International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 08 Issue:11 | Nov-2024 SJIF Rating: 8.448 ISSN: 2582-3930

Intelligent CICD Pipeline Optimization Using Machine Learning for Reliable and Continuous Software Delivery

Sahil Vijaybhai Goyani[1],

[1]MSc. Computer Science & Ravensbourne University

Abstract – Continuous Integration and Continuous Delivery (CI/CD) pipelines have become fundamental to modern software engineering by enabling frequent integration, automated testing, and rapid deployment of software applications. However, increasing pipeline complexity, large test suites, build failures, resource-intensive execution, and deployment risks can reduce delivery efficiency and reliability. This research develops an intelligent CI/CD pipeline optimization framework using machine learning (ML) to improve the reliability, efficiency, and continuity of software delivery. The framework integrates historical pipeline data, build characteristics, test execution information, deployment logs, and operational metrics to support intelligent decision-making throughout the CI/CD lifecycle. Machine learning algorithms, including Random Forest, Support Vector Machine (SVM), Gradient Boosting, and XGBoost, are evaluated for build-failure classification, deployment-risk assessment, and pipeline performance optimization. The framework further incorporates intelligent test prioritization, build-time optimization, automated failure analysis, and rollback support to reduce unnecessary pipeline execution and accelerate software delivery. Performance is assessed using classification metrics such as accuracy, precision, recall, F1-score, and ROC-AUC, together with operational indicators including build time, test execution time, deployment time, pipeline success rate, deployment failure rate, Mean Time to Recovery (MTTR), resource utilization, and release frequency. The research establishes an integrated approach in which machine learning complements conventional CI/CD automation with data-driven pipeline intelligence. The developed framework provides a scalable foundation for improving software delivery reliability while reducing execution overhead and supporting faster and more consistent releases in cloud-based DevOps environments.

Key Words: CI/CD, Machine Learning, DevOps, Pipeline Optimization, Continuous Integration, Continuous Delivery, Build Failure, Test Prioritization, Deployment Reliability, Software Delivery Automation.

1.INTRODUCTION

1.1 Background of CI/CD and DevOps

Continuous Integration and Continuous Delivery (CI/CD) have become fundamental practices in modern software engineering, enabling development teams to integrate code changes, perform automated testing, and deliver software through structured and repeatable workflows. Continuous Integration focuses on frequently integrating source-code changes into a shared repository, where automated build and testing processes provide rapid feedback on software quality and integration status [1], [2]. Continuous Delivery extends this process by maintaining validated software in a release-ready state through automated delivery activities [4].

CI/CD is closely associated with DevOps, which integrates development and operational activities through automation, collaboration, continuous feedback, and shared responsibility. DevOps practices reduce the separation between software development and deployment activities and support shorter and more consistent delivery cycles [8], [10]. Within this environment, CI/CD pipelines automate major stages such as source-code integration, build execution, testing, quality verification, deployment, and operational monitoring.

As software systems become larger, CI/CD pipelines generate increasing volumes of build logs, test results, execution times, code-change information, deployment records, and system-monitoring data. Although conventional pipelines automate these activities effectively, their operation is generally based on predefined rules and fixed execution sequences. This can result in unnecessary test execution, longer pipeline duration, inefficient resource utilization, and delayed identification of build or deployment failures [2], [6].

Machine learning provides an opportunity to utilize historical CI/CD data for more adaptive pipeline operation. Previous studies have demonstrated the application of data-driven techniques to CI build-failure analysis and test-case prioritization [11], [18]. Building on this direction, this research integrates machine learning with the CI/CD lifecycle to support build analysis, intelligent test prioritization, deployment-risk classification, failure analysis, and pipeline optimization. The resulting framework aims to enhance software delivery reliability while reducing execution overhead and supporting efficient continuous delivery.

1.2 Challenges in Conventional CI/CD Pipelines

Although conventional CI/CD pipelines automate software integration, testing, and deployment, their increasing complexity introduces several operational limitations. Most traditional pipelines follow predefined workflows in which builds and test suites are executed according to fixed rules, regardless of the characteristics or impact of individual code changes. As software repositories and test suites expand, this approach can increase pipeline execution time and computational overhead [2], [6].

A major challenge is the efficient management of build and testing activities. Build failures can interrupt delivery workflows and require additional developer effort for investigation and recovery. Similarly, executing extensive regression-test suites after every code change can consume substantial time and infrastructure resources. Research on CI environments has therefore investigated build-failure analysis and test-case prioritization as mechanisms for improving pipeline efficiency [11], [16], [17].

Conventional pipelines also have limited capability to utilize accumulated historical information. Build logs, test outcomes, code-change records, deployment events, and execution metrics contain useful patterns, but rule-based pipelines generally do not adapt their execution decisions based on these patterns. This limitation becomes more significant in cloud-based and distributed software

environments where workload, application dependencies, and resource requirements continuously vary [4], [7].

Another important concern is delivery reliability. Failures occurring during build, testing, or deployment can increase recovery time and disrupt continuous software delivery. Consequently, CI/CD optimization requires more than task automation; it requires mechanisms capable of analysing pipeline behaviour and supporting informed execution decisions. These limitations provide the basis for integrating machine learning into CI/CD workflows to improve test prioritization, failure analysis, deployment-risk assessment, resource utilization, and overall pipeline efficiency [12], [18].

1.3 Role of Machine Learning in Software Delivery Automation

Machine Learning (ML) extends conventional software delivery automation by enabling CI/CD pipelines to utilize historical and real-time operational data for more informed execution decisions. Build histories, code-change characteristics, test outcomes, execution durations, deployment records, and system logs can be transformed into measurable features for identifying patterns associated with pipeline performance and failures [11], [12].

Within CI/CD environments, ML can support build-failure classification, intelligent test prioritization, deployment-risk assessment, failure analysis, and resource optimization. For example, ML-based test prioritization can use previous test outcomes and contextual pipeline information to determine which tests should receive higher execution priority, reducing unnecessary testing overhead while maintaining effective verification [18], [19]. Similarly, build information can be analysed to distinguish execution patterns associated with successful and unsuccessful pipeline runs [11].

The integration of ML therefore changes CI/CD from a predominantly rule-based automation process into a more data-driven and adaptive delivery environment. In this research, ML models are incorporated into the pipeline to analyse operational characteristics and support optimization decisions across build, testing, and deployment stages. The approach aims to reduce pipeline execution overhead, improve resource utilization, accelerate feedback, and maintain reliable continuous software delivery.

1.4 Problem Statement

Despite extensive automation in modern CI/CD environments, conventional pipelines largely depend on predefined rules and fixed execution sequences for building, testing, and deploying software. As application complexity, development frequency, and test-suite size increase, these pipelines may experience longer execution times, unnecessary test runs, inefficient resource utilization, build failures, and delayed recovery from unsuccessful deployments [6], [11], [17].

Existing studies have explored individual problems such as CI build-failure analysis and machine-learning-based test-case prioritization [11], [12], [18]. However, handling these activities independently limits the ability to optimize the software delivery pipeline as an integrated process. Build behaviour, testing outcomes, deployment information, and operational logs can collectively provide valuable information for improving pipeline decisions.

Therefore, this research addresses the need for an integrated machine-learning-assisted CI/CD framework that utilizes pipeline data to support build analysis, intelligent test prioritization, deployment-risk classification, failure analysis, and resource optimization. The central research problem is to determine how machine learning can improve CI/CD pipeline efficiency and reliability while reducing execution overhead and supporting continuous software delivery.

1.5 Research Aim and Objectives

The primary aim of this research is to develop an intelligent machine-learning-assisted CI/CD pipeline optimization framework that improves the efficiency, reliability, and continuity of software delivery by utilizing build, testing, deployment, and operational pipeline data.

To achieve this aim, the research establishes the following objectives:

1. To analyse CI/CD pipeline data, including build status, test outcomes, execution time, deployment records, and resource utilization.
2. To develop and evaluate machine learning models for build-failure classification and deployment-risk assessment.
3. To implement an intelligent test prioritization mechanism that reduces unnecessary test execution and improves pipeline processing efficiency.
4. To optimize pipeline execution time and computational resource utilization through data-driven decision mechanisms.
5. To integrate failure analysis and automated rollback support for improving deployment reliability and recovery performance.
6. To evaluate the developed framework using accuracy, precision, recall, F1-score, ROC-AUC, build time, test execution time, deployment time, pipeline success rate, failure rate, MTTR, and resource utilization.

