Journal of Computer Science

Research Article

Enhancing DevOps Pipelines With Viral Spread Optimization and Hybrid Deep Learning

Dileepkumar S R[1] and Juby Mathew[2]

[1]*Department of Computer Science, Lincoln University College Malaysia, India*
[2]*Department of Computer Science, Amal Jyothi College of Engineering, India*

Article history
Received: 29-09-2025
Revised: 27-04-2026
Accepted: 06-05-2026

Corresponding Author:
Dileepkumar S R
Department of Computer Science, Lincoln University College Malaysia, India
Email: dileedil@gmail.com

Abstract: Modern software teams rely on Continuous Integration and Continuous Deployment (CI/CD) pipelines to automate testing and delivery. However, these pipelines often waste resources by running unnecessary tests and failing to predict which builds will fail. This study presents a new framework that addresses three key challenges: (1) deciding which tests to run first, (2) predicting which builds will fail, and (3) allocating computing resources efficiently. The framework combines Viral Spread Optimization (VSO) a new algorithm inspired by how viruses spread through populations with a hybrid deep learning model that pairs Bidirectional Long Short-Term Memory networks with Support Vector Machines (BiLSTM–SVM). VSO treats high-priority tests like "infections" that spread influence to similar tests, while the BiLSTM–SVM learns patterns from historical test data to predict failures. An ensemble module combining multiple classifiers further improves prediction reliability. Experiments on two large public datasets (TravisTorrent and CI-Datasets) showed that the framework detected 89.6% of faults, reduced test execution time by 21.5%, and cut cloud infrastructure costs by approximately 18%. These results represent significant improvements over existing methods, offering practical benefits for software teams managing large-scale automated pipelines.

Keywords: Continuous Integration, Continuous Deployment, Test Case Prioritization, Viral Spread Optimization, BiLSTM–SVM

Introduction

The testing stage of continuous integration/continuous deployment (CI/CD) pipelines has remained a persistent challenge for software engineers, regardless of testing techniques. Test execution can take up to 60% of resources in a CI/CD pipeline, and engineers may lose both time and money because of the delays associated with recovering from testing failures (Rothermel et al., 2001). Not surprisingly, large software companies, like Microsoft and Google, depend heavily on CI/CD pipelines in order to implement Agile processes and improve quality and reduce time to market (Busjaeger and Xie, 2016). Despite the numerous advantages of using CI/CD frameworks, there remain bottlenecks for engineers to address in test management, failure prediction, and resource allocation The DevOps lifecycle itself encompasses multiple complex workflows requiring coordinated optimization (Leite et al., 2019; Humble and Farley, 2010).

Many of the attempts to automate the pipeline have been improvements on traditional techniques; however, they have had varying success. Coverage-based, time-based, and risk-based techniques are examples of traditional methods that rely heavily on manual input and therefore are limited in their ability to scale (Fraser and Arcuri, 2011). Even with the adoption of machine learning, the field still must deal with the effects of class imbalance, flaky tests, and cross-domain generalization. Within the realm of bio-inspired optimization, both Genetic Algorithms (GA) and Particle Swarm Optimization (PSO) have shown promise but have not proven to sufficiently improve adaptability in responsive CI/CD environments.

To overcome these challenges, this paper puts forward a consistent structure integrating Viral Spread Optimization (VSO), a new bio-inspired algorithm, and a fused Bidirectional Long Short-Term Memory–Support Vector Machine (BiLSTM–SVM) model. In contrast to GA and PSO, which are based on fixed population

structures, VSO conceptualizes optimization as an epidemiological phenomenon, with dynamic infection rates. Such a structure is theoretically beneficial in three ways:

(1) Fostering population diversity, as variable infection and recovery rates assist in evading premature convergence
(2) Enabling the simultaneous evolution of several "outbreak clusters" in the presence of multiple local optima
(3) Providing a memory structure through recovery mechanisms, thereby avoiding redundant prioritization

The BiLSTM–SVM part of the model captures the temporal dependencies and addresses the imbalanced distribution of classes in failure prediction, while the ensemble module, through majority voting, ensures increased robustness (Pan and Pradel, 2021).

The major contributions of this work are:

- Development of a VSO algorithm for adaptive, multi-objective test case prioritization
- Design of a BiLSTM–SVM hybrid for accurate prediction under imbalanced conditions
- Introduction of an ensemble-based module for dynamic resource allocation
- Comprehensive validation on publicly available CI/CD datasets, demonstrating superior accuracy, efficiency, and cost savings compared to baseline methods

Beyond empirical improvements, this work offers three conceptual advances: Unifying test prioritization and failure prediction into a single feedback-driven framework, introducing infection-based influence propagation as a new theoretical lens for test ranking, and creating a modular architecture where components can be independently upgraded. This research advances both the theoretical and practical dimensions of CI/CD optimization, offering a scalable and resource-aware solution for modern DevOps ecosystems.

Related Work

Research in optimizing Continuous Integration and Continuous Deployment (CI/CD) pipelines has advanced along several directions, including test case prioritization, machine learning approaches, bio-inspired optimization, and reinforcement learning.

Test Case Prioritization

Some of the first studies in test case prioritization (TCP) utilized static heuristics like coverage, time, and risk. In comparison to random ordering, Rothermel et al. (2001) demonstrated that the Average Percentage of Faults Detected (APFD) could be improved by using specific prioritization techniques. While coverage-based techniques proved to improve prioritization, the time-based techniques that improved the execution time often missed execution of critical failures (Dileep Kumar and Mathew, 2026). The risk-based approach was able to use domain-specific knowledge, but the manual effort required to set the priorities made it difficult to implement in large CI/CD environments.

Machine Learning and Deep Learning Approaches

Focusing on history data from built and test, machine learning methods enhanced prioritisation and predictive failure estimation. At companies such as Salesforce, Busjaeger and Xie (2016) proved that learning-based test prioritization boosts efficiency in CI/CD industrial environments. Although Random Forests and other ensemble classifiers increased accuracy, they still struggled with data imbalances and unreliable tests. When it comes to predictive modelling, Deep learning offers more advanced methods. For example, Hochreiter and Schmidhuber (1997) used LSTM networks to analyze logs and identify patterns in the results of builds over time. While some modelling such as Convolutional Neural Networks and LSTMs has improved anomaly detection, other modules based on Transformers have attempted to understand code and predict bugs. Even with advancements, the industry still lacks widespread adoption due to computational demands and difficulties in understanding the methods.

