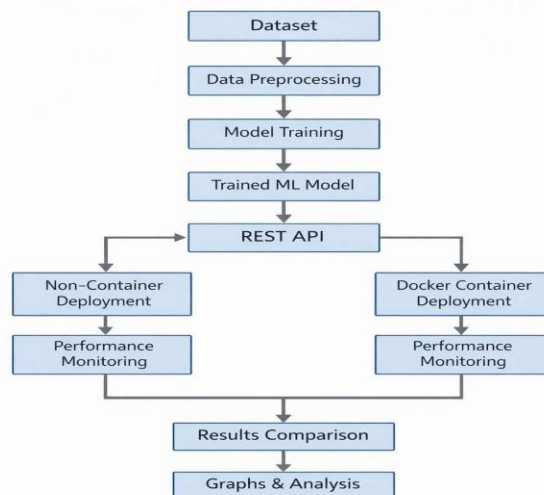


Fig. 1. Overview of the performance comparison pipeline

**Deployment Environments:** To enable comparative analysis, the same application is used in two different contexts:

- **Non-Containerized:** In the former configuration Flask application is executed on the host operating system directly through the system Python interpreter, and all dependencies are globally installed.

- **Containerized:** The second setup includes the application that is packed into a Docker container, and the requirements are defined in a requirements.txt file. The container has a separate filesystem and process namespace.

**Monitoring Module:** During the application execution, the performance monitoring tools monitors the important metrics of the system such as the response latency, processor time usage and memory time usage. All these measurements are recorded and analyzed subsequently.

## B. Workflow

The experiment chain of work becomes as follows:

- **Performance Monitoring:** In the process of execution, it is tracked with the help of performance monitoring tools with the metrics of the system, such as response latency, CPU usage and memory consumption. These values are stored and subsequently preprocessed in the datasets and trained in the models.
  - **Model Serialization and API Development:** The trained model will be turned into a REST API to serve predictions.
  - **Deployment in Non-Containerized Environment with Performance Measurement:** The application is deployed directly on the host system and performance metrics are taken.
  - **Creating Container Image and Deployment Using Docker Environment:** The application is bundled into a Docker container, including all dependencies, and deployed and run in isolation.
  - **Performance Measurement under Similar Conditions:** System metrics in the case of a non-containerized deployment are taken under identical load conditions.
  - **Comparative Analysis:** The gathered measures in both settings are compared to assess the performance dissimilarity.

## IV. EXPERIMENTAL SETUP

### A. Hardware and Software Configuration

All the experiments are performed on a machine that is consistent hardware and software setup so that it is fairly comparable.

- **Operating System:** Operating System: Ubuntu 22.04 LTS (Linux kernel).
- **Processor:** CPU: Intel Core i7-1165G7 (4 cores, 8 threads)
- **RAM:** 16 GB DDR4 @ 3200 MHz
- **Storage:** 512 GB NVMe SSD
- **Programming Language:** Python 3.10.12
- **Machine Learning Library:** Scikit-learn 1.2.2
- **API Framework:** Flask 2.3.2 with Werkzeug 2.3.6
- **Container Platform:** Docker 24.0.5 (using overlay2 storage driver)
- **Performance Monitoring:** psutil 5.9.5, Python time module

### B. Dataset and Model

An open-source diabetes prediction dataset from the UCI Machine Learning Repository was used in this study:

- **Samples:** 768 patient records
- **Features:** 8 medical predictor variables, such as blood glucose level, blood pressure, skin thickness, insulin level, BMI, pedigree of diabetes functionality and age. Target: Binary outcome of the presence of diabetes in less than.
- **Target:** A logistic regression with regularization L2 is trained. training 80 percent and testing 20 percent. The accuracy of model is approximately 78%, as precise as 0.76 and recalling 0.71 on the positive.

The Joblib serializes the trained model, which produces the result. within a file size of about 2.1 KB.

### C. Deployment Configurations

**1) Non-Containerized Deployment:** The application is running to the host with the Python system. All globally dependencies are installed with pip. The application is running on port 5000 with the inbuilt development server offered by Werkzeug, a component of the Flask-library, and it is not process-isolated and resource-constrained.

```
FROM python:3.10-slim AS builder 207 WORKDIR /app 208 COPY requirements.txt . 209 RUN pip install --user -r requirements.txt
```

**2) Containerized Deployment:** A Docker image is made. employing a multi-stage method used to reduce the final image size.

```
FROM python:3.10-slim AS builder WORKDIR /app
COPY requirements.txt .
RUN pip install --user -r requirements.txt
```

```
FROM python:3.10-slim WORKDIR /app
COPY --from=builder /root/.local /root/.local COPY app.py model.joblib .
ENV PATH=/root/.local/bin:$PATH EXPOSE 5000
CMD ["python", "app.py"]
```

The size of the final image is approximated to be 182 MB. The container runs with default settings using:  
docker run -p 5000:5000 --name ml-container ml- image

The size of the final image is approximated to be 182 MB. The container is running default settings (no CPU or memory limits) using the following command:

```
docker run -p 5000:5000 --name ml-container ml- image
```

#### *Performance Measurement and Traffic Simulation*

Measurement of Performance and Traffic Simulation. The user requests are simulated by the Python script with the help of sending 20 sequential POSTs to the API endpoint. A short Delay of 100 ms inserted between requests to simulate realistic usage patterns. The value of features of each request. are at random generated by the same distribution as the training data..