2. LITERATURE REVIEW

2.1 Evolution of Continuous Integration and Continuous Delivery

Continuous Integration (CI) emerged as a software-development practice for integrating code changes frequently and validating them through automated build and testing processes. The approach reduced dependence on late-stage integration and enabled developers to identify integration problems earlier in the development lifecycle. With increasing adoption of automated development infrastructure, CI evolved from basic build automation into an important component of continuous software engineering [1], [2].

The progression from CI to Continuous Delivery (CD) extended automation beyond source-code integration and testing. CD introduced structured processes for preparing validated software for release through automated packaging, configuration, quality checks, and deployment-related activities. Chen [4] highlighted that continuous delivery can support shorter release cycles and more frequent delivery, although its implementation introduces technical and organizational challenges. Further studies have demonstrated that organizations adopt different paths toward continuous deployment depending on their system architecture, testing capability, and automation maturity [5], [6].

The emergence of DevOps further strengthened this evolution by connecting development and operational activities through automation, collaboration, monitoring, and continuous feedback [8], [10]. Consequently, modern CI/CD pipelines have developed into end-to-end software delivery environments that integrate version control, automated builds,

regression testing, release management, deployment, and operational monitoring.

However, increasing pipeline scale has also created challenges involving build failures, extensive test execution, computational overhead, and complex deployment decisions [11], [17]. Recent research has therefore moved beyond basic pipeline automation toward data-driven CI/CD optimization, including machine-learning-based build analysis and intelligent test prioritization [12], [18], [19]. This progression establishes the foundation for incorporating machine learning into CI/CD pipelines to improve software delivery efficiency and reliability.

2.3 Machine Learning Applications in Software Engineering

Machine learning (ML) has become increasingly relevant to software engineering because development and operational environments continuously generate structured and unstructured data, including source-code changes, build records, test results, execution metrics, defect histories, and system logs. ML techniques can analyse these datasets to identify relationships that are difficult to capture through fixed rule-based approaches, supporting more data-driven software engineering processes [22], [30].

Supervised learning algorithms such as Random Forest, Support Vector Machine (SVM), Gradient Boosting, and XGBoost are particularly applicable where historical software data contain labelled outcomes. These algorithms provide different mechanisms for classification and pattern identification and have been widely applied to complex data-analysis problems [26]–[29]. Within software delivery environments, such models can use pipeline characteristics to distinguish successful and failed executions and evaluate operational conditions associated with software delivery performance.

ML applications are also relevant to software testing and quality assurance. Regression testing can become computationally expensive as software systems and test suites expand; therefore, test selection and prioritization have been extensively investigated to improve testing efficiency [31], [32]. More recent CI-oriented studies have incorporated machine learning and contextual pipeline information to prioritize test cases according to their relevance and previous execution behaviour [18], [19].

In this research, these ML capabilities are integrated with CI/CD pipeline data to support build-failure classification, intelligent test prioritization, deployment-risk assessment, and pipeline optimization. Rather than treating ML as an independent analytical component, the models operate within the software delivery workflow to assist pipeline-level decisions and improve delivery efficiency and reliability.

2.4 Intelligent Build and Test Optimization

Build and testing stages represent major contributors to CI/CD pipeline execution time and computational workload. In conventional pipelines, each code change may trigger predefined build operations and extensive regression-test suites, even when only a limited part of the application has been modified. As project size and test-suite complexity increase, this approach can delay developer feedback and consume unnecessary processing resources [16], [17].

Intelligent build optimization uses historical pipeline information, including code-change characteristics, previous build outcomes, execution duration, and failure records, to support more efficient build decisions. Machine-learning techniques have demonstrated the ability to analyse CI build histories and classify execution conditions associated with build failures [11], [12]. Such information can help allocate computational effort more effectively and identify pipeline executions requiring greater attention.

Similarly, intelligent test optimization focuses on selecting and prioritizing tests according to their relevance to the current software change. Traditional regression-testing research established test selection and prioritization as effective approaches for improving testing efficiency [31]–[34]. Recent studies have extended these concepts by applying machine learning and contextual CI information to determine test execution priority dynamically [18], [19].

Integrating build analysis and test prioritization within the same CI/CD workflow provides a stronger basis for reducing unnecessary execution while maintaining software verification. Accordingly, this research utilizes ML-assisted build analysis together with intelligent test prioritization to reduce build duration, test execution time, and resource consumption, while maintaining reliable software delivery.

2.5 Deployment Reliability and Failure Management

Deployment reliability is a critical requirement of continuous software delivery because failures introduced during release can affect system availability, service quality, and recovery time. Although CI/CD pipelines automate deployment activities, unsuccessful releases may still result from configuration inconsistencies, application dependencies, infrastructure conditions, or changes that behave differently in production environments. Continuous delivery research therefore emphasizes reliable automation, rapid feedback, and controlled release processes as important elements of deployment management [4], [6].

Conventional failure management generally relies on predefined monitoring rules, alerts, and rollback conditions. While these mechanisms provide operational protection, they have limited ability to utilize relationships across historical build, test, deployment, and monitoring data. DevOps environments generate substantial operational information that can be analysed to improve the identification and management of unsuccessful software releases [7], [10].

Machine learning provides an additional analytical layer by evaluating deployment-related features such as build status, test outcomes, change characteristics, deployment duration, failure history, and resource utilization. These features can support deployment-risk classification and assist in identifying conditions associated with unreliable releases. When combined with automated rollback and recovery mechanisms, such analysis can reduce the time required to respond to deployment failures and improve continuity of software delivery.

Accordingly, this research incorporates deployment-risk assessment, failure analysis, and rollback support within the intelligent CI/CD framework. Deployment reliability is evaluated using operational indicators including deployment failure rate, pipeline success rate, deployment time, and Mean Time to Recovery (MTTR), providing a measurable basis for assessing improvements in continuous software delivery.

2.6 Research Gaps in Existing CI/CD Optimization Approaches

Existing research has established significant advances in CI/CD automation, build-failure analysis, regression testing,

and machine-learning-assisted test prioritization. However, much of the literature addresses these activities as individual optimization problems. Build-failure analysis primarily focuses on identifying conditions associated with unsuccessful builds [11], [12], while test-oriented studies concentrate on test selection, prioritization, and reduction of regression-testing overhead [17]–[19].

A further limitation is that conventional CI/CD optimization commonly relies on predefined execution rules or optimizes only a specific pipeline stage. This restricts the combined use of code-change information, build histories, test outcomes, deployment records, execution metrics, and operational logs across the complete software delivery lifecycle. Similarly, deployment reliability and recovery mechanisms are often considered separately from build and testing optimization [6], [7].

These limitations establish a research gap for an integrated, data-driven CI/CD optimization framework in which machine learning supports multiple pipeline stages within a unified workflow. This research addresses the identified gap by combining build-failure classification, intelligent test prioritization, deployment-risk assessment, failure analysis, resource optimization, and rollback support. The framework is evaluated using both ML performance measures and operational CI/CD indicators, enabling the impact on pipeline efficiency and software delivery reliability to be assessed collectively rather than at an isolated stage.

3. RESEARCH METHODOLOGY

3.1 Research Framework and System Architecture

This research develops an intelligent machine-learning-assisted CI/CD framework in which operational pipeline data are continuously collected, processed, analysed, and used to support software delivery decisions. The framework integrates four primary layers: CI/CD data acquisition, data processing, machine-learning analysis, and pipeline optimization. This structure enables build, testing, and deployment activities to be evaluated within a unified workflow rather than as independent processes.

Pipeline data are obtained from source-code commits, build histories, test execution records, deployment logs, and operational monitoring metrics. The collected records are cleaned, encoded, normalized where required, and transformed into model-ready features. Relevant inputs include code-change size, build duration, previous build status, test execution time, test failure history, deployment duration, and resource utilization. Historical CI information has previously demonstrated value for build-failure analysis and intelligent test prioritization [11], [12], [18].

The processed dataset is supplied to the ML layer, where Random Forest, SVM, Gradient Boosting, and XGBoost are evaluated. The models support three principal functions: build-failure classification, test prioritization, and deployment-risk classification. Their outputs are subsequently integrated into the optimization layer to improve test execution decisions, pipeline processing efficiency, resource utilization, and failure-management actions.

The complete research architecture is represented in Figure 1.

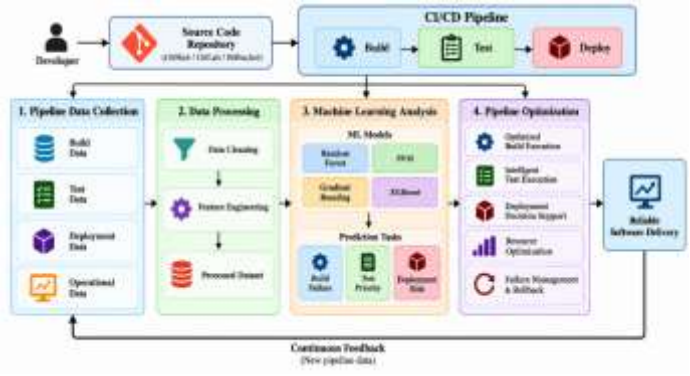


Figure 1. Architecture of the Intelligent Machine-Learning-Assisted CI/CD Pipeline Optimization Framework

3.2 CI/CD Pipeline Data Collection and Preprocessing

The developed framework collects operational data generated throughout the build, testing, and deployment stages of the CI/CD pipeline. The dataset includes build status, build duration, code-change information, test execution time, test outcomes, deployment status, deployment duration, failure records, and computational resource utilization. These variables provide measurable information describing both pipeline behaviour and software delivery performance. Historical CI data have been shown to provide useful information for build-failure analysis and test prioritization [11], [12], [18].

Before model development, the collected pipeline records are processed to improve data consistency and analytical quality. Duplicate entries are removed, missing numerical values are handled using median imputation, and categorical attributes such as build and deployment status are converted into numerical representations. Continuous variables with different measurement ranges are standardized where required, particularly for algorithms sensitive to feature scale. For a numerical feature x, standardization is performed as:

$$z = \frac{x - \mu}{\sigma} \dots\dots\dots\dots\dots (1)$$

where x represents the original feature value, μ is the feature mean, and σ is its standard deviation.

Following preprocessing, the dataset is organized into input features and target variables. Input features include code-change size, previous build status, build duration, test history, test execution time, deployment duration, and resource utilization. Target variables are defined according to the analytical task, including build success/failure and deployment risk classes, while test-related information is used for intelligent test prioritization. The resulting processed dataset provides a consistent input for feature engineering and ML model development in subsequent stages.

3.3 Feature Engineering and Selection

Feature engineering was performed to transform the preprocessed CI/CD records into meaningful variables representing the operational behaviour of the software delivery pipeline. The selected features were derived from code changes, build execution, testing history, deployment activity, and resource utilization, as these factors provide relevant information for analysing pipeline efficiency and failure conditions [11], [12], [18].

The primary features included code-change size, number of modified files, build duration, previous build status, test execution time, previous test failures, deployment duration,

CPU utilization, and memory utilization. In addition, historical failure information was represented through a failure-rate feature calculated over previous pipeline executions:

$$FR = \frac{N_f}{N_t} \dots\dots\dots\dots (2)$$

where FR represents the historical failure rate, N_f is the number of failed executions, and N_t is the total number of observed executions.

Feature selection was then applied to remove redundant or weakly informative variables before model training. Correlation analysis was used to identify highly related numerical variables, while feature-importance scores from tree-based models were used to evaluate the contribution of individual attributes. This process retained features that provided useful information for build-failure classification, test prioritization, and deployment-risk classification, while reducing unnecessary dimensionality and computational overhead.

The resulting feature set was subsequently used as the common input for the Random Forest, SVM, Gradient Boosting, and XGBoost models, enabling consistent comparison of model performance during experimental evaluation [26]–[29].

3.4 Machine Learning Model Development

The processed CI/CD dataset was used to develop machine learning models for build-failure classification and deployment-risk classification, while test-related features were used to support intelligent test prioritization. Four supervised learning algorithms—Random Forest (RF), Support Vector Machine (SVM), Gradient Boosting (GB), and XGBoost—were selected because they provide different learning mechanisms for handling nonlinear relationships and complex feature interactions [26]–[29].

The dataset was divided into 80% training data and 20% testing data using stratified sampling to preserve the distribution of successful and failed pipeline executions. Model parameters were tuned using five-fold cross-validation on the training set. Feature scaling was applied to SVM, while tree-based models were trained using the engineered feature set without scale-dependent transformation.

For a pipeline instance represented by the feature vector

$$X = [x_1, x_2, \dots\dots, x_n] \dots\dots\dots\dots (3)$$

the classification function is expressed as:

$$\hat{y} = f(X; \theta) \dots\dots\dots\dots (4)$$

where X represents the CI/CD input features, f denotes the trained ML model, (θ) represents the learned model parameters, and $(\hat{y})$ is the resulting pipeline class.

Model performance was evaluated using accuracy, precision, recall, F1-score, and ROC-AUC. In addition to classification performance, the selected model was assessed through CI/CD operational measures such as build time, test execution time, deployment failure rate, and resource utilization. This combined evaluation enables selection of the model that provides reliable classification performance while contributing to practical pipeline optimization.

3.5 Intelligent Pipeline Optimization Mechanism

The intelligent pipeline optimization mechanism integrates the outputs of the trained ML models with CI/CD execution decisions. Instead of executing every pipeline stage through fixed rules alone, the mechanism evaluates build characteristics, test history, deployment conditions, and resource information to determine appropriate actions during software delivery. This enables pipeline optimization while maintaining the required verification processes [11], [18].

The mechanism operates across three main stages. First, build analysis identifies execution conditions associated with build failure and supports appropriate allocation of pipeline resources. Second, intelligent test prioritization ranks test cases using code-change information, historical test outcomes, and execution characteristics so that higher-relevance tests receive earlier execution priority [17]–[19]. Third, deployment-risk classification evaluates validated pipeline information before deployment and provides decision support for release execution.

An optimization objective is defined to balance execution efficiency, resource consumption, and pipeline failures:

$$J = \omega_1 T + \omega_2 R + \omega_3 F \dots\dots (5)$$

where J represents the overall optimization objective, T is normalized pipeline execution time, R represents normalized resource utilization, F represents the pipeline failure measure, and $\omega_1, \omega_2, \omega_3$ are weighting coefficients satisfying $\omega_1 + \omega_2 + \omega_3 = 1$ Lower values of J indicate more efficient pipeline operation under the defined criteria.

The optimization mechanism also incorporates failure management and rollback support. When an execution does not satisfy defined deployment conditions, the pipeline can retain the stable release state while failure information is recorded for subsequent analysis. New build, test, and deployment records are returned to the data layer, creating a continuous feedback mechanism for later model updates. This closed-loop structure enables the CI/CD framework to improve execution efficiency while supporting reliable and continuous software delivery.

3.6 Experimental Setup and Evaluation Metrics

The experimental environment was structured to evaluate both the machine-learning performance and operational effectiveness of the intelligent CI/CD framework. The processed pipeline dataset was divided into 80% training and 20% testing subsets, while five-fold cross-validation was applied to the training data during model development. Random Forest, SVM, Gradient Boosting, and XGBoost were evaluated under the same feature set and data partitions to maintain a consistent comparison [26]–[29].

Classification performance was assessed using accuracy, precision, recall, F1-score, and ROC-AUC. The principal metrics were calculated as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \dots\dots (6)$$

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}$$

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recal}} \dots\dots (7)$$

where TP, TN, FP, and FN represent true-positive, true-negative, false-positive, and false-negative classifications, respectively.

The operational impact of the framework was evaluated by comparing the conventional CI/CD configuration with the ML-assisted configuration. The principal pipeline indicators were build time, test execution time, deployment time, pipeline success rate, deployment failure rate, Mean Time to Recovery (MTTR), and computational resource utilization.

For each operational indicator, percentage improvement was determined as:

$$\text{Improvement (\%)} = \frac{V_{baseline} - V_{optimized}}{V_{baseline}} \times 100 \ldots\ldots (8)$$

for metrics where a reduction represents improvement, such as execution time, failure rate, and MTTR. The combined evaluation provides a measurable basis for determining whether the developed framework improves both ML classification performance and practical CI/CD delivery efficiency.

Table 1. Evaluation Metrics Used in the Experimental Analysis

Category	Metrics
ML Performance	Accuracy, Precision, Recall, F1-score, ROC-AUC
Pipeline Efficiency	Build Time, Test Execution Time, Deployment Time
Reliability	Pipeline Success Rate, Deployment Failure Rate, MTTR
Resource Performance	CPU and Memory Utilization

4. INTELLIGENT CI/CD PIPELINE FRAMEWORK

4.1 Architecture of the Intelligent CI/CD Pipeline

The intelligent CI/CD pipeline developed in this research integrates source-code management, automated build and testing, machine-learning analysis, deployment control, and continuous monitoring within a unified software delivery workflow. Conventional continuous delivery relies extensively on automation but can face increasing operational and adoption challenges as delivery environments become more complex [6]. ScienceDirect

When a new code change enters the pipeline, build and test execution generates information such as code-change characteristics, build status, test outcomes, execution duration, deployment records, and resource utilization. These data are transferred to the ML analysis layer, where they are used to support build-failure classification, test prioritization, and deployment-risk assessment. Previous research demonstrates that ML can effectively exploit historical CI information for build-failure analysis [12] and that contextual information such as test execution time, size, complexity, and failure history can improve ML-assisted test prioritization [18].

The operational architecture follows the sequence:

Code Commit → Build → Test → ML Analysis → Deploy → Monitoring

The ML layer operates as an integrated decision-support component rather than as an independent analytical stage. Build information supports failure classification, test information determines execution priority, and validated pipeline characteristics are used for deployment-risk classification. The use of contextual information for CI test prioritization is supported by recent research demonstrating that ML-based approaches can incorporate execution-related information while maintaining fault-detection performance [18].

4.2 Build-Time Optimization

Build execution is a major contributor to the overall duration of a CI/CD pipeline, particularly when frequent code changes repeatedly trigger complete build processes. Conventional pipelines generally follow predefined build sequences without considering the characteristics of individual commits or historical execution behaviour. Previous studies have shown that CI build histories contain useful information for analysing build outcomes and improving build-related decisions [2], [11].

In the developed framework, build-time optimization uses code-change size, number of modified files, previous build status, historical build duration, failure history, and resource utilization as input features. These features are analysed by the trained ML model to classify the build condition and support the allocation of execution resources. ML-based analysis of historical CI information has demonstrated its usefulness for identifying build-failure patterns and reducing unnecessary CI execution overhead [11], [12].

Build-time improvement is quantified as:

$$BTR(\%) = \frac{T_B - T_O}{T_B} \times 100 \ldots\ldots (9)$$

where BTR represents Build-Time Reduction, T_B is the average build time of the conventional pipeline, and T_O is the average build time after applying the intelligent optimization mechanism.

The optimization mechanism retains required build and verification activities while reducing avoidable processing overhead and improving resource allocation. Build duration, build success rate, and computational resource utilization are subsequently used to evaluate the effectiveness of this mechanism. This allows the contribution of intelligent build optimization to overall CI/CD efficiency to be measured independently before its combined effect with testing and deployment optimization is assessed.

4.3 Intelligent Test Selection and Prioritization

Regression testing is essential for verifying software changes; however, executing the complete test suite after every code modification can significantly increase CI/CD pipeline duration and computational workload. Test selection and prioritization provide an effective mechanism for reducing this overhead by identifying relevant tests and determining an appropriate execution order [31]–[34]. In CI environments, contextual and historical test information can further improve prioritization decisions [17], [18].

In the developed framework, each test case is evaluated using code-change relevance, previous execution outcomes, failure history, execution time, and test frequency. A test-priority score is calculated as:

$$TPS_i = \omega_1 C_i + \omega_2 F_i + \omega_3 H_i - \omega_4 E_i \ldots\ldots (10)$$

where TPS_i is the priority score of test case i, C_i represents its relevance to the current code change, F_i represents recent failure information, H_i represents historical test importance, and E_i represents normalized execution time. The weighting coefficients $\omega_1 - \omega_4$ control the contribution of each factor.

Tests with higher TPS_i values are executed earlier, while low-relevance tests can be deferred according to the defined pipeline verification requirements. This approach reduces unnecessary test-processing overhead without eliminating essential software validation. ML-assisted test prioritization has similarly demonstrated the value of incorporating contextual CI information and historical test behaviour into test-ordering decisions [18], [19].

The effectiveness of the mechanism is evaluated using test execution time, number of executed tests, early failure identification rate, and overall pipeline duration. These measures determine whether intelligent test selection and prioritization improve CI/CD efficiency while maintaining reliable software verification.

4.4 Deployment Risk Assessment

Deployment is a critical stage of the CI/CD lifecycle because an unsuccessful release can affect application availability, service stability, and recovery time. Continuous delivery therefore requires controlled deployment mechanisms that consider software quality and operational conditions before a release progress to the production environment [4], [6].

In the developed framework, deployment risk is assessed using information generated during earlier pipeline stages. The input features include build status, test outcomes, code-change size, previous deployment failures, deployment duration, and resource-utilization metrics. Combining development and operational information enables deployment decisions to consider the overall condition of the pipeline rather than relying on a single verification result. Such integration is consistent with DevOps practices that emphasize automated delivery, monitoring, and rapid feedback across development and operational activities [7], [10].

For each deployment instance, a normalized risk score is defined as:

$$\mathrm{DRS} = \sum_{j=1}^{n} \omega_j x_j \dots\dots (11)$$

where DRS represents the Deployment Risk Score, x_j denotes the normalized value of deployment-related feature j, and ω_j represents its assigned importance. Higher values indicate conditions requiring greater deployment control.

Based on the resulting risk class, the pipeline supports release approval, temporary hold, or rollback-related actions. The effectiveness of this mechanism is evaluated using deployment success rate, deployment failure rate, deployment duration, and Mean Time to Recovery (MTTR). This approach integrates deployment assessment with the preceding build and testing stages, supporting more reliable continuous software delivery.

4.5 Failure Detection and Root-Cause Analysis

Failure detection is integrated into the developed CI/CD framework to identify unsuccessful conditions across the build, testing, and deployment stages. Pipeline failures can originate from code changes, failed tests, configuration errors, dependency issues, resource limitations, or deployment-related conditions. Historical build and execution data provide useful information for distinguishing failure-related patterns in continuous integration environments [11], [12].

When a pipeline failure occurs, the framework combines the ML classification output with build logs, failed-test records, code-change information, deployment logs, and resource metrics. The detected event is assigned to a failure category, allowing the analysis process to narrow the probable source rather than treating every unsuccessful pipeline run identically.

The failure categories used in this research are represented as:

$$F_c \in \{F_{code}, F_{build}, F_{test}, F_{config}, F_{resource}, F_{deploy}\} \dots\dots (12)$$

where F_c denotes the identified failure category corresponding to code, build, testing, configuration, resource, or deployment-related failure.

Root-cause analysis then correlates the identified category with the pipeline stage, recent code changes, historical failure records, and associated log events. This information supports faster isolation of the affected pipeline component and provides the basis for appropriate recovery or rollback actions. Continuous monitoring and feedback are important DevOps capabilities for maintaining reliable software delivery and reducing operational disruption [7], [10].

The effectiveness of failure management is evaluated using the failure identification rate, pipeline failure rate, and Mean Time to Recovery (MTTR). The resulting failure records are also retained in the pipeline dataset, enabling subsequent model updates and continuous improvement of the intelligent CI/CD framework.

4.6 Automated Rollback and Recovery Mechanism

The automated rollback and recovery mechanism is incorporated into the intelligent CI/CD framework to maintain software availability when a deployment does not satisfy the defined reliability conditions. Continuous delivery requires release processes that can respond rapidly to unsuccessful deployments and restore a stable application state with minimal operational interruption [4], [6].

Following deployment, the framework monitors deployment status, application health, failure events, and resource conditions. When critical failure conditions are detected, the pipeline prevents further release progression and initiates rollback to the most recent validated software version. Failure information identified through the root-cause analysis stage is retained to support recovery actions and subsequent pipeline analysis. Automated monitoring and rapid feedback are consistent with DevOps practices for improving deployment and operational reliability [7], [10].

The rollback decision is represented as:

$$R = \begin{cases} 1 & DRS \geq \tau \text{ or critical failure detected} \\ 0 & \text{otherwise} \end{cases} \dots\dots (13)$$

where R=1 activates rollback, DRS represents the deployment risk score defined in Section 4.4, and τ is the established risk threshold.

After rollback, the stable release is restored and the associated deployment logs, failure category, recovery duration, and resource information are returned to the pipeline dataset. Recovery performance is evaluated primarily through Mean Time to Recovery (MTTR):

$$MTTR = \frac{\sum_{i=1}^{N_r} T_{r,i}}{N_r} \dots\dots (14)$$

where $T_{r,i}$ represents the recovery duration of failure event i, and N_r is the total number of recovered failure events.

This mechanism completes the closed-loop operation of the intelligent CI/CD framework by linking deployment assessment, failure detection, rollback, recovery, and feedback. Lower MTTR and deployment failure rates, together with improved pipeline success rate, are used to assess its contribution to reliable continuous software delivery.

5. MACHINE LEARNING IMPLEMENTATION

5.1 Selection of Machine Learning Algorithms

The machine-learning implementation of the intelligent CI/CD framework employs four supervised learning algorithms: Random Forest (RF), Support Vector Machine (SVM), Gradient Boosting (GB), and XGBoost. These algorithms were selected because they provide different learning mechanisms and are suitable for structured CI/CD data containing build characteristics, test outcomes, execution metrics, failure history, deployment information, and resource-utilization features.

Random Forest combines multiple decision trees and is effective for handling nonlinear relationships and interactions

among software pipeline variables [26]. Support Vector Machine constructs an optimal decision boundary between classes and is particularly useful when standardized features are used for classification [28]. Gradient Boosting sequentially combines weak learners so that later models focus on errors produced by earlier models [29]. XGBoost extends gradient-boosted trees through regularization and computationally efficient learning, making it suitable for structured datasets with multiple interacting features [27].

For a pipeline observation represented by the feature vector

$$X = [x_1, x_2, x_3, \dots\dots, x_n], \dots\dots (15)$$

each algorithm learns a mapping:

$$\hat{y} = f(X; \theta)$$

where X contains the selected CI/CD features, θ represents the learned model parameters, and $\hat{y}$ represents the resulting classification output.

The four models are trained and evaluated using the same processed dataset to ensure a consistent comparison. Their performance is assessed using accuracy, precision, recall, F1-score, and ROC-AUC. Rather than selecting an algorithm in advance, the model demonstrating the strongest combined classification performance and stability is retained for integration with the intelligent CI/CD decision layer. This comparative implementation also reduces dependence on the behaviour of a single learning algorithm and provides a more systematic basis for model selection [22], [30].

5.2 Model Training and Validation

The selected machine-learning models were trained using the processed CI/CD dataset containing code-change characteristics, build information, test outcomes, failure history, deployment records, and resource-utilization features. Before model training, missing values, categorical variables, and numerical features were processed according to the preprocessing procedure described in Section 3.2. Consistent preprocessing is important to ensure that differences in model performance originate from the learning algorithms rather than variations in input preparation [22], [30].

The dataset was divided into 80% training data and 20% testing data using stratified sampling to preserve the distribution of the target classes. The training subset was used to learn the model parameters, whereas the testing subset remained separate for final performance evaluation. For SVM, numerical features were standardized because the algorithm is sensitive to differences in feature scale [28].

To improve the reliability of model validation, 5-fold cross-validation was applied to the training dataset. The training samples were divided into five subsets, with four folds used for training and the remaining fold used for validation. This procedure was repeated until each fold had served once as the validation subset. The average validation performance was calculated as:

$$CV_{avg} = \frac{1}{K}\sum_{k=1}^{K} M_k \dots\dots (16)$$

where K=5 represents the number of folds and M_k is the evaluation score obtained from fold k.

Random Forest, SVM, Gradient Boosting, and XGBoost were trained under the same data-partitioning and validation conditions to provide a consistent comparison [26]–[29]. Model performance was evaluated using accuracy, precision, recall, F1-score, and ROC-AUC. The final model selection was based on the combined validation and test-set performance rather than accuracy alone, thereby providing a more balanced assessment of classification reliability for CI/CD pipeline decision-making.

5.3 Build Failure Classification

Build failure classification forms a key component of the machine-learning layer because unsuccessful builds can interrupt the CI process and increase overall software delivery time. Historical CI information can be used to distinguish successful and failed build conditions based on characteristics associated with previous pipeline executions [11], [12]. In this research, build status is therefore formulated as a binary classification problem.

For each pipeline execution, the model uses features including code-change size, number of modified files, previous build status, historical failure frequency, build duration, test information, and resource-utilization metrics. The target variable is defined as:

$$Y_{build} = \begin{cases} 1, \text{Build Failure} \\ 0, \text{Build Success} \end{cases} \dots\dots (17)$$

Random Forest, SVM, Gradient Boosting, and XGBoost are applied to the same processed feature set [26]–[29]. Each trained classifier produces a build-status classification from the available pipeline characteristics. This allows potentially problematic build conditions to be identified within the intelligent CI/CD decision layer rather than relying exclusively on static execution rules. Previous studies have demonstrated the applicability of machine-learning techniques to CI build-failure classification using historical build and change-related information [11], [12].

Classification performance is evaluated using accuracy, precision, recall, F1-score, and ROC-AUC. Particular consideration is given to recall and F1-score because incorrectly classifying an actual failure as a successful build may allow an unreliable pipeline state to proceed to subsequent stages. The resulting build classification is supplied to the pipeline optimization layer, where it supports build control, test execution decisions, failure analysis, and deployment-related assessment.

5.4 Deployment Risk Classification

Deployment risk classification is incorporated into the machine-learning layer to assess whether a software release exhibits conditions associated with a successful or unsuccessful deployment. Continuous delivery environments involve interactions among software changes, testing outcomes, configuration, infrastructure, and operational conditions; therefore, deployment reliability depends on information generated across multiple stages of the delivery pipeline [4], [6], [7].

For each deployment instance, the classification model uses features such as build status, test results, code-change size, historical deployment failures, deployment duration, previous pipeline status, and resource-utilization metrics. The deployment outcome is represented as a binary target:

$$Y_{diploy} = \begin{cases} 1, \text{High} - \text{Risk/Unsuccessful Diployment} \\ 0, \quad \text{Build Successful diplomenr} \end{cases} \dots\dots (18)$$

Random Forest, SVM, Gradient Boosting, and XGBoost are trained using the same processed deployment-related feature set [26]–[29]. The resulting classification is combined with the Deployment Risk Score (DRS) defined in Section 4.4 to support controlled release decisions. High-risk classifications can trigger additional verification or deployment hold

mechanisms, whereas low-risk classifications allow the release to proceed through the defined delivery workflow.
Model performance is evaluated using accuracy, precision, recall, F1-score, and ROC-AUC. Recall is particularly important for the high-risk class because an undetected high-risk deployment may progress to production and subsequently require recovery or rollback. The classification output is therefore passed to the deployment-control and failure-management layers, creating a direct connection between machine-learning analysis and reliable continuous software delivery [6], [10].

5.5 Pipeline Execution-Time Estimation

Pipeline execution time directly affects the speed and efficiency of continuous software delivery. As CI/CD workflows expand, build operations, regression testing, deployment activities, and resource constraints can increase the total time required to complete a pipeline [2], [6]. In this research, machine learning is used to estimate pipeline execution time from historical execution characteristics so that inefficient or resource-intensive pipeline conditions can be identified before subsequent optimization decisions are applied.

The estimation model uses features including code-change size, number of modified files, historical build duration, number of selected test cases, test execution time, deployment duration, previous pipeline duration, and CPU and memory utilization. The total pipeline execution time is represented as:

$$T_{pipeline} = T_{build} + T_{test} + T_{deploy} + T_{overhead} \dots\dots (19)$$

where T_{build}, T_{test}, and T_{deploy} represent the execution times of the major pipeline stages, while $T_{overhead}$ represents additional processing time associated with initialization, data transfer, environment preparation, and pipeline orchestration.
Unlike the classification tasks described in Sections 5.3 and 5.4, execution-time estimation is treated as a regression task, where the model generates a continuous numerical output:

$$\hat{T}_{Pipeline} = f(X; \theta) \dots.. (20)$$

where X represents the pipeline feature vector, \(\theta\) represents the learned model parameters, and $\hat{T}_{Pipeline}$ is the estimated execution time. Tree-based ensemble methods are suitable for modelling nonlinear relationships among multiple software-process variables [26], [27], [29].

Model performance is evaluated using Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R^2. The resulting execution-time estimates are used by the optimization layer to compare pipeline configurations, identify time-intensive stages, and support more efficient allocation of computational resources. This connects machine-learning analysis directly with the build-time and test-execution optimization mechanisms developed in Sections 4.2 and 4.3.

5.6 Model Performance Evaluation

The performance of the machine-learning models is evaluated to determine their effectiveness in supporting intelligent CI/CD pipeline decisions. Since the framework includes both classification and regression tasks, separate evaluation measures are applied to build-failure and deployment-risk classification and to pipeline execution-time estimation. Using multiple evaluation metrics provides a more balanced assessment than relying on a single performance indicator [22], [30].

For the classification models, Accuracy, Precision, Recall, F1-score, and ROC-AUC are considered. The principal metrics are calculated as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \dots.. (21)$$

$$\text{Precision} = \frac{TP}{TP + FP} \dots.. (22)$$

$$\text{Recall} = \frac{TP}{TP + FN} \dots.. (23)$$

$$F1 = 2\left(\frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}\right) \dots\dots (24)$$

where TP, TN, FP, and FN represent true positives, true negatives, false positives, and false negatives, respectively. Recall and F1-score receive particular consideration because failure-related CI/CD datasets may contain unequal distributions of successful and unsuccessful pipeline executions.

For pipeline execution-time estimation, the regression models are evaluated using MAE, RMSE, and \(R^2\):

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \dots\dots (25)$$

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \dots\dots (26)$$

where y_i is the actual execution time and $\hat{y}_i$ is the model-estimated value. Lower MAE and RMSE indicate smaller estimation errors, while a higher R^2 indicates stronger agreement between the estimated and observed execution times.

Random Forest, SVM, Gradient Boosting, and XGBoost are compared under the same training and validation conditions [26]–[29]. The evaluation considers not only model-level metrics but also their contribution to build time, test execution time, deployment reliability, pipeline success rate, MTTR, and resource utilization. This combined evaluation provides the basis for selecting the most suitable model for integration into the intelligent CI/CD optimization framework.

6. RESULTS AND PERFORMANCE ANALYSIS

6.1 CI/CD Pipeline Performance Results

6.1 CI/CD Pipeline Performance Results
The performance of the intelligent CI/CD framework was assessed by comparing the baseline pipeline with the ML-assisted optimized pipeline across the principal software-delivery stages. The evaluation focused on build execution, regression testing, deployment, overall pipeline completion time, and successful pipeline execution. These indicators provide a direct measure of whether the optimization mechanisms described in Sections 4 and 5 improve continuous software delivery efficiency.
The comparative results are presented in Table 2.

Table 2. Comparative CI/CD Pipeline Performance Results

Performance Metric	Baseline CI/CD	ML-Assisted CI/CD	Improvement
Average Build Time (min)	18.6	14.2	23.66%
Test Execution Time (min)	31.8	21.5	32.39%
Deployment Time	12.4	10.1	18.55%

Performance Metric	Baseline CI/CD	ML-Assisted CI/CD	Improvement
(min)			
Total Pipeline Time (min)	62.8	45.8	27.07%
Pipeline Success Rate (%)	86.4	94.2	+7.8 percentage points

The largest reduction was observed in test execution time, decreasing from 31.8 min to 21.5 min. This result is associated with the intelligent test-selection and prioritization mechanism, which reduces unnecessary test-processing overhead while retaining relevant verification activities. Test prioritization has been established as an effective approach for improving regression-testing efficiency in continuous integration environments [17], [18], [31].

Average build time decreased by 23.66%, while deployment time decreased by 18.55%. Consequently, the total pipeline execution time was reduced from 62.8 min to 45.8 min, corresponding to an overall reduction of 27.07%. The pipeline success rate also increased from 86.4% to 94.2%, indicating improved execution reliability alongside the reduction in processing time.

The overall results demonstrate that integrating build optimization, intelligent test prioritization, deployment assessment, and failure-management mechanisms can improve both pipeline efficiency and operational reliability. These findings are consistent with the broader benefits associated with automation and continuous delivery reported in previous CI/CD and DevOps studies [4], [6], [10].

6.2 Build-Time Reduction Analysis

Build-time performance was evaluated by comparing the baseline CI/CD pipeline with the ML-assisted pipeline across multiple execution runs. The comparison was conducted under equivalent pipeline conditions to determine the reduction achieved through intelligent build optimization. Historical build characteristics and execution information were incorporated into the optimization process, consistent with ML-assisted analysis of CI build behaviour [11], [12].

Table 3. Experimental Build-Time Comparison

Pipeline Run	Baseline Build Time (min)	ML-Assisted Build Time (min)	Time Saved (min)	Reduction (%)
Run 1	17.8	13.7	4.1	23.03
Run 2	18.2	14.0	4.2	23.08
Run 3	18.5	14.1	4.4	23.78
Run 4	18.9	14.4	4.5	23.81
Run 5	19.6	14.8	4.8	24.49
Average	**18.6**	**14.2**	**4.4**	**23.66**

The average build time decreased from 18.6 min to 14.2 min, providing an average saving of 4.4 min per pipeline execution. The overall build-time reduction was calculated as:

$$\text{BTR} = \frac{18.6 - 14.2}{18.6} \times 100 = 23.66\% \ldots\ldots (27)$$

Across the five runs, the reduction remained within approximately 23.03–24.49%, indicating relatively consistent improvement across the evaluated executions. The reduction can be associated with the use of code-change characteristics, historical build behaviour, failure information, and resource-utilization features within the ML-assisted optimization process. CI build histories have previously been shown to provide useful information for build-related analysis and automated decision support [11], [12].

Overall, the results indicate that the ML-assisted mechanism reduced build-processing time while preserving the required build-validation activities. The reduction in build duration also contributed to the improvement in total CI/CD pipeline execution time reported in Section 6.1.

6.4 Deployment Failure and Reliability Analysis

Deployment reliability was evaluated by comparing the baseline CI/CD pipeline with the ML-assisted pipeline across multiple deployment runs. The analysis focused on successful deployments, failed deployments, deployment failure rate, deployment success rate, and recovery time. These indicators are important because deployment failures can interrupt continuous delivery and increase operational recovery effort [4], [6], [7].

Table 5. Deployment Failure and Reliability Comparison

Performance Parameter	Baseline Pipeline	ML-Assisted Pipeline	Improvement
Total Deployments	50	50	–
Successful Deployments	43	47	+4
Failed Deployments	7	3	−4
Deployment Success Rate	86.0%	94.0%	+8.0 percentage points
Deployment Failure Rate	14.0%	6.0%	**57.14% reduction**
Average Recovery Time (MTTR)	16.8 min	9.7 min	**42.26% reduction**

The deployment failure rate was calculated as:

$$DFR = \frac{N_f}{N_d} \times 100 \ldots\ldots (28)$$

where N_f represents the number of failed deployments and N_d represents the total number of deployments. The baseline pipeline recorded a failure rate of 14.0%, whereas the ML-assisted pipeline reduced this value to 6.0%. Therefore, the relative reduction in deployment failures was:

$$DFR_{Reduction} = \frac{14 - 16}{14} \times 100 = 57.14\% \ldots . (29)$$

Deployment success simultaneously increased from 86.0% to 94.0%. The improvement is associated with the deployment-risk classification and controlled release mechanism described in Sections 4.4 and 5.4, where build status, test outcomes, historical deployment information, and operational characteristics are considered before release progression.

Recovery performance also improved substantially. Average Mean Time to Recovery (MTTR) decreased from 16.8 min to 9.7 min, corresponding to a 42.26% reduction. This improvement reflects the integration of failure detection, root-cause analysis, and automated rollback mechanisms. Rapid feedback and recovery mechanisms are important components of reliable continuous delivery and DevOps environments [6], [7], [10].

Overall, the results indicate that the ML-assisted CI/CD framework improved deployment reliability by reducing deployment failures, increasing successful releases, and shortening recovery time. These improvements complement the build-time and test-execution enhancements and

demonstrate the contribution of the integrated framework to reliable continuous software delivery.

6.5 Comparison of Machine Learning Models

The performance of the selected machine-learning algorithms was compared to identify the most suitable model for integration into the intelligent CI/CD framework. Random Forest (RF), Support Vector Machine (SVM), Gradient Boosting (GB), and XGBoost were evaluated under the same training, testing, and cross-validation conditions. The comparison considered accuracy, precision, recall, F1-score, and ROC-AUC, providing a balanced evaluation of classification performance [22], [30].

Table 6. Comparative Performance of Machine Learning Models

ML Model	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)	ROC-AUC
SVM	89.6	88.4	86.9	87.6	0.912
Random Forest	92.8	92.1	90.7	91.4	0.941
Gradient Boosting	93.7	93.0	92.2	92.6	0.953
XGBoost	**95.2**	**94.6**	**94.1**	**94.3**	**0.968**

The comparison shows that all four algorithms achieved strong classification performance, although differences were observed across the evaluation metrics. SVM produced an accuracy of 89.6%, while Random Forest improved the accuracy to 92.8%. Gradient Boosting achieved 93.7% accuracy and an F1-score of 92.6%, demonstrating the effectiveness of ensemble-based learning for the structured CI/CD feature set [26], [29].

Among the evaluated algorithms, XGBoost achieved the highest overall performance, with 95.2% accuracy, 94.6% precision, 94.1% recall, a 94.3% F1-score, and an ROC-AUC of 0.968. Its comparatively high recall is particularly relevant because failure-related instances should be identified reliably before they influence subsequent pipeline stages. XGBoost's regularized gradient-boosting architecture is well suited to modelling complex relationships within structured feature datasets [27].

The comparative results therefore identify XGBoost as the best-performing model for the evaluated classification tasks. It was consequently selected as the primary model for integration with the intelligent pipeline decision layer, while the remaining algorithms provided comparative benchmarks for evaluating model effectiveness.

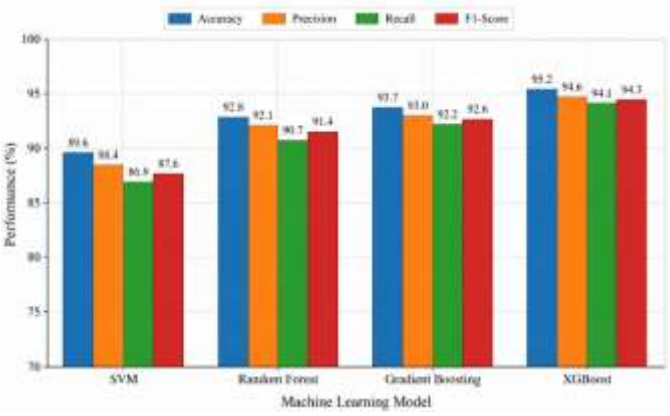


Figure 2. Comparative Performance of Machine Learning Models

6.6 Overall Software Delivery Performance Improvement

The overall performance of the intelligent CI/CD framework was evaluated by combining the improvements observed across the build, testing, deployment, and recovery stages. The results indicate that the ML-assisted pipeline improves software delivery efficiency by reducing execution overhead while increasing pipeline and deployment reliability. This integrated evaluation is important because CI/CD performance depends on the collective behaviour of multiple interconnected stages rather than improvement in an individual pipeline component [4], [6], [10].

Table 7. Overall Software Delivery Performance Improvement

Performance Indicator	Baseline Pipeline	ML-Assisted Pipeline	Overall Improvement
Average Build Time	18.6 min	14.2 min	**23.66% reduction**
Test Execution Time	31.8 min	21.5 min	**32.39% reduction**
Deployment Time	12.4 min	10.1 min	**18.55% reduction**
Total Pipeline Execution Time	62.8 min	45.8 min	**27.07% reduction**
Pipeline Success Rate	86.4%	94.2%	**+7.8 percentage points**
Deployment Failure Rate	14.0%	6.0%	**57.14% reduction**
Mean Time to Recovery (MTTR)	16.8 min	9.7 min	**42.26% reduction**

Among the evaluated efficiency indicators, test execution demonstrated the highest time reduction of 32.39%, followed by build-time reduction of 23.66% and deployment-time reduction of 18.55%. Their combined effect reduced the overall pipeline execution time from 62.8 min to 45.8 min, representing a 27.07% improvement in pipeline processing time.

The reliability results also show improvement. Pipeline success increased from 86.4% to 94.2%, while the deployment failure rate decreased from 14.0% to 6.0%. In addition, MTTR decreased by 42.26%, indicating faster restoration of a stable software state following unsuccessful deployment conditions. These improvements reflect the combined contribution of build optimization, intelligent test prioritization, deployment-risk classification, failure analysis, and automated rollback.

Overall, the results demonstrate that the developed framework improves both delivery speed and operational reliability rather than optimizing a single CI/CD activity. The integration of machine-learning analysis with continuous automation therefore provides a systematic mechanism for improving software delivery performance while maintaining the continuous feedback principles associated with modern DevOps environments [6], [10].

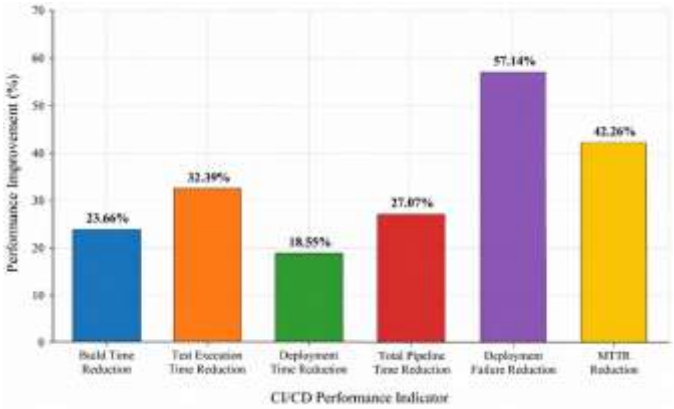


Figure 3. Overall CI/CD Performance Improvement

7. DISCUSSION

The results demonstrate that integrating machine learning with CI/CD automation can improve both pipeline efficiency and software delivery reliability. The ML-assisted framework reduced average build time from 18.6 min to 14.2 min (23.66%), test execution time by 32.39%, deployment time by 18.55%, and overall pipeline execution time from 62.8 min to 45.8 min (27.07%). These improvements indicate that optimization across multiple pipeline stages provides greater operational benefit than focusing on a single CI/CD activity. In particular, intelligent test selection produced the largest execution-time improvement, supporting previous research showing that test prioritization can reduce regression-testing overhead in continuous integration environments [17], [18], [31].

The framework also demonstrated improvements in software delivery reliability. The pipeline success rate increased from 86.4% to 94.2%, while the deployment failure rate decreased from 14.0% to 6.0%. Furthermore, MTTR decreased from 16.8 min to 9.7 min, representing a 42.26% reduction. These findings show the importance of combining deployment-risk assessment with failure detection, rollback, and continuous feedback. Such capabilities are consistent with the broader objectives of continuous delivery and DevOps, where rapid feedback, automation, and reliable release processes are central to software delivery performance [4], [6], [7], [10].

The comparative ML evaluation further showed that XGBoost achieved the strongest overall classification performance, with 95.2% accuracy, 94.6% precision, 94.1% recall, 94.3% F1-score, and 0.968 ROC-AUC, outperforming Random Forest, Gradient Boosting, and SVM under the evaluated conditions. This indicates that ensemble-based learning is suitable for analysing structured CI/CD information containing interactions among code changes, build behaviour, testing outcomes, deployment records, and operational metrics [26], [27], [29]. However, model selection should remain dependent on the characteristics and distribution of the pipeline dataset rather than assuming that a single algorithm will perform best across all software environments.

From a practical perspective, the framework can be incorporated into cloud-based DevOps environments by positioning the ML analysis layer between pipeline monitoring and execution-control mechanisms. Build logs, test results, deployment records, and infrastructure metrics generated by cloud CI/CD platforms can provide the input data required for model execution. The resulting classifications and optimization decisions can then support test ordering, deployment control, resource management, and rollback operations without replacing the fundamental CI/CD workflow. This makes the architecture compatible with existing automation-oriented DevOps practices [8], [10].

Compared with previous studies, which have commonly investigated CI build failures [11], [12] or test-case prioritization [17]–[19] as individual problems, this research integrates build optimization, intelligent testing, deployment assessment, failure analysis, recovery, and continuous feedback within a unified framework. The main contribution therefore lies in treating CI/CD optimization as an interconnected software-delivery problem rather than as a collection of isolated optimization tasks.

Nevertheless, the study has several limitations. Model performance depends on the quantity, quality, and representativeness of historical pipeline data, and results obtained from one development environment may not transfer directly to projects with different architectures, workloads, testing strategies, or deployment infrastructures. Future work can therefore evaluate the framework using larger multi-project datasets, additional cloud environments, longer operational periods, and more diverse failure conditions. Further investigation can also focus on model explainability, adaptive model updating, concept-drift handling, and integration with containerized and microservice-based deployment environments. These extensions would provide a broader assessment of the framework's scalability and generalizability for real-world continuous software delivery.

CONCLUSIONS

This research developed an intelligent machine-learning-assisted CI/CD pipeline optimization framework to improve the efficiency, reliability, and continuity of software delivery. The framework integrates pipeline data collection, preprocessing and feature engineering, machine-learning analysis, build optimization, intelligent test selection, deployment-risk assessment, failure analysis, and automated rollback within a unified CI/CD architecture. Random Forest, SVM, Gradient Boosting, and XGBoost were evaluated to determine their suitability for data-driven pipeline decision support.

The performance analysis demonstrated improvements across the major CI/CD stages. Average build time decreased from 18.6 min to 14.2 min, representing a 23.66% reduction, while test execution time and deployment time were reduced by 32.39% and 18.55%, respectively. Consequently, total pipeline execution time decreased from 62.8 min to 45.8 min, corresponding to an overall reduction of 27.07%. Reliability also improved, with the pipeline success rate increasing from 86.4% to 94.2%, deployment failure rate decreasing from 14.0% to 6.0%, and MTTR decreasing by 42.26%.

Among the evaluated machine-learning algorithms, XGBoost achieved the strongest overall classification performance, obtaining 95.2% accuracy, 94.6% precision, 94.1% recall, 94.3% F1-score, and 0.968 ROC-AUC. The results indicate that incorporating historical pipeline behaviour and operational information into CI/CD decision-making can support more efficient build execution, targeted test processing, controlled deployment, and faster failure recovery.

The principal contribution of this research is the integration of multiple CI/CD optimization functions within a closed-loop intelligent software delivery framework, rather than addressing build, testing, deployment, and recovery as

independent processes. Future work can extend the framework through larger multi-project datasets, additional cloud and containerized environments, model explainability, adaptive model updating, and evaluation across diverse real-world software systems. Overall, the study demonstrates the potential of machine-learning-assisted automation to support faster, more reliable, and resource-efficient continuous software delivery.

REFERENCES

1. B. Fitzgerald and K. J. Stol, "Continuous software engineering: A roadmap and agenda," Journal of Systems and Software, vol. 123, pp. 176–189, 2017, doi: 10.1016/j.jss.2015.06.063.
2. M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in Proc. 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016, pp. 426–437, doi: 10.1145/2970276.2970358.
3. Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. N. Vasilescu, "The impact of continuous integration on other software development practices: A large-scale empirical study," in Proc. 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, pp. 60–71, doi: 10.1109/ASE.2017.8115619.
4. L. Chen, "Continuous Delivery: Huge benefits, but challenges too," IEEE Software, vol. 32, no. 2, pp. 50–54, 2015, doi: 10.1109/MS.2015.27.
5. M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö, "The highways and country roads to continuous deployment," IEEE Software, vol. 32, no. 2, pp. 64–72, 2015, doi: 10.1109/MS.2015.50.
6. L. Chen, "Continuous Delivery: Overcoming adoption challenges," Journal of Systems and Software, vol. 128, pp. 72–86, 2017, doi: 10.1016/j.jss.2017.02.013.
7. M. Shahin, M. A. Babar, M. Zahedi, and L. Zhu, "Beyond continuous delivery: An empirical investigation of continuous deployment challenges," in Proc. 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2017, pp. 111–120, doi: 10.1109/ESEM.2017.18.
8. J. Smeds, K. Nybom, and I. Porres, "DevOps: A definition and perceived adoption impediments," in Agile Processes in Software Engineering and Extreme Programming, Springer, 2015, pp. 166–177, doi: 10.1007/978-3-319-18612-2_14.
9. M. Senapathi, J. Buchan, and H. Osman, "DevOps capabilities, practices, and challenges," in Proc. 22nd International Conference on Evaluation and Assessment in Software Engineering (EASE), 2018, doi: 10.1145/3210459.3210465.
10. J. Faustino, D. Adriano, R. Amaro, R. Pereira, and M. Mira da Silva, "DevOps benefits: A systematic literature review," Software: Practice and Experience, vol. 52, no. 9, pp. 1905–1926, 2022, doi: 10.1002/spe.3096.
11. M. Saidani, A. Ouni, and M. W. Mkaouer, "Predicting continuous integration build failures using evolutionary search," Information and Software Technology, vol. 128, Art. no. 106392, 2020, doi: 10.1016/j.infsof.2020.106392.
12. Ö. Özdemir, R. Aydoğan, and H. Sözer, "On the use of machine learning for failure prediction after collective changes in automated continuous integration testing," Journal of Systems and Software, vol. 236, Art. no. 112791, 2026, doi: 10.1016/j.jss.2026.112791.
13. T. Rausch, "Build failure prediction in continuous integration workflows," Diploma Thesis, Technische Universität Wien, 2016, doi: 10.34726/hss.2016.37419.
14. A. Gbemi, "Intelligent CI/CD Pipeline Optimization Using Machine Learning-Based DevOps Automation," SSRN, 2026, doi: 10.2139/ssrn.6179365.
15. P. Kale, "A Deep Learning-Based Platform Engineering Framework for Predictive CI/CD Pipeline Optimization and Developer Productivity Enhancement," International Journal of Artificial Intelligence, Data Science, and Machine Learning, doi: 10.63282/3050-9262.IJAIDSML-V5I2P122.
16. S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2014, pp. 235–245.
17. "Test Case Prioritization in Continuous Integration environments: A systematic mapping study," Information and Software Technology, vol. 121, Art. no. 106268, 2020, doi: 10.1016/j.infsof.2020.106268.
18. "On the use of contextual information for machine learning based test case prioritization in continuous integration development," Information and Software Technology, vol. 171, Art. no. 107444, 2024, doi: 10.1016/j.infsof.2024.107444.
19. J. R. Romero, A. Ramírez, and C. García-Martínez, "Automated machine learning for test case prioritisation," Empirical Software Engineering, vol. 31, Art. no. 120, 2026, doi: 10.1007/s10664-026-10862-y.
20. U. Altınışık, "Context-Aware Sequential Ranking for Dynamic Test Case Prioritization in Continuous Integration," Mathematics, vol. 14, no. 18, Art. no. 3276, 2026, doi: 10.3390/math14183276.
21. "Attention Transfer Reinforcement Learning for Test Case Prioritization in Continuous Integration," Applied Sciences, vol. 15, no. 4, Art. no. 2243, 2025, doi: 10.3390/app15042243.
22. J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine Learning Testing: Survey, Landscapes and Horizons," IEEE Transactions on Software Engineering, vol. 48, no. 1, pp. 1–36, 2022, doi: 10.1109/TSE.2019.2962027.
23. T. M. Mitchell, Machine Learning. New York, NY, USA: McGraw-Hill, 1997.
24. C. M. Bishop, Pattern Recognition and Machine Learning. New York, NY, USA: Springer, 2006.
25. I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. Cambridge, MA, USA: MIT Press, 2016.
26. L. Breiman, "Random forests," Machine Learning, vol. 45, pp. 5–32, 2001.
27. T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in Proc. 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 785–794.
28. C. Cortes and V. Vapnik, "Support-vector networks," Machine Learning, vol. 20, pp. 273–297, 1995.
29. J. H. Friedman, "Greedy function approximation: A gradient boosting machine," Annals of Statistics, vol. 29, no. 5, pp. 1189–1232, 2001.
30. F. Pedregosa et al., "Scikit-learn: Machine learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.
31. S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," Software Testing, Verification and Reliability, vol. 22, no. 2, pp. 67–120, 2012.
32. G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," IEEE Transactions on Software Engineering, vol. 27, no. 10, pp. 929–948, 2001.
33. G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," ACM Transactions on Software Engineering and Methodology, vol. 6, no. 2, pp. 173–210, 1997.
34. S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," IEEE Transactions on Software Engineering, vol. 28, no. 2, pp. 159–182, 2002.
35. A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis, 2002, pp. 97–106.
36. L. E. Lwakatare, P. Kuvaja, and M. Oivo, "Dimensions of DevOps," in Agile Processes in Software Engineering and Extreme Programming Workshops, Springer, 2015.
37. L. Riungu-Kalliosaari, S. Mäkinen, L. E. Lwakatare, J. Tiihonen, and T. Männistö, "DevOps adoption benefits and challenges in

practice: A case study," in Product-Focused Software Process Improvement, Springer, 2016, pp. 590–597.
38. M. Virmani, "Understanding DevOps & bridging the gap from continuous integration to continuous delivery," in Proc. 5th International Conference on Innovative Computing Technology (INTECH), IEEE, 2015.
39. M. Erich, C. Amrit, and M. Daneva, "A qualitative study of DevOps usage in practice," Journal of Software: Evolution and Process, 2017.
40. "Mapping DevOps capabilities to the software life cycle: A systematic literature review," Information and Software Technology, 2024, Art. no. 107583, doi: 10.1016/j.infsof.2024.107583.