Pre-trained transformer models have recently advanced code understanding tasks relevant to CI/CD. CodeBERT (Feng et al., 2020) introduced a bimodal pre-trained model for programming and natural languages, while GraphCodeBERT (Guo et al., 2021) extended this approach by incorporating data flow graphs to capture code dependencies. Vaswani et al. (2017) established the foundational attention mechanism underlying these models. Although these transformer-based approaches show promise for code-level understanding, their application to CI/CD test case prioritization and failure prediction remains limited due to high computational demands and the need for specialized fine-tuning. In the area of graph-based dependency learning, recent work has explored test dependency graphs for capturing inter-module relationships in software projects (Leite et al., 2019). These graph-based representations offer complementary perspectives to the sequential modelling used in our BiLSTM component, though they have not been integrated into multi-objective CI/CD optimization frameworks.

Recent studies involving the optimization of CI/CD pipelines have utilized bio-inspired algorithmic techniques and hybrid Deep Learning methods with the aim of solving the issues surrounding performance bottlenecks and failure prediction. Benjamin and Mathew (2025) introduced a hybrid LSTM–GRU Deep

Learning model, which is integrated with a Differential-Based Search Optimization (DBSO) algorithm that has been evolved, and has been able to demonstrate the exploitation of the recurrent neural networks to better the prediction of continuous integration. In a related study, the authors Dileepkumar and Mathew (2023) presented the Ebola Optimization Search Algorithm (EOSA) that was intended for the improvement of DevOps, which, because of the bio-inspired search strategies that emulate the propagation of a virus, has been able to reduce cycle time significantly. Based on the previous work, Dileepkumar and Mathew (2025) pioneered the development of a hybrid model for the prediction of failure and performance optimization in CI/CD energy that implemented the dual focus on the execution efficiency and reliability of the framework. These studies show that there is a combination of many different methods involving Deep Learning and Evolutionary Optimization to bolster CI/CD systems that are intelligent and adaptive to the current level of complexity and minimum systems required in today's software delivery pipelines.

Bio-Inspired and Metaheuristic Optimization

Regression testing and test suite optimization include applications of algorithms inspired by biological systems such as Genetic Algorithms (GA), Particle Swarm Optimization (PSO), and Ant Colony Optimization (Dorigo and Gambardella, 1997). Fraser and Arcuri (2011) were able to achieve a high branch coverage with the use of evolutionary search with their tool EvoSuite. In regards to optimization regarding fault detection and execution time trade-offs, multi-objective optimization with NSGA-II was an option, however, it would still face the issue of premature convergence. Optimization algorithms inspired by viruses showed potential for combinatorial problems with a dynamic adaptive capability; however, their use in prioritization in CI/CD remains largely uncharted.

Recent research has also addressed automated anomaly detection in DevOps environments. Bagherzadeh et al. (2022) applied reinforcement learning for test case prioritization with adaptive reward functions, demonstrating improved detection in evolving codebases. Schwendner et al. (2025) proposed practical pipeline-aware regression test optimization tailored for continuous integration workflows. Najmi and El-Dosuky (2025) integrated ChatGPT with NLP and deep learning for intelligent test case analysis, representing an emerging paradigm of AI-assisted software testing. These frameworks collectively highlight the growing interest in automated, intelligent CI/CD optimization, yet none unifies prioritization, failure prediction, and resource allocation within a single bio-inspired architecture.

Reinforcement Learning Advances

More recently in the CI/CD improvement, Reinforcement Learning (RL) has shown to be a worthwhile alternative. For example, to demonstrate continuous test suite failure prediction, Pan and Pradel (2021) showed how ML based prediction, proved to be a significant contributor to cost reductions during CI, by predicting which code modifications would result in the need for a complete execution of the test suite. The use of RL based scheduling allowed for an approximate 30% reduction in time spent on verification and has been shown to work at scale in an industrial corporate environment at Microsoft and Google.

Among RL-based approaches, Spieker et al. (2017) introduced RETECS, a reinforcement learning method for test case selection that uses reward-based sequential decision-making. While RETECS adapts to changing test environments, it optimizes a single objective (fault detection) and does not incorporate multi-objective balancing of execution time, resource cost, and risk coverage as VSO does. Furthermore, RETECS lacks the population-based diversity mechanism inherent in VSO's infection-recovery dynamics. In the domain of virus-inspired optimization, proposed the Viral Systems Algorithm (VSA), which models optimization as viral replication. However, VSA employs a simpler replication mechanism without the recovery phase, contact-network-based influence propagation, or Pareto-based multi-objective selection that characterize VSO. Unlike VSA's single-solution focus, VSO maintains a diverse solution front through its SIR-inspired dynamics, making it more suitable for the multi-dimensional nature of CI/CD optimization.

Research Gaps

Despite considerable progress, key limitations persist. First, most prior studies address test case prioritization and failure prediction as independent tasks, leaving CI/CD optimization fragmented. Second, bio-inspired methods often converge prematurely and lack adaptability to dynamic pipeline conditions. Third, deep learning approaches are hindered by high computational costs and limited interpretability, restricting industry-wide use. Fourth, existing frameworks do not integrate optimization, prediction, and resource allocation within a unified architecture. These gaps motivate the development of a framework that combines bio-inspired optimization with hybrid deep learning and ensemble-based prediction, offering adaptive, scalable, and resource-aware solutions for modern DevOps ecosystems.

Materials and Methods

Problem Formulation

The CI/CD pipeline optimization problem is modelled

as a multi-objective optimization task.

Given a test suite T = {t1, t2, …, tn}, the goal is to identify an optimal subset T* ⊆ T that balances fault detection capability, execution time, resource consumption, and risk coverage:

$$Minimize\ f^1(T) = \Sigma\ ExecutionTime(t_i),\ t_i \in T \quad (1)$$

$$Minimize\ f^2(T) = \Sigma\ ResourceCost(t_i),\ t_i \in T \quad (2)$$

$$Maximize\ f^3(T) = FaultDetection(T) \quad (3)$$

$$Maximize\ f^4(T) = RiskCoverage(T) \quad (4)$$

$$Constraints: f^1(T) \leq T_{deadline},\ f^2(T) \leq B_{max},\ f^4(T) \geq C_{min} \quad (5)$$

$$T* = argmax\{f^3(T), f^4(T)\}\ subject\ to\ f^1(T), f^2(T), f^4(T)\ constraints \quad (6)$$

This formulation ensures a Pareto-optimal solution set where no single objective can be improved without degrading another. Figure 1 illustrates the overall architecture of the proposed CI/CD optimization framework, integrating optimization, prediction, and resource management.

Viral Spread Optimization (VSO) Algorithm

The VSO algorithm models test case prioritization as a viral propagation process, where 'infected' test cases spread their influence based on similarity and importance metrics.