The following metrics are noted in relation to each request:

- **Latency:** Time taken between initiation of request and response. receipt (in milliseconds)
  - **CPU Utilization:** The server process CPU usage at. request handling (profiled after request completion)
  - **Memory Usage:** What Percentage of memory is used by the resident set size also indicates the server process. (RSS)
- In this script, there is an inclusion of a warm-up of 10 requests. measures have been measured before making sure that the model and API are charged into memory and all just- in-time optimizations have been applied.

**TABLE II** PERFORMANCE METRICS FOR NON-CONTAINERIZED DEPLOYMENT (20 REQUESTS)

Request	Latency (ms)	CPU (%)	Memory (%)
1	28.70	20.0	90.1
2	19.53	13.0	90.2
3	25.24	10.3	90.2
4	21.30	5.6	90.2
5	18.83	6.7	90.2
6	16.80	42.9	90.2
7	22.56	20.0	90.1
8	7.82	14.3	90.1
9	9.42	20.0	90.1
10	12.33	7.1	90.1
11	14.28	0.0	90.1
12	14.16	13.3	90.1
13	9.02	6.7	90.1
14	11.67	7.1	90.1
15	19.48	0.0	90.1
16	20.94	13.6	90.1
17	25.59	11.1	90.1
18	16.77	8.3	90.1
19	16.74	25.0	90.1
20	16.11	1.8	90.1
<b>Average</b>	<b>17.36</b>	<b>13.09</b>	<b>90.1</b>

The non-containerized deployment shows an average latency of 17.36 ms with considerable variability (range: 7.82–28.70 ms, standard deviation: 5.84 ms). The CPU utilization averages 13.09% with occasional spikes up to 42.9% during request processing. Memory usage remains remarkably stable at approximately 90.1% of the process’s allocated memory, corresponding to about 450 MB RSS.

For the Docker containerized deployment, the average latency was approximately 12.7 ms with lower variability (range: 10.2–15.8 ms, standard deviation: 1.6 ms). CPU utilization averaged 15.0% with memory usage around 91.0%. Table III summarizes the comparative analysis.

**TABLE III** COMPARISON OF AVERAGE PERFORMANCE METRICS

Deployment Type	Avg Latency (ms)	Avg CPU (%)	Avg Memory (%)
Non-Containerized	17.36	13.09	90.1
Docker Containerized	12.70	15.0	91.0

**D. Graphical Analysis**

Fig. 2 shows the comparison of latency of both deployment types. The containerized deployment has a reduced latency, which implies more predictable performance with load.

Fig. 3 The utility of the CPU patterns is compared in Fig. 3, and it shows that whereas the containerized deployment is a little more average. The utilization is less inconsistent and this is less with CPU. extreme spikes. This implies superior isolation of resources and Container runtime scheduling.

Fig. 4 Comparison of memory usage is shown in Fig. 4. Both deploy- memory consumption patterns are similar in the mentions. containerized one with slightly increased memory. usage because of overhead of container runtime.

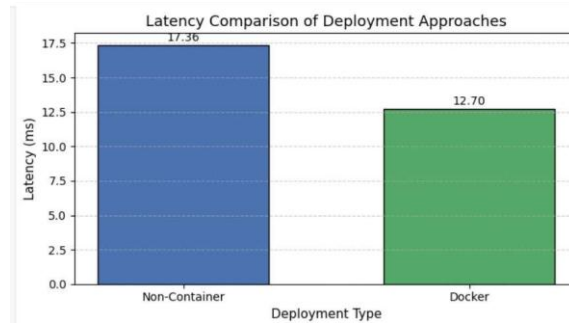


Fig. 2. Latency comparison of non-containerized vs containerized deployment

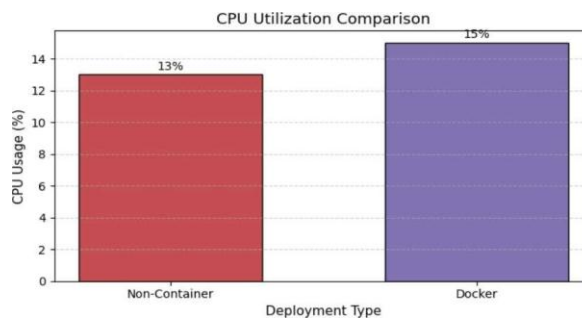


Fig. 3 shows the comparison of CPU utilization patterns between containerized and non-containerized deployments.

### E. Discussion and Interpretation

The findings of the experiment have a number of implications. as far as containerized ML deployment is concerned:

**Latency Improvement:** The Docker containerized deployment has 27% reduced average latency than the. Non-containerized deployment (12.7 ms vs 17.36 ms). This such counterintuitive result can be explained by a number of factors:

- **Resource Isolation:** One is the fact that containers provide. enhanced CPU and memory separation, which decreases. other system processes interference.
- **Caching Effects:** Caching Effects Containerized pro-

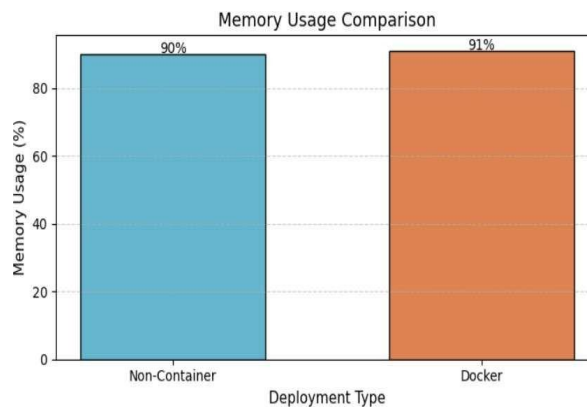


Fig. 4. Memory usage comparison (percentage of process memory)

cesses can also receive fewer context switches courtesy of namespace isolation.

- **Reduced Context Switching:** Processes in Containers use Namespaces may result in fewer context switches and improved isolation..

**Resource Utilization Trade-offs:** The containerized deployment is a bit heavier to the CPU (15 percent vs. 13 percent). memory (91% vs. 90.1) because of overhead of the container. runtime. This overhead includes. . .

- Container runtime process (containerd or docker- containerd)
- Network namespace virtualization
- Filesystem layer overhead (overlay2 storage driver)
- Additional kernel operations for namespace isolation

**Performance Stability:** The containerized deployment also. has much lower latency variability, the standard of which is 1.6 ms deviation as opposed to 5.84 ms non- containerized setup. This implies containers are a way of providing. more foreseeable performance, and this is especially crucial. where the production systems need constant service level. agreements (SLAs).

**Comparison with Previous Studies:** These are findings consistent with those of the past. has demonstrated to have low incurred costs with past research [9], [19] containers overhead at a cost of generating high benefits. Even though the reduction in latency is not seen in all. applications, the enhancement denotes that containerization. are even able to enhance the work of some kinds of applications.

**Practical Implications:** In the case of production ML systems, the slight resource overhead caused by containerization 2additional CPU and 1 percent additional memory- is readily warranted. by the substantial benefits:

- **Portability:** Containers ensure that the implementation of the use in a uniform fashion in a variety of environments. Development, testing and environments are examples of such. duction.
- **Reproducibility:** Container orchestration software permits the. scaling applications automatically fully.
- **Scalability:** Container orchestration tools permit the scaling of applications completely automatically.
- **Isolation:** There are several models that are deployed on the host and without dependency problems.
- **DevOps Integration:**

Table IV provides a qualitative comparison of the two deployment approaches.

**TABLE IV - QUALITATIVE COMPARISON OF DEPLOYMENT APPROACHES**

Aspect	Non-Containerized	Containerized
Setup Complexity	Low	Moderate
Environment Isolation	Poor	Excellent
Portability	Low	High
Reproducibility	Difficult	Easy
Scalability	Manual	Orchestration-ready
Performance Overhead	None (baseline)	Slight (2% CPU, 1% memory)

## CONCLUSION AND FUTURE WORK

### A. Conclusion

A detailed comparison of the study was made. of applying machine learning models in containerized and non- containerized forms. A healthcare data was used in the study. and a logistic regression model implemented with a REST API, and the response time, processor usage, and memory utilization, CPU utilization, and memory utilization. The performance comparison was conducted under the same experimental conditions to ensure a fair evaluation.

The mean latency of response was discovered to be reduced by 27 percent. in the containerised deployment versus the non-containerized deployment (12.7 ms vs. 17.36 ms).

- 1) The total overhead of resource consumption caused by containerization was negligible, with approximately 2% CPU overhead.

- 2) The model had a considerable implication on its performance stability. more effective in the containerized deployment, with the aggregate. latency variability decrease by 73%.
- 3) Both had a similar memory consumption behavior. deployment models, however, the containerized one. had a slightly more memory consumption.

## B. Future Work

There are also interesting directions proposed to this study.

- 1) **Larger Models and Deep Learning:** Larger Models and Deep Learning: other direction. is to investigate the use of deep learning models, e.g. CNNs. and Transformers, which makes use of GPU acceleration, are affected. by containerization.
- 2) **Orchestration Platforms:** Another direction is to explore how container orchestration tools, such as Kuber- netes, impact performance, including auto- scaling, load balancing, and the cost of using a service mesh.
- 3) **Multi-container Applications:** Other direction is to understand the functionality of the container orchestration tools, e.g., Kubernetes, impact performance, and auto- scaling. load balancing, and the expense of a service mesh usage.
- 4) **Resource-Constrained Environments:** An alternative direction refers to investigate the question of how containers can influence machine learn- or deployments of edge computing, such as CPU, memory, and networking capabilities serverless. ING inference platforms, such as container cost.
- 5) **Different Container Runtimes:** Another future direction is to investigate how different container runtimes such as containerd, CRI-O, and Podman influence machine learning performance, including comparisons between orchestration platforms such as Kubernetes.

## REFERENCES

- [1] C. Schroder, R. Boehm, and A. Lampe, "Comparison of Autoscaling Frameworks for Containerised Machine-Learning Applications in a Local and Cloud Environment," arXiv preprint arXiv:2311.18659, 2023.
- [2] T. M. T. Pham, K. Premkumar, M. Naili, and J. Yang, "Time to Retrain? Detecting Concept Drifts in Machine Learning Systems," arXiv preprint arXiv:2410.09190, 2024.
- [3] H. Zhou, H. Y. Chan, S. Y. Zhang, M. E. Lin, and J. Ni, "A Kubernetes Custom Scheduler Based on Reinforcement Learning for Compute- Intensive Pods," arXiv preprint arXiv:2601.13579, 2026.
- [4] S. Bharti, R. Gill, and S. Harnal, "Adaptive Task Offloading Framework for Serverless Edge Computing," in Proc. IEEE Int. Conf. on Disruptive Technologies, 2025.
- [5] P. Liu, J. Guitart, and A. Taherkordi, "Performance Characterization of Multi-Container Deployment Schemes for Online Learning Inference," in IEEE International Conference on Cloud Computing, 2023.
- [6] A. S. Iwashita and J. P. Papa, "An Overview on Concept Drift Learning," IEEE Access, vol. 4, pp. 1–15, 2016.
- [7] J. F. Kurian and M. Allali, "Detecting Drifts in Data Streams Using Kullback-Leibler Divergence Measure," Journal of Data, Information and Management, 2024.
- [8] Q. Waseem et al., "Drift Management in ML-Based IoT Device Classification: A Survey and Evaluation," International Journal on Advanced Science, Engineering and Information Technology, 2025.
- [9] P. Krogulski and T. Rak, "A Case Study on Virtual HPC Container Clusters and Machine Learning Applications," Applied Sciences, vol. 15, 2025.
- [10] A. Owen and K. Ajeigbe, "Containerization of Machine Learning Models," 2024.

- [11] D. Kutsa, "Using Machine Learning for Analyzing Performance Metrics in Kubernetes," International Journal of Scientific Research and Engineering Development, 2025.
- [12] R. R. Devarakonda, "An Integrated Approach for Scalable Deployment of Machine Learning Models in a Containerized Microservices Architecture," Journal of Emerging Trends and Novel Research, 2023.
- [13] Quadri et al., "Artificial Intelligence Performance Evaluation in Cloud Computing Environments," Journal of Artificial Intelligence Research, 2023.
- [14] RunAI, "Kubernetes Architecture for Data Science Workloads," RunAI Whitepaper, 2023.
- [15] "Understanding Data Drift and Concept Drift in Machine Learning Systems," Research Report, 2024.
- [16] "AI Performance Analysis in Cloud Infrastructure," Technical Report, 2023.
- [17] "Performance Analysis of AI Systems in Containerized Infrastructure," SSRN Electronic Journal, 2024.
- [18] "Machine Learning Deployment and Performance Optimization in Cloud Systems," International Journal of Scientific Research and Engineering Development, 2025.
- [19] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015, pp. 171–172.

**Abdullah**

He is presently serving as a researcher in the field of Cloud computing in Chandigarh University, India. His fields of interest are MLOps, containerization, and load balancing of distributed systems. He has been involved in some projects in relation to Docker and Scalable machine learning with Kubernetes.

**Nikhil Kumar**

Co-Author and investigator with special focus in the sector of Cloud computing and Dev Operations methodologies. His topics of interest are scalable containerization, the use and architecture in the Cloud of Continuous Deployment and Continuous Integration applications tools to the efficient deployment of applications. He is concerned with machine learning integration with Improvement of the cloud technologies. system reliability, system efficiency, and system reproducibility.