International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 07 July 2026
Page No: 4653-4660
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0611

CI/CD Pipeline for MLOps: Real-Time Fire Detection Using YOLO and Computer Vision

Sinchana S[1], H L Shilpa[2], Meghana HR[3]
[1,3]Department of Master of Computer Applications, PES College of Engineering, Mandya, 571401, Karnataka, India.
[2]Assistant Professor, Department of Master of Computer Applications, PES College of Engineering, Mandya, 571401, Karnataka, India.
Email ID: *sshivappa610@gmail.com[1], shilpa@pesce.ac.in[2], meghayashodha@gmail.com[3]*

Abstract
Fire accidents pose severe threats to human life, industrial infrastructure, public safety, and environmental resources. Conventional fire detection mechanisms rely primarily on smoke sensors, heat detectors, and manual surveillance, which often result in delayed alerts and limited spatial coverage. This paper presents a Continuous Integration and Continuous Deployment (CI/CD)-enabled Machine Learning Operations (MLOps) framework for real-time fire detection using a custom-trained You Only Look Once (YOLOv8) object detection model integrated with computer vision techniques. MLOps practices are employed to automate the training, deployment, monitoring, and maintenance lifecycle of the detection model, while the CI/CD pipeline ensures continuous integration, automated testing, and rapid, consistent deployment of application updates. The proposed system captures live and prerecorded video streams, performs frame-level preprocessing using OpenCV, and applies the trained YOLOv8 model to identify fire instances with high accuracy and low inference latency. GitHub, Jenkins, Docker, and AWS EC2 are integrated to automate the complete machine learning deployment workflow: whenever new code is pushed to the GitHub repository, Jenkins automatically triggers the pipeline, builds a Docker image, deploys the containerized application, and starts the fire detection service on a cloud instance. When fire is detected above a predefined confidence threshold, the system generates audio alarms, dispatches email notifications, and logs detection events for subsequent monitoring and analysis. Experimental evaluation demonstrates that the proposed framework achieves high detection accuracy, low inference latency, and reliable, repeatable automated deployment. The results confirm that combining MLOps discipline with CI/CD automation substantially improves the scalability, reproducibility, maintainability, and operational efficiency of real-time computer vision applications.

Keywords: *MLOps, CI/CD Pipeline, Fire Detection, YOLOv8, Computer Vision, Jenkins, Docker, AWS EC2, OpenCV, Deep Learning.*

1. Introduction
Fire outbreaks continue to pose significant threats to human life, industrial facilities, public infrastructure, and natural ecosystems. Incidents originating in manufacturing plants, warehouses, educational institutions, residential complexes, and forest regions can escalate within minutes, resulting in severe economic loss, environmental degradation, and irreversible safety hazards. Conventional fire detection technologies, including ionization and photoelectric smoke detectors and thermal sensors, typically depend on the physical diffusion of smoke or heat before an alarm is triggered. This dependency introduces a detection lag that can prove critical in fast-spreading fire scenarios, and it also limits monitoring coverage to enclosed spaces equipped with dedicated hardware. Consequently, there is a pressing need for vision-based detection mechanisms capable of identifying fire visually, at a distance, and in real time, without waiting for smoke or heat to reach a physical sensor. Advances in deep learning and computer vision have enabled object detection models to identify visual patterns associated with fire

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 07 July 2026
Page No: 4653-4660
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0611

and flame with high accuracy directly from video frames. Among the available architectures, the You Only Look Once (YOLO) family of single-stage object detectors is particularly well suited to real-time surveillance applications because it performs localization and classification in a single forward pass, achieving a favorable trade-off between detection accuracy and inference speed. However, deploying a trained deep learning model into a functioning, continuously available surveillance system introduces challenges that extend well beyond model training. Production systems must be packaged, deployed, monitored, and updated in a consistent and repeatable manner, and any change to the underlying model or application code must be validated and rolled out without manual intervention or service disruption. Machine Learning Operations (MLOps) addresses this gap by extending DevOps principles to the machine learning lifecycle, encompassing dataset versioning, automated training, continuous integration, containerized deployment, and post-deployment monitoring. When combined with a Continuous Integration and Continuous Deployment (CI/CD) pipeline, MLOps allows a computer-vision-based detection service to be updated, tested, and redeployed automatically whenever the underlying code, configuration, or model artifacts change. This paper presents a CI/CD-enabled MLOps framework for real-time fire detection that combines a custom-trained YOLOv8 object detection model with an automated deployment pipeline built on GitHub, Jenkins, Docker, and AWS EC2. The proposed system processes live video streams captured from an IP camera, performs fire detection using the trained YOLOv8 model, and, upon detecting fire above a defined confidence threshold, triggers an audio alarm and sends email notifications while logging the event for later review. The remainder of this paper is organized as follows. Section II reviews related work on MLOps, CI/CD automation, and computer-vision-based fire detection, and identifies the gap addressed by the proposed system. Section III describes the proposed methodology, including dataset preparation, model training, real-time detection, CI/CD automation, containerization, and cloud deployment. Section IV presents the experimental results and discusses their implications. Section V concludes the paper and outlines directions for future enhancement.

2. Related Work

Several studies have examined the application of MLOps, CI/CD practices, and cloud-native technologies to machine learning deployment workflows, though few have addressed real-time computer vision systems directly. Chikkala, Pathi, and Abhi [1] proposed an automated MLOps framework built on GitHub Actions and Microsoft Azure for training, validating, and deploying machine learning models. Their approach improved reproducibility and deployment efficiency but was designed for structured healthcare datasets and did not extend to real-time computer vision or continuous operational monitoring. Chatterjee and Mittal [2] investigated the integration of CI/CD practices within DevOps environments to streamline software deployment; while their pipeline reduced deployment delays and improved release quality, it did not address machine-learning-specific concerns such as model versioning, retraining, and drift monitoring. Ramesh, Pai, Birau, et al. [3] presented a broad analysis of cloud-native architectures, Docker, Kubernetes, and MLOps technologies for scalable machine learning systems; although the review offered valuable architectural insight, it remained largely theoretical and was not validated through a deployed real-time application. Niranjan and Mohana [4] proposed a Jenkins-based automation framework for machine learning workflows that automated training and deployment but stopped short of supporting real-time computer vision or continuous performance monitoring. Baitha, Kothari, Panda, Soorya, and Rajagopal [5] demonstrated the benefits of CI/CD automation for general software engineering, achieving improved deployment reliability and release velocity, but their study did not consider the additional lifecycle requirements introduced by machine learning models, such as dataset versioning and periodic retraining. Sinha and Nand [6] and Benjamin and Mathew [7] separately analyzed the causes of CI build failures using historical build logs and project metadata, concluding that dependency conflicts, configuration changes, and project complexity strongly influence build

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 07 July 2026
Page No: 4653-4660
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0611

reliability; neither study, however, considered machine learning pipelines or vision-based inference workloads. Zhou and Yu [8] proposed a generalized MLOps pipeline platform covering training, deployment, and monitoring, improving reproducibility and scalability without demonstrating a concrete real-time application. Cepuc, Ivanciu, and Botez [9] presented a CI/CD pipeline for deploying containerized applications on AWS using Jenkins, Docker, Ansible, and Kubernetes, improving deployment speed for conventional software but not addressing machine-learning-specific requirements. Soni [10] and Batista and Cruz [11] examined, respectively, end-to-end build-pipeline automation and CI/CD security vulnerabilities, the latter emphasizing access control and vulnerability scanning without incorporating machine-learning-based threat analysis. Velumani and Muthukamatchi [12] introduced a cloud-native defect-prediction framework using Random Forest models within containerized CI/CD pipelines, achieving strong predictive accuracy for software quality but not addressing real-time inference workloads. Saxena and Singh [13] explored Infrastructure-as-Code tools such as Terraform and AWS CloudFormation for automating cloud infrastructure deployment, improving consistency but lacking intelligent, model-aware monitoring. Reddy, Padal, Mayan, and Shanthamalar [14] investigated GitOps methodologies for improving CI/CD reliability through Git-based automation, again without incorporating machine learning techniques. Umesh, H G, K, Sharma, and P. [15] proposed an MLOps framework for predicting H-1B visa approvals using ensemble models over structured tabular data, demonstrating gains in scalability and automation that do not directly transfer to vision-based, streaming workloads. Nogueira [16] applied machine learning techniques to optimize CI/CD pipelines by predicting build failures, improving deployment reliability without addressing real-time AI deployment scenarios. Weiß, Navalgund, Stümpfle, Dettinger, and Weyrich [17] proposed a CI/CD pipeline supporting AI model deployment and over-the-air updates for software-defined vehicles, validated only under laboratory conditions. Nikolov and Aleksieva-Petrova [18], [19] developed a threat-modeling framework integrated with Jenkins-based DevSecOps pipelines, strengthening vulnerability detection but not incorporating machine-learning-based threat prediction. Batista and Cruz [20] further analyzed common CI/CD security vulnerabilities and proposed corresponding DevSecOps mitigations. Across these studies, a consistent gap emerges: existing MLOps and CI/CD frameworks are either validated on structured, non-visual datasets or remain theoretical proposals without a deployed, continuously monitored, real-time computer vision application. Furthermore, none of the reviewed frameworks specifically target live camera streams, which introduce additional challenges such as variable lighting, smoke interference, and reflective surfaces that can degrade detection reliability. The present work addresses this gap by combining a custom-trained YOLOv8 fire detection model with a fully automated CI/CD pipeline—spanning GitHub, Jenkins, Docker, and AWS EC2—and by validating the resulting system under real-time streaming conditions with integrated alert generation and operational monitoring.

3. Proposed Methodology

The proposed system couples a real-time, YOLOv8-based fire detection application with an automated CI/CD and MLOps deployment pipeline. Figure 1 illustrates the complete system architecture, which is organized into two coordinated workflows: a deployment workflow, in which code changes are propagated from the developer to a running cloud service, and an operational workflow, in which the deployed application continuously ingests video frames and performs fire detection. In the deployment workflow, developers push source-code and configuration changes to a GitHub repository, which automatically triggers the Jenkins pipeline. Jenkins checks out the latest code, builds a Docker image containing the updated fire detection application, and deploys the resulting container to an AWS EC2 instance. In the operational workflow, the hosted application connects to an IP camera stream, captures video frames using OpenCV, preprocesses each frame, and passes it to the trained YOLOv8 model. Based on the resulting confidence score, the system determines whether fire is present and, if the confirmation criteria are satisfied, raises an alert and

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 07 July 2026
Page No: 4653-4660
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0611

logs the detection event.

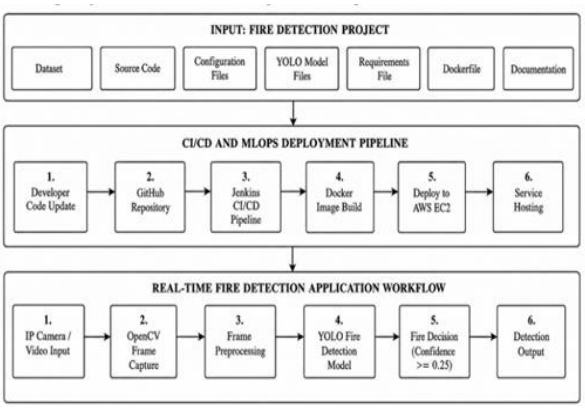

Figure 1 System Architecture

3.1.Dataset Preparation

The dataset used for training the YOLOv8 fire detection model consists of images depicting both fire and non-fire scenes, collected to represent a range of indoor, outdoor, industrial, and forest-like environments. Each image is annotated in YOLO format, in which every label file stores a class identifier together with normalized bounding-box coordinates corresponding to the visible fire regions. Annotation files were manually validated to verify accurate bounding-box placement, since incorrect localization labels would otherwise propagate directly into the trained model's detection accuracy. The complete dataset is partitioned into training, validation, and testing subsets to allow the model's learning progress to be tracked independently of its generalization performance on unseen images.

3.2.Data Preprocessing and Augmentation

Prior to training, every image is resized to a fixed resolution of 960 × 960 pixels, ensuring a consistent input shape for the YOLOv8 backbone regardless of the original image dimensions. The model is trained using a batch size of 16 over 50 epochs, values selected to balance convergence speed against the available computational resources. To improve generalization across the variable conditions encountered in real-world fire scenes, several data augmentation techniques are applied during training, including horizontal flipping, scaling, brightness adjustment, and mosaic augmentation. Horizontal flipping and scaling expose the model to varying object orientations and sizes, brightness adjustment simulates differing lighting and time-of-day conditions, and mosaic augmentation combines multiple training images into a single composite frame, encouraging the model to detect fire regions of varying scale against cluttered backgrounds. Collectively, these preprocessing steps help the trained model detect fire reliably under diverse lighting conditions, flame sizes, and background environments.

3.3.YOLOv8-Based Fire Detection Model Training

The core detection component of the proposed framework is a custom-trained YOLOv8 object detection model, fine-tuned specifically on the fire and non-fire image dataset described above. YOLOv8 was selected for this task because its single-stage detection architecture performs region proposal, feature extraction, and classification within one forward pass, giving it a substantial inference-speed advantage over two-stage detectors while retaining competitive localization accuracy. During training, the model learns to regress bounding-box coordinates around fire regions while simultaneously predicting a class confidence score, which is later used as the operational decision threshold during real-time inference. The trained model weights are versioned alongside the application source code so that every deployment corresponds to a specific, reproducible combination of model and application logic.

3.4.Real-Time Fire Detection Workflow

During real-time operation, OpenCV captures frames from an IP camera or a video file and forwards each frame to the trained YOLOv8 model for inference. The model predicts whether fire is present in the current frame; if the predicted class corresponds to fire and the associated confidence score meets or exceeds the operating threshold, the frame is marked as fire-positive. The fire detection rule is formally defined as:

$$\text{Fire Event} = \text{TRUE, if Class} = \text{Fire and Confidence} \geq 0.25 \quad (1)$$

To reduce spurious detections caused by transient reflections or momentary visual artifacts, the system requires fire to be detected continuously across a fixed number of consecutive frames before confirming the event. Only after this temporal

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 07 July 2026
Page No: 4653-4660
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0611

confirmation does the system trigger the associated alerts, which reduces the false-alarm rate without materially increasing detection latency.

3.5.CI/CD Pipeline Automation

The CI/CD pipeline automates the end-to-end deployment process for the fire detection application using GitHub, Jenkins, Docker, and AWS EC2. Application source code, YOLO model files, configuration files, and the Dockerfile are version-controlled in a GitHub repository. Whenever a developer pushes new code changes, Jenkins is automatically triggered and executes a sequence of pipeline stages: it checks out the latest code from GitHub, removes existing containers and unused resources to guarantee a clean build environment, builds a new Docker image containing the updated fire detection application, and deploys the resulting image as a container on an AWS EC2 instance. Following deployment, Jenkins verifies that the service has started successfully and monitors the application logs to confirm correct execution of the fire detection functionality. This automated sequence removes manual deployment steps, ensuring that every code change is built, tested, and released in a consistent and repeatable manner.

3.6.Docker Containerization

Docker is used to containerize the complete fire detection application, packaging the Python runtime environment, the trained YOLOv8 model file, the OpenCV library, application source code, and the alert-generation modules into a single deployable image. Containerization guarantees that the application executes identically across development, testing, and production environments, eliminating dependency conflicts that would otherwise arise from differing library versions or operating-system configurations. The resulting container is deployed on an AWS EC2 instance, which functions as the production server and hosts the fire detection service continuously. The containerized design also simplifies operational tasks such as restarting, updating, and scaling the application, since a new container can be built and swapped in without modifying the underlying host environment.

3.7.Cloud Deployment on AWS EC2

Following the Docker build stage, the containerized fire detection application is deployed to an AWS EC2 instance, which provides the compute resources required for continuous video processing. After deployment, the application establishes a connection to the IP camera stream, using Tailscale to create a secure network tunnel to the camera source, and begins continuous real-time fire monitoring. Hosting the application on EC2 allows the detection service to run independently of any single on-premises workstation, improving availability and enabling remote monitoring from any location with network access to the deployed instance.

3.8.Alert Generation and Monitoring

Upon confirming a fire event, the system generates an audio alarm to provide an immediate local warning and simultaneously dispatches an email notification to designated recipients, enabling a timely response even when no observer is physically present at the monitored site. Each detection event, together with its associated confidence score and timestamp, is recorded in an event log to support subsequent review, auditing, and performance analysis. Jenkins additionally monitors the deployed service's logs after each release, allowing operational issues to be identified quickly following a new deployment.

4. Results and Discussion

4.1.Fire Detection Performance

The proposed YOLOv8-based fire detection system was evaluated using fire and non-fire video samples spanning indoor, outdoor, industrial, and forest-like scenarios. The trained model achieved an overall detection accuracy of 95.2%, precision of 94.1%, recall of 93.6%, and an F1-score of 93.8%, as summarized in Table 1. These results indicate that the model reliably distinguishes fire from visually similar non-fire regions while maintaining a low false-alarm rate. The high precision score suggests that the model rarely raises an alert in the absence of genuine fire, which is particularly important in deployment scenarios where false alarms can erode operator trust in the system, while the correspondingly strong recall indicates that genuine fire events are detected with high consistency across the tested scenarios. The balanced F1-score confirms that the single-stage YOLOv8 architecture, combined with the augmentation strategy described in Section III-B, generalizes effectively across the varied lighting conditions, flame scales, and background clutter

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 07 July 2026
Page No: 4653-4660
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0611

present in the evaluation samples.

Table 1 Fire Detection Performance of the Proposed YOLOv8 Model

Performance Metric	Value (%)
Accuracy	95.2
Precision	94.1
Recall	93.6
F1-Score	93.8

Beyond the aggregate metrics, real-time inference performance was assessed using both live and prerecorded video streams. The deployed system sustained a throughput of approximately 28 to 32 frames per second, with an average fire identification latency of approximately 0.8 seconds measured from the first visible appearance of fire in the stream. This response time is well within the range required for early fire warning applications, where even a few seconds of earlier detection can materially improve the available response window.

4.2.CI/CD Deployment Performance

The CI/CD pipeline was evaluated by executing multiple code-update and deployment cycles and measuring the duration of each pipeline stage. As summarized in Table 2, source-code retrieval from GitHub required an average of 10 seconds, Docker image construction required approximately 95 seconds, container deployment required approximately 25 seconds, and service initialization required approximately 15 seconds, yielding a total end-to-end deployment time of approximately 2 to 3 minutes per release.

Table 2 CI/CD Pipeline Deployment Performance

Deployment Stage	Average Time
Source Code Retrieval	10 sec
Docker Image Build	95 sec
Container Deployment	25 sec
Service Initialization	15 sec
Total Deployment Time	2–3 min

This deployment turnaround is significant from an operational standpoint: a two-to-three-minute release cycle allows model updates, bug fixes, and configuration changes to reach the production environment with minimal delay and without manual intervention, in clear contrast to traditional deployment practices that require an operator to manually rebuild and restart the service. The dominant contributor to the total deployment time is the Docker image build stage, which reflects the cost of installing the Python runtime, OpenCV, and the YOLOv8 dependencies; this stage represents the most promising target for further optimization, for example through image layer caching or the use of a slimmer base image.

4.3.System Resource Utilization and Reliability

During continuous video processing, the deployed application exhibited stable resource consumption on the AWS EC2 instance, with CPU utilization ranging between 45% and 60% and memory usage ranging between 1.8 GB and 2.4 GB. The system maintained 99.5% container uptime and 99.2% application availability over the testing period, indicating that the Docker-based deployment provides consistent behavior across repeated restarts and updates. These reliability figures support the suitability of the proposed architecture for continuous, unattended operation, which is a practical requirement for safety-critical monitoring systems that cannot tolerate frequent service interruptions.

4.4.Discussion

The experimental results collectively demonstrate that combining a YOLOv8-based computer vision pipeline with CI/CD-enabled MLOps automation improves both detection performance and deployment reliability simultaneously, rather than treating them as separate concerns. The high detection accuracy and low inference latency confirm that the trained model is well suited to real-time surveillance, while the short, repeatable deployment cycle demonstrates that the operational pipeline can absorb frequent model or code updates without introducing service downtime. From a practical standpoint, this integration means that as new fire-scene data becomes available, the detection model can be retrained and pushed to production through the same automated pipeline used for ordinary application updates, without requiring a separate,

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 07 July 2026
Page No: 4653-4660
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0611

manually managed release process. Nevertheless, certain limitations remain. The model may occasionally generate false detections in the presence of strong light reflections, welding sparks, or other visually fire-like patterns, and detection accuracy can be affected by poor image quality, dense smoke, or partial camera occlusion. These limitations indicate that, while the system is well suited as an early-warning screening tool, it should be deployed as a complement to, rather than a replacement for, established fire-safety infrastructure and human oversight.

Conclusion

This paper presented an MLOps-based real-time fire detection framework that integrates a custom-trained YOLOv8 object detection model with a fully automated CI/CD deployment pipeline. The proposed system combines computer vision, containerization, cloud deployment, and automated monitoring to provide a reliable and scalable fire surveillance solution. By integrating Jenkins, Docker, GitHub, and AWS EC2, the framework streamlines the deployment lifecycle, reduces manual effort, and improves deployment consistency across releases. Experimental results demonstrate that the trained YOLOv8 model achieves effective real-time fire detection across indoor, outdoor, industrial, and forest-like environments, while the automated alert mechanisms—comprising audio alarms, email notifications, and event logging—enable a rapid response to detected incidents. Furthermore, the CI/CD pipeline supports continuous application and model updates without significant service interruption, completing full deployment cycles in approximately two to three minutes. The study highlights the practical value of integrating MLOps discipline with computer vision applications to improve the scalability, maintainability, and operational efficiency of safety-critical monitoring systems.

Future Enhancement

Future work will focus on strengthening system robustness and automation through several directions. Automated model retraining pipelines could incorporate newly collected fire-scene data without manual intervention, while dedicated smoke-detection modules could complement flame-based detection to improve early warning in low-visibility conditions. Migrating the deployment layer from a single EC2 instance to a Kubernetes-orchestrated environment would improve horizontal scalability and fault tolerance for multi-camera deployments. Incorporating IoT-based environmental sensors alongside the vision pipeline could enable multi-modal fire confirmation, reducing false-alarm rates further. Additional avenues include evaluating newer YOLO architectures and ensemble detection strategies, introducing model-drift detection to trigger retraining automatically, and integrating the alert pipeline directly with emergency-response systems to further enhance real-world applicability.

Acknowledgment

The authors express their sincere gratitude to the Department of Master of Computer Applications, faculty members, project guide, and technical staff at PES College of Engineering, Mandya, for their guidance, support, and encouragement throughout the completion of this research work.

References

[1]. Lokesh Kumar Chikkala, Nishanth Kumar Pathi, and Shinu Abhi, "Automating MLOps Pipelines with GitHub Actions and Azure: A Case Study on Diabetes Prediction," 2025.
[2]. Partha Sarathi Chatterjee and Harish Kumar Mittal, "Enhancing Operational Efficiency through the Integration of CI/CD and DevOps in Software Deployment," 2024.
[3]. G. Ramesh, T. Vaikunta Pai, Ramona Birau, et al., "A Comprehensive Review on Scaling Machine Learning Workflows Using Cloud Technologies and DevOps," IEEE Access, 2025.
[4]. Niranjan D. R. and Mohana, "Jenkins Pipelines: A Novel Approach to Machine Learning Operations (MLOps)," 2022.
[5]. Sanjay Baitha, Osho Kothari, Niharika Panda, Vijayakumar Soorya, and Shinu M. Rajagopal, "Streamlining Software Development: A Comprehensive Study on CI/CD Automation," IEEE, 2024.
[6]. Prameet Keshore Sinha and Parma Nand, "Study of Failures in Continuous Integration Environment," IEEE Conference Publication, 2024.

International Research Journal on Advanced Engineering Hub (IRJAEH)
e ISSN: 2584-2137
Vol. 04 Issue: 07 July 2026
Page No: 4653-4660
https://irjaeh.com
https://doi.org/10.47392/IRJAEH.2026.0611

[7]. Jetty Benjamin and Juby Mathew, "Study of the Contextual Factors of a Project Affecting the Build Performance in Continuous Integration," 2023.
[8]. Yue Zhou and Yue Yu, "Towards MLOps: Machine Learning Pipeline Platform," 2020.
[9]. Artur Cepuc, Iustin-Alexandru Ivanciu, and Robert Botez, "Implementation of a Continuous Integration and Deployment Pipeline for Containerized Applications in Amazon Web Services Using Jenkins, Ansible and Kubernetes," 2020.
[10]. Mitesh Soni, "End-to-End Automation on Cloud with Build Pipeline," 2015.
[11]. Luís Batista and Dinis Cruz, "Securing DevOps by Identifying the Most Common Vulnerabilities in CI/CD Pipelines," 2025.
[12]. M. V. Velumani and P. K. Muthukamatchi, "Cloud-Native Software Defect Prediction: Leveraging Machine Learning Models in Scalable CI/CD Pipelines," IEEE ICESC, 2025.
[13]. Adarsh Saxena and Sudhakar Singh, "DevOps Automation Pipeline Deployment with Infrastructure as Code (IaC)," 2024.
[14]. S. Reddy J., D. P. S. Padal, A. Mayan J., C. A., and J. J. Shanthamalar, "Efficient Application Deployment: GitOps for Faster and Secure CI/CD Cycles," IEEE, 2024.
[15]. Aditya Umesh, Harsha H G, Tarunkumar K, Sanchit Sharma, and Pooja P., "Integrating MLOps for Scalable and Accurate US Visa Approval Prediction," 2025.
[16]. Ana Filipa Nogueira, "Improving La Redoute's CI/CD Pipeline and DevOps Processes by Applying Machine Learning Techniques," 2018.
[17]. Matthias Weiß, Anish Navalgund, Johannes Stümpfle, Falk Dettinger, and Michael Weyrich, "An Open-Source CI/CD Pipeline for Variant-Rich Software-Defined Vehicles," 2025.
[18]. Lyuben Nikolov and Adelina Aleksieva-Petrova, "Framework for Integrating Threat Modeling into a DevOps Pipeline," 2023.
[19]. Lyuben Nikolov and Adelina Aleksieva-Petrova, "Framework for Integrating Threat Modeling into a DevOps Pipeline," 2023.
[20]. Luís Batista and Dinis Cruz, "Securing DevOps by Identifying the Most Common Vulnerabilities in CI/CD," 2025.