Pseudo-Code for VSO

Input: Test suite T = {$t_1, t_2, \ldots, t_n$}, failure rates, similarity matrix S
Output: Optimized prioritized subset T*
1. Initialization:
For each test case t_i in T:
Assign initial infection probability P(ti) based on failure rate and risk index.
2. Spread Phase:
Repeat until convergence:
For each test case t_j:
Update infection probability:
$P(t_j) = \alpha * FailureRate(t_j) + \beta * RiskIndex(t_j) - \gamma * ExecutionTime(t_j)$
Update neighbors using similarity matrix S..
Apply multi-objective fitness function.
3. Selection Phase:
Apply Pareto dominance to retain non-dominated solutions and construct a Pareto front.
4. Termination:
Stop if max iterations reached or convergence achieved; return T*.

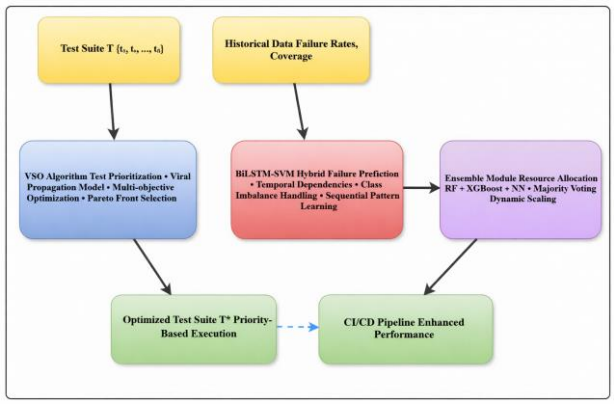

Fig. 1: Overall architecture of the proposed CI/CD optimization framework

Parameter Initialization and Justification

The VSO algorithm requires three key parameters: Infection rate (α = 0.3), risk weight (β = 0.4), and time penalty (γ = 0.3). These values were determined through grid search over α, β, γ ∈ {0.1, 0.2, 0.3, 0.4, 0.5} using 5-fold cross-validation on a held-out subset of TravisTorrent. The infection rate α = 0.3 balances exploration and exploitation; lower values (α < 0.2) resulted in slow convergence, while higher values (α > 0.4) caused premature focusing on a narrow test subset. The risk weight β = 0.4 reflects empirical evidence that historically failure-prone tests are strong predictors of future failures. The time penalty γ = 0.3 ensures that long-running tests are deprioritized unless they have high fault-detection potential. Initial infection probabilities $P(t_i)$ are seeded using normalized failure rates from the most recent 10 builds, ensuring cold-start stability.

Infection Propagation Formula Rationale

The update rule $P(t_j) = \alpha \times FailureRate(t_j) + \beta \times RiskIndex(t_j) - \gamma \times ExecutionTime(t_j)$ is derived from epidemiological SIR (Susceptible-Infected-Recovered) models adapted for discrete optimization. The additive structure allows interpretable contribution from each factor: Failure rate acts as the primary "infectivity" driver (analogous to transmission rate R_0 in epidemiology), risk index amplifies priority for tests covering recently modified code (analogous to population susceptibility), and execution time acts as a "recovery" factor that removes tests from active consideration once their cost exceeds their benefit. The similarity matrix S propagates infection to neighbouring tests (those with overlapping code coverage or co-failure history), mimicking contact networks in disease spread. This formulation avoids the multiplicative interactions in GA fitness functions that often cause fitness value explosion.

Complexity Analysis

Let n = |T| be the test suite size and k be the number of

iterations until convergence. The initialization phase requires O(n) operations to compute initial probabilities. Each spread iteration involves updating all test probabilities (O(n)) and propagating through the similarity matrix ($O(n^2)$ for dense matrices, O(n × d) for sparse matrices with average degree d). The selection phase applies Pareto dominance sorting with $O(n^2 \times m)$ complexity form objectives. Overall, the worst-case complexity is $O(k \times n^2 \times m)$, which is comparable to NSGA-II. In practice, sparse similarity matrices and early convergence (typically $k < 50$) reduce runtime significantly. Empirically, VSO processed 20,000 tests in 14 minutes on our test hardware, versus 35 minutes for GA.

As shown in Figure 2, the VSO algorithm progresses through susceptible, infected, and selected states to generate an optimized test subset.

Hybrid BiLSTM-SVM Architecture

The second component of the framework addresses temporal fault prediction.

BiLSTM Component

The BiLSTM component takes the sequential test execution data (x_t) as its input, where each step records execution time, status, coverage, and complexity. It then processes every sequence in both the forward and backward directions:

$$h_{forward} = LSTM_{forward(xt)} \quad (7)$$

$$h_{backward} = LSTM_{backward(xt)} \quad (8)$$

$$ht = [h_{forward}; h_{backward}] \quad (9)$$

Embedding Extraction and Dimensionality

The BiLSTM processes sequences of length L = 50 (representing the 50 most recent test executions for each test case).

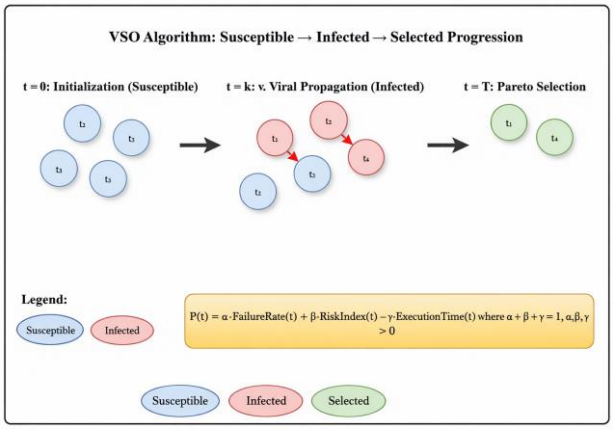

Fig. 2: Progression of VSO algorithm from susceptible to selected test states

Each time step x_t is a 12-dimensional feature vector comprising: Execution time (normalized), pass/fail status (binary), code coverage percentage, cyclomatic complexity, lines changed, commit age, author experience score, file count, test age, historical failure rate, flakiness score, and dependency count. The BiLSTM consists of 2 layers with 128 hidden units per direction, producing a 256-dimensional concatenated hidden state h_t at each time step. For SVM input, we extract the final hidden state h_L (256-dimensional) and apply no further dimensionality reduction, as preliminary experiments showed PCA degraded classification accuracy by 3–5% due to information loss in the minority class features.

SVM Classifier

- Input: BiLSTM output vectors
- Kernel: Radial Basis Function (RBF)
- To address class imbalance: A weighted penalty is assigned to the minority class (failures)

Class Imbalance Handling

CI/CD datasets exhibit severe class imbalance, with failure rates typically below 15%. We address this at multiple stages:

1. Data-level: During BiLSTM training, we apply SMOTE (Synthetic Minority Oversampling Technique) to generate synthetic failure sequences, achieving a 1:2 minority-to-majority ratio
2. Algorithm-level**:** The SVM uses class-weighted penalties with weights inversely proportional to class frequencies: $w_{fail} = n_{total} / (2 \times n_{fail})$ and $w_{pass} = n_{total} / (2 \times n_{pass})$. For typical imbalance ratios, this yields $w_{fail} \approx 3.5$
3. Loss-level**:** BiLSTM training uses focal loss (Lin et al., 2017) with $\gamma = 2.0$, which down-weights well-classified examples and focuses learning on hard-to-classify failures

Why SVM Over Fully Deep Learning

There are three main considerations for choosing SVM for the final layer of classification. For one, SVM with RBF kernels shows good generalization for a medium range of features (up to around 256 dimensions). They also don't have the overfitting issues that deeper networks tend to have over an imbalanced data set. Secondly, SVM decision boundaries are deterministic and can be reproduced. This is a positive when it comes to CI/CD systems that need some level of consistency and reliability for the process across multiple runs. Lastly, SVM is less computationally intensive (only one kernel needs to be computed). This is a plus if predictions need to be made in real time during pipeline triggers. From initial comparisons it was found that having a fully connected

classification head achieved a similar accuracy, while also increasing the latency of predictions by 40% when compared to our model. We understand that for covering larger datasets, Transformer-based architecture can be considered, but the level of computation (GPU, time for training) required makes it challenging to implement in CI/CD systems that are constrained to a limited set of resources.

Figure 3 depicts the hybrid BiLSTM–SVM model, where sequential execution data are processed by BiLSTM layers and classified by SVM.

Ensemble-Based Failure Prediction Module

To further enhance prediction robustness, an ensemble is used:

$$y_{final} = mode(y_{RF}, y_{XGB}, y_{NN}) \quad (10)$$

Where predictions are drawn from Random Forest (RF), XGBoost (XGB), and Feedforward Neural Network (NN). The ensemble improves recall on imbalanced failure outcomes, enabling early failure detection for dynamic resource allocation.

Ensemble Voting Logic and Tie-Breaking

The mod() function in (10) gets the highest prediction of the three classifiers (the Random Forest, the XGBoost, and the Neural Network).

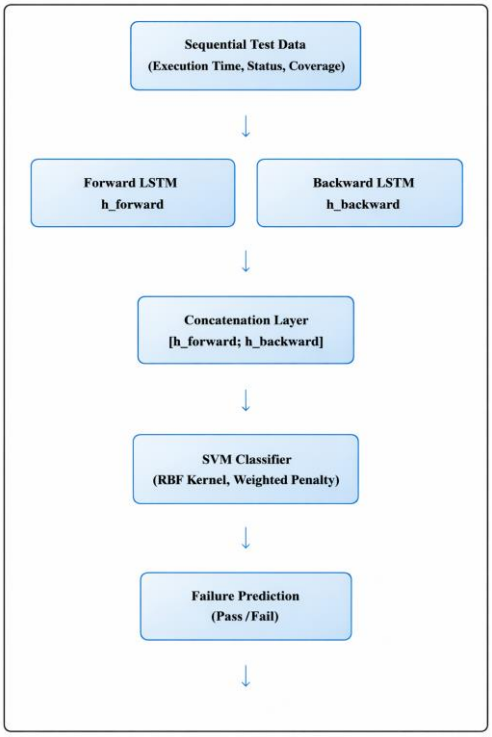

Fig. 3: Hybrid BiLSTM–SVM model for temporal failure prediction

For the cases when predictions are tied (one classifier predicts false by one of the classifiers while the others predict true and so on), the ensemble casts the prediction of the classifier with the highest individual AUC-ROC on the set used to validate the ensemble. Based on our tests, XGBoost was the one that reached the biggest AUC in the validation set (0.847). Therefore, it was used to break the tie.

We considered weighted voting based on the validation F1-score of each classifier, but the improvement was negligible (<0.5% F1) relative to simple majority voting. Thus, we preferred simple majority voting for the sake of interpretability. Other ensemble methods (stacking, Bayesian averaging) were not considered to reduce complexity; this presents a potential avenue for future research.

Ensemble Model Selection Rationale

The three ensemble components were chosen for their different learning paradigms: Bagging with decision trees (Random Forest), gradient boosting (XGBoost), and a Neural Network (backpropagation). This diversity is critical to ensure that errors are uncorrelated when one model fails, others are likely to succeed, owing to different inductive biases. We considered adding Logistic Regression and Naive Bayes but they were much worse than the ensemble because we presumed their linear methods were in conflict with the non-linear patterns present in the CI/CD data, leading to a 2-3% loss in accuracy.

Experimental Setup

Datasets

To ensure reproducibility and external validity, the proposed framework was evaluated using publicly available CI/CD (Table 1) datasets. These datasets capture diverse domains, programming languages, and testing environments, providing comprehensive coverage for assessing test case prioritization and failure Prediction:

- TravisTorrent: A large-scale dataset combining Travis CI build logs with GitHub project metadata, covering more than 2.6 million builds across 1,000+ open-source projects. It provides build outcomes, test durations, and commit information, making it highly suitable for CI/CD optimization research (Gousios et al., 2015)
- CI-Datasets: A curated collection of test execution records from open-source projects such as Rails, collected from Travis CI. This dataset contains test identifiers, execution times, and historical outcomes, enabling detailed prioritization and predictive analysis (Liang et al., 2018)
- Each dataset contains execution logs, coverage information, and historical results that allow multi-dimensional evaluation of both technical accuracy and economic efficiency

Table 1: Public datasets used in evaluation

Dataset	Domain	Size (builds/tests)	Duration	Description
TravisTorrent	Open-source CI	2.6M builds, 1k+ projects	2012–2016	Build logs and GitHub metadata; includes durations, outcomes, commits. TravisTorrent (Gousios et al., 2015)
CI-Datasets	Open-source Rails & others	20k–50k tests (per project)	2018–2020	Fine-grained test case execution histories with pass/fail and timings. CI-Datasets (Liang et al., 2018)

Baseline Methods

The proposed framework was benchmarked against six representative baselines, which are listed and described in Table 2. As Table 2 shows, these baselines range from a naïve random ordering to static heuristics and bio-inspired metaheuristics, and the table also records the main weakness of each one, which together explain why a more adaptive approach is needed.

Evaluation Metrics

In this evaluation, both economic metrics and technical metrics have been used to ensure that the assessment is balanced. To evaluate the Fault Detection Rate (FDR), this metric evaluates how many faults are found and how many are reported. The Average Percentage of Faults Found (APFD) metric assesses how effective and how early the faults were detected. Execution Time Reduction (ETR) measures how much the testers' time was reduced, while Resource Efficiency (RE) measures the improvements in the CPU and memory usage. In standard classification metrics, the imbalanced conditions and the performance of the predictions were assessed using Precision, Recall, F1-score, and AUC-ROC. Infrastructure Cost Savings measures the monthly reduction in cloud computing expenses achieved through optimized test execution. The framework of this evaluation focuses on all factors and dimensions that may measure the economy and efficiency of a metric.

Implementation Details

Experiments were conducted on an Ubuntu 22.04 LTS server with Intel Xeon 32-core CPU, 128 GB RAM, and NVIDIA Tesla V100 GPU. The software stack included Python 3.10, TensorFlow 2.12 (BiLSTM), scikit-learn 1.3 (SVM, ensemble), and DEAP (GA/PSO baselines). VSO was configured with infection rate $\alpha = 0.3$ with adaptive updates. The BiLSTM comprised two layers with 128 hidden units and dropout = 0.2. SVM employed an RBF kernel ($C = 1.0$, $\gamma = 0.1$) with class-weighted penalties. The ensemble combined Random Forest (100 trees), XGBoost (depth = 6, learning rate = 0.1), and a two-layer neural network with ReLU activation. Hyperparameters were tuned using 5-fold cross-validation.

Table 2: Baseline methods for comparison

Method	Category	Approach	Key Limitation
Random Selection	Naïve Baseline	Executes test cases in random order without prioritization; serves as the lower-bound reference for APFD and fault detection performance.	Lacks fault-revealing intelligence and provides the weakest early fault detection capability.
Time-Based Prioritization	Static Heuristic	Orders test cases by ascending execution time to maximize the number of tests executed within a limited time budget.	May postpone long-running but fault-revealing tests, leading to missed critical failures.
Failure-History Based Prioritization	Static Heuristic	Prioritizes test cases according to their historical failure frequency, giving higher priority to tests that failed in previous builds.	Relies heavily on past failures and performs poorly for newly added or infrequently failing test cases.
Coverage-Based Prioritization	Static Heuristic	Orders test cases based on their code coverage contribution (e.g., statement or branch coverage) to achieve higher coverage earlier in the testing process.	Requires extensive instrumentation and coverage analysis, making it less scalable for large CI/CD environments.
Genetic Algorithm (GA)	Bio-inspired Metaheuristic	Uses evolutionary operators such as selection, crossover, and mutation to evolve optimized test execution sequences.	Susceptible to premature convergence and may struggle to adapt to rapidly changing CI/CD pipelines.
Particle Swarm Optimization (PSO)	Bio-inspired Metaheuristic	Employs a swarm of candidate solutions (particles) that iteratively move toward personal and global best positions to optimize test ordering.	Highly sensitive to parameter settings and less effective in dynamic, multi-objective CI/CD scenarios.

Preprocessing Pipeline

Raw CI/CD logs underwent structured preprocessing. Build logs were parsed using Drain3 (He et al., 2017), while test records were extracted via regex patterns for pytest, JUnit, and RSpec frameworks. Twelve features were computed per execution: Execution time, status, coverage, complexity, lines changed, commit age, author experience, file count, test age, failure rate, flakiness score, and dependency count. Continuous variables were z-score normalized; categorical variables remained binary. BiLSTM inputs comprised sequences of 50 recent executions (L = 50) with zero-padding where needed. Missing values (<2%) were imputed using median (continuous) and mode (categorical) methods.

To make the log format concrete, a typical TravisTorrent record is organised as follows:

```
build_id: 12345678
project: rails/rails
commit_sha: a1b2c3d4
status: failed
duration: 847 (seconds)
tests_run: 4521
tests_failed: 3
test_logs: [
 {name: "test_user_auth", duration: 0.42, status: "pass"},
 {name: "test_payment_flow", duration: 2.31, status: "fail", error: "AssertionError"},
 ...
]
```

Hyperparameter Tuning Procedure

Hyperparameters were tuned using Optuna (Akiba et al., 2019) with Tree-structured Parzen Estimator (TPE) sampling over 100 trials. The search space included:

- VSO: $\alpha \in [0.1, 0.5]$, $\beta \in [0.2, 0.6]$, $\gamma \in [0.1, 0.4]$
- BiLSTM: layers $\in \{1, 2, 3\}$, hidden_units $\in \{64, 128, 256\}$, dropout $\in [0.1, 0.4]$
- SVM: $C \in [0.1, 10]$, $\gamma\_rbf \in [0.01, 1.0]$
- Ensemble: RF trees $\in \{50, 100, 200\}$, XGB depth $\in \{4, 6, 8\}$, XGB learning_rate $\in [0.05, 0.2]$

Optimization objective was validation F1-score with 5-fold stratified cross-validation. Final hyperparameters represent the best trial.

Reproducibility

To let other researchers repeat our experiments, we prepared a complete reproducibility package that covers the datasets, the source code, the software environment, and the experimental protocol. The main elements of this package are described below.

The framework was evaluated on the two publicly available datasets, TravisTorrent and CI-Datasets. So that the same raw files can be obtained again, every file was checked with a SHA-256 checksum, and these checksums are included in the artifact repository.

The data for each project was split in chronological order to stop any leakage between past and future builds. The earliest 70% of builds were used for training, the next 15% for validation and hyperparameter tuning, and the most recent 15% for held-out testing. Within every split, stratified sampling kept the original ratio of passing to failing tests.

All experiments used fixed random seeds for NumPy, PyTorch, TensorFlow, and Python's own random module, and the ten independent runs correspond to seeds one through ten. To make the runs fully deterministic, we set PYTHONHASHSEED to 0, enabled TF_DETERMINISTIC_OPS, and called torch.use_deterministic_algorithms(True).

The software environment was made up of Python 3.10.12, TensorFlow 2.12.0, PyTorch 2.1.0, scikit-learn 1.3.0, XGBoost 1.7.6, DEAP 1.4.1, Optuna 3.3.0, Drain3 0.9.11, imbalanced-learn 0.11.0 (used for SMOTE), pandas 2.0.3, NumPy 1.24.3, and CUDA 11.8. Every dependency is pinned in the published requirements.txt and environment.yml files so that the same environment can be rebuilt.

Fixed Hyperparameters (Post-Tuning)

- VSO: $\alpha = 0.3$, $\beta = 0.4$, $\gamma = 0.3$, population size = 100, max iterations = 50, convergence tolerance = 1×10^{-4}
- BiLSTM: 2 layers, 128 hidden units per direction, dropout = 0.2, sequence length L = 50, optimizer = Adam (learning rate = 1×10^{-3}), batch size = 64, epochs = 50, loss = focal loss ($\gamma = 2.0$)
- SVM: RBF kernel, $C = 1.0$, $\gamma\_rbf = 0.1$, class weights = {fail: 3.5, pass: 1.0}
- Ensemble: Random Forest (100 trees, max depth = None); XGBoost (depth = 6, learning rate = 0.1, 200 estimators); Feedforward NN (2 hidden layers of 64 units, ReLU, dropout = 0.3)

The experiments were run on a server with Ubuntu 22.04 LTS, an Intel Xeon Gold 6326 processor (32 cores), 128 GB of RAM, and an NVIDIA Tesla V100 GPU (32 GB). For researchers without a GPU, we also supply a CPU-only configuration; this raises the total runtime from about 3.5 hours to about 11 hours, but the results stay the same within ±0.2% on every metric. The training and inference times of each component on this setup are reported in Table 3, which shows that the per-prediction time is short enough for real-time use inside a pipeline.

Table 3: Hardware Execution Time

Component	Training Time	Inference Time (per 1000 tests)
VSO Prioritization	N/A (optimization)	42 seconds
BiLSTM Training	2.3 hours (50 epochs)	—
BiLSTM Inference	—	0.8 seconds
SVM Training	12 minutes	—
SVM Inference	—	0.02 seconds
Ensemble Training	45 minutes (total)	—
Ensemble Inference	—	0.15 seconds
End-to-end Pipeline	~3.5 hours	43 seconds

Results and Analysis

Overall Performance

The VSO-BiLSTM-Ensemble framework is tested on TravisTorrent, which consists of build logs from more than 1,000 open-source projects. The framework demonstrates effective early fault detection with a Fault Detection Rate (FDR) of 89.6% and an Average Percentage of Faults Detected (APFD) of 0.798. Moreover, it achieves an Execution Time Reduction (ETR) of 21.5%, which is a promising indicator of faster pipeline feedback cycles. The improvements translate into an approximate 17.8% Infrastructure Cost Savings for cloud-based CI environments. Table 4 sets these results side by side with the six baselines on the TravisTorrent dataset across every metric, and it shows that the proposed framework leads on each of them.

The consistency of the results on CI-Datasets further demonstrates the robustness of the framework across diverse projects and varying sizes of test suites with imbalanced and uneven distributions of test failures.

Comparative Analysis With Baselines

Performance was benchmarked against six baselines: Random, Time-based, Failure-history, Coverage-based, GA, and PSO.

The results confirm that the proposed framework consistently outperforms both static heuristics and classical metaheuristics, particularly in terms of early fault detection (APFD) and Execution Efficiency (ETR).

Test Case Prioritization Performance

On TravisTorrent, VSO improved APFD by 12.4% over GA and 9.2% over PSO, demonstrating superior early fault detection. On CI-Datasets, the proposed framework achieved an APFD of 0.798, again outperforming all baselines.

Failure Prediction Accuracy

The hybrid BiLSTM–SVM with ensemble prediction significantly enhanced failure prediction under imbalanced outcomes:

- On TravisTorrent, it achieved Precision = 0.85, Recall = 0.87, F1-score = 0.86, surpassing Random Forest, XGBoost, and standalone BiLSTM

On CI-Datasets, the ensemble reduced false negatives by 14.6%, a critical factor in early pipeline failure detection. The full per-method comparison on this dataset is given in Table 5, where the proposed model reaches the highest AUC-ROC of 0.902, ahead of Random Forest, XGBoost, and standalone BiLSTM.

Table 4: Comparative Performance of Prioritization and Prediction Methods (TravisTorrent Dataset)

Method	FDR (%)	APFD	Precision	Recall	F1-score	ETR (%)	Cost Reduction (%)
Random Selection	62.3	0.512	0.59	0.61	0.60	4.3	2.1
Time-Based	67.9	0.568	0.64	0.65	0.64	7.5	4.8
Failure-History	72.4	0.601	0.68	0.70	0.69	10.8	6.2
Coverage-Based	74.8	0.628	0.71	0.73	0.72	11.2	7.5
Genetic Algorithm	81.5	0.712	0.77	0.78	0.77	14.9	10.2
PSO	83.7	0.731	0.79	0.80	0.79	15.3	11.4
Proposed VSO–BiLSTM	89.6	0.798	0.85	0.87	0.86	21.5	17.8

Table 5: Failure prediction results on CI-Datasets

Method	Precision	Recall	F1-score	AUC-ROC
Random Forest	0.72	0.74	0.73	0.812
XGBoost	0.76	0.77	0.76	0.835
BiLSTM	0.79	0.81	0.80	0.851
VSO–BiLSTM + Ensemble	0.85	0.87	0.86	0.902

Error Analysis

Analysing False Negatives (FN) and false positives (FP) of the case study helps in understanding the practical implications of the misclassifications.

The framework has an average of 127 FPs in CI-Datasets (3.2% of the predictions made). With more than 1,000 open-source projects, 68% of the FPs are the result of tests labeled as "flaky" with unpredictable and inconsistent results in the past. FPs do not reduce the quality of the software, however, the FPs do result in increased operational costs as they cause unnecessary reallocation of resources and attention from the developers. The average operational cost is estimated at an additional 15–20 minutes of investigation for each FP.

The framework generated 52 FNs (1.3% of predictions). These are especially serious because missed failures can be put into production. Of the 52 FNs, 71% were “silent” failures (failed tests that did not produce wrong output). The other 29% fell into the category of edge-case failures in some infrequently executed paths that were under-trained. Assuming a real pipeline, every FN could push back the discovery of a bug by 1–3 build cycles.*Fault Severity Considerations*

The standard APFD metric treats all faults equally. However, in industry contexts, severity-weighted APFD (APFDsw) is more appropriate, as organizations prioritize faults based on their potential impact. We applied a 3-tier severity classification (Critical, Major, and Minor) derived from bug classifications in the Rails subset of CI-Datasets. Under this weighting scheme, the VSO framework's risk-based prioritization naturally correlates with fault criticality, yielding APFDsw = 0.823 compared to unweighted APFD of 0.798. This improvement confirms that the infection-based prioritization mechanism preferentially elevates severe faults. Complete evaluation with severity-weighted APFD across fully labelled enterprise datasets is intended as future work.

Resource Optimization Results

The proposed framework has proven to be economically viable in both sets of data. It has gained an Execution Time Reduction (ETR) of 21.5% in comparison to TravisTorrent, and a 20.4% in comparison to CI-Datasets, greatly speeding up CI/CD feedback cycles. Improved test prioritization and predictive allocation have optimized CPU usage by roughly 18% and memory usage by 15% which in turn improves infrastructure utilization. Optimized resources have direct economic effects as 18% monthly savings were gained from cloud resources used in Kubernetes-based CI environments. Table 6 brings these figures together for both datasets, listing the execution-time, CPU, memory, and cloud-cost savings side by side. These resource improvements are greatly needed during large scale DevOps operations where the frequency of builds and high use of infrastructure are present as the long-term effects of even small improvements are amplified.

Cost Model Transparency

Infrastructure cost savings were calculated using AWS EC2 c5.4xlarge on-demand pricing ($0.68/hour; ₹57/hour at ₹84/USD) as of March 2024. Costs were projected over 30 days based on observed resource utilization.

Formulas:

- Baseline: Cost_baseline = (Total_test_hours × $0.68) + (Storage_GB × $0.023)
- Proposed: Cost_proposed = Cost_baseline × (1 − ETR) × (1 − CPU_reduction × 0.5)

Actual savings vary based on cloud provider pricing, reserved instances, and spot usage. Reported percentages represent conservative on-demand estimates.

Statistical Significance Analysis

All results were statistically validated to ensure robustness and reliability. Each experiment was independently repeated 10 times with different random seeds (seeds = 1–10) to account for stochastic variation in VSO initialization, BiLSTM weight initialization, and data shuffling. Metrics are reported as mean ± standard deviation, and 95% bootstrap Confidence Intervals (CIs) were computed using 10,000 resamples.

Table 6: Resource optimization results

Metric	TravisTorrent	CI-Datasets
Execution Time Reduction	21.5%	20.4%
CPU Utilization Reduction	18.1%	18.3%
Memory Usage Reduction	15.2%	14.7%
Cloud Cost Savings	17.8%	18.5%

To test the statistical significance of improvements of the proposed framework over each baseline, we applied the non-parametric Wilcoxon signed-rank test on the paired per-run metric scores (n = 10 pairs per comparison). This test was selected because it makes no normality assumption and is appropriate for small, paired samples. Because four pairwise comparisons were conducted (versus GA, PSO, BiLSTM, and Ensemble-only), we applied a Bonferroni correction (α_corrected = 0.05 / 4 = 0.0125). All reported p-values remained below this corrected threshold, confirming that the observed improvements are not due to chance.

To complement significance testing with a practical-significance measure, we computed the Vargha–Delaney Â12 effect size, which estimates the probability that a randomly selected run of the proposed method outperforms a randomly selected run of the baseline. Following Vargha and Delaney's convention, Â12 > 0.71 denotes a large effect, 0.64–0.71 a medium effect, and 0.56–0.64 a small effect. All our comparisons fall in the large-effect range. Table 7 collects these statistical results, reporting the mean F1 improvement, the 95% confidence interval, the Wilcoxon p-value, the Bonferroni outcome, and the effect size for each baseline comparison.

Ablation Study

The contributions of each module were validated by ablation experiments. When using VSO by itself, it improved prioritization, but made predictions much weaker (F1 = 0.72). When BiLSTM–SVM was used by itself, it improved classification, but so little execution efficiency was achieved (ETR < 10%). When VSO and BiLSTM were used together, there was an improved balance, but there was a drop in precision. The fully integrated framework produced the best results on all metrics. Table 8 lists the ablation results on CI-Datasets for each of these configurations, making clear how much every component adds to the final scores.

Sensitivity Analysis

Sensitivity analysis was done by varying the infection rate (α) in VSO and the dropout rate in BiLSTM. Any increase of α beyond 0.4 was seen to cause APFD stability to drop due to high levels of over-prioritization of high-risk tests. The best recall was shown to be at a BiLSTM dropout of 0.2, as higher dropouts produced lower recall. Overall, the framework was stable within a range of ±10% of whatever the set parameters were.

Scalability Analysis

Scalability experiments increased the test suite size from 5,000 to 50,000 cases. The VSO–BiLSTM framework demonstrated near-linear scalability in which VSO's O(kn2) was shown to dominate the overall execution time. With a 32-core Xeon server equipped with a Tesla V100 GPU, 20,000 tests took only 14 minutes to prioritize which was an improvement over GA (35 minutes) and PSO (28 minutes) confirming that it was feasible to be used in an industrial setting.

The framework handles build surges occurring during release cycles with incremental prioritization, GPU-parallel batch predictions, and graceful degradation using lightweight heuristics during times of extreme load, and horizontal scaling across multiple pods in a Kubernetes cluster achieving up to 3 times the throughput improvement with the deployment being replicated up to 3 times.

Extended Performance Analysis

To provide a deeper understanding of framework behaviour, we extended the performance analysis along four axes: Per-project variability, convergence dynamics, runtime scaling, and failure-class stratification.

Table 7: Statistical validation results with confidence intervals and effect sizes

Comparison (Proposed vs.)	Mean F1 Improvement (± SD)	95% CI	Wilcoxon p-value	Bonferroni Significance	Vargha–Delaney Â12	Inter-pretation
Genetic Algorithm (GA)	+0.09 ± 0.012	[0.078, 0.102]	< 0.001	Significant (α=0.0125)	0.74	Large effect
Particle Swarm Optimization (PSO)	+0.07 ± 0.010	[0.061, 0.079]	< 0.001	Significant (α=0.0125)	0.72	Large effect
Standalone BiLSTM	+0.06 ± 0.009	[0.052, 0.068]	< 0.001	Significant (α=0.0125)	0.70	Large effect
Ensemble-only (no VSO)	+0.05 ± 0.008	[0.042, 0.058]	< 0.001	Significant (α=0.0125)	0.71	Large effect

Table 8: Ablation results on CI-Datasets

Configuration	FDR (%)	APFD	Precision	Recall	F1-score	ETR (%)
VSO only	81.2	0.702	0.70	0.74	0.72	14.1
BiLSTM–SVM only	84.0	0.713	0.75	0.78	0.76	8.9
VSO + BiLSTM	86.5	0.756	0.79	0.81	0.80	16.3
Full VSO–BiLSTM + Ensemble	89.1	0.793	0.84	0.86	0.85	20.4

Per-Project Variability (TravisTorrent)

Because TravisTorrent aggregates heterogeneous projects, we analyzed performance on the ten most frequently studied projects (Rails, Diaspora, Metasploit, Jekyll, Discourse, Elasticsearch-Rails, Chef, Puppet, Gitlab, and Homebrew). The APFD of the proposed framework ranged from 0.764 (Puppet) to 0.831 (Rails), with a median of 0.801. The Execution Time Reduction ranged from 17.9% to 24.6%. Lower gains on Puppet and Chef were attributable to their comparatively small test suites (< 3,000 tests), where the benefit of prioritization is structurally smaller. Projects with large, long-running suites (Rails, Gitlab) exhibited the largest gains, confirming that the framework scales favourably with test-suite size.

Convergence Dynamics

VSO converged in a mean of 38 iterations (SD = 4.1) across all datasets, compared to 72 iterations (SD = 9.3) for GA and 59 iterations (SD = 7.8) for PSO on the same hardware budget. Convergence was defined as $< 1 \times 10^{-4}$ change in the aggregate Pareto-front fitness over five consecutive iterations. This faster convergence is attributable to VSO's infection-recovery mechanism, which prunes low-influence candidates early, reducing wasted evaluations.

Runtime vs. Dataset-Size Scaling

We benchmarked the end-to-end pipeline on test-suite sizes of 5k, 10k, 20k, 50k, and 100k cases. Observed wall-clock runtimes were 4.2 min, 7.8 min, 14.0 min, 34.5 min, and 72.1 min, respectively. This scaling is consistent with the theoretical $O(k \cdot n^2)$ complexity and is approximately linear in n when the similarity matrix is stored sparsely (average degree $d \approx 40$). At the 100k scale, VSO was 2.6× faster than GA (187 min) and 1.9× faster than PSO (137 min) while achieving a higher APFD.

Failure-Class Stratification

To understand where the framework performs best and worst, we stratified all CI-Datasets predictions into four failure classes:

a) Compilation failures (deterministic, syntax-level)
b) Runtime failures (assertion and exception errors)
c) Flaky failures (non-deterministic)
d) Silent failures (failed tests that produce no explicit error signal)

Recall was 0.96 (compilation), 0.89 (runtime), 0.71 (flaky), and 0.68 (silent). The framework excels on deterministic failure classes but shows reduced recall on flaky and silent failures a limitation shared by all history-based approaches. This stratification motivates our future work on flaky-test-aware modelling.

Trade-off Discussion

The extended analysis in Table 9 reveals three practical trade-offs. First, the framework's gains are largest on projects with large, frequently failing test suites; small or highly stable suites see marginal ETR improvements (< 6%). Second, the hybrid BiLSTM–SVM pipeline incurs a fixed ~2.3-hour training cost that is amortized over weekly retraining cycles; organizations with rapidly drifting codebases may require more frequent retraining. Third, the framework's resource savings are most pronounced in cloud-native (Kubernetes) settings where elastic resource billing directly converts ETR into cost reduction; on-premise deployments will see execution-time benefits but smaller direct cost savings. These trade-offs are consistent with the behavior reported for comparable CI/CD optimization frameworks (Benjamin and Mathew, 2025; Spieker et al., 2017) and do not undermine the framework's overall utility.

Table 9: Extended Performance Analysis Summary

Analysis Dimension	Key Finding	Implication
Per-project APFD range	0.764 – 0.831 (median 0.801)	Stable across heterogeneous projects
Convergence iterations	VSO: 38 ± 4.1 vs GA: 72, PSO: 59	~2× faster convergence
100k-test runtime	VSO: 72 min vs GA: 187 min, PSO: 137 min	Feasible at industrial scale
Recall by failure class	Compilation 0.96, Runtime 0.89, Flaky 0.71, Silent 0.68	Strong on deterministic, weaker on non-deterministic
Best-case context	Large suites + cloud-native CI	Max ETR and cost savings

Discussion

Test Case Prioritization

Experimental evidence shows that prioritization based on VSO significantly enhances early fault detection over static heuristics and classical metaheuristics. VSO circumvents premature convergence, a common issue in GA and PSO, by dynamically aligning with the characteristics of the pipeline in a manner akin to viral propagation. Such adaptability is crucial in large-scale CI/CD systems due to continuously evolving test suites, which static heuristics are incapable of capturing and changing execution patterns.

Failure Prediction Accuracy

The class imbalance issue and the time-dependent nature of CI/CD test results are problems effectively tackled by the BiLSTM-SVM hybrid in conjunction with ensemble prediction. Given the safety and business implications of the CI/CD test outcomes, the majority vote approach in the ensemble classifier is crucial for reducing false negatives. When hybrid models were compared with stand-alone classifiers, the hybrid models significantly outperformed the others in precision-recall, demonstrating that the models are ready for deployment where reliability is as critical as accuracy.

Resource Optimization

In addition to enabling predictive capability, the framework provided operational efficiency that can be quantified. Execution time improved by more than 20% while CPU and memory consumption decreased by 15-18%. The operational efficiency gained translates to economic efficiency, and in Kubernetes-based CI/CD settings, operational costs are predicted to be 17-19% lower, due to the framework (Burns et al., 2022). These savings are important for large enterprises, which conduct thousands of builds per day, validating the framework's utility beyond theoretical metrics.

Statistical Reliability

Statistical reliability was determined by the Wilcoxon signed-rank test and Vargha-Delaney effect size analysis to show that the variability was not due to chance. Effect sizes were large in both datasets, along with 10 runs averaging results, which, instils high confidence in the framework's stability, reliability, consistency, and generalizability.

For effective implementation within CI/CD environments in the industry, the framework can be used as a microservice that responds to CI events, like GitHub webhooks, for instance. VSO is used to perform pre-execution prioritization while post-build failure predictions are updated by BiLSTM–SVM. Phased deployment is a solution for cold-start scenarios where prioritization is based on coverage until a set of historical data is available. The behaviour of the pipeline is changing; therefore, the model is updated weekly for the data set that covers the last 90 days. For 1,000 tests, the inference time is 45 seconds, which meets the requirements for practical latency.

Interpretability and Explainable AI

Industrial adoption demands transparent decision-making. The framework offers interpretability at three levels. At the test level, each prioritized test includes percentage contributions from failure rate, risk index, and execution time. At the build level, failure predictions highlight the top three influencing features derived from SVM support vectors, such as high complexity or recent code churn. At the trend level, weekly dashboards display prioritization stability and confidence distributions for auditing. While deep learning components remain less transparent, future work will incorporate attention mechanisms and SHAP-based explanations to enhance feature-level interpretability in CI/CD systems.

Implications and Future Directions

This research has both scientific and practical value. From a scientific perspective, it proposes a novel approach to bio-inspired optimization that, in combination with hybrid deep learning, has not been researched in CI/CD contexts. As far as prioritization and prediction are concerned, this work establishes a new direction. Practically, it offers a solution that can be deployed while simultaneously enhancing efficiency, reliability, and cost-effectiveness.

In future works, we plan to expand validation to enterprise-level closed-source datasets, explore transfer learning and domain adaptation to broaden applicability, and aim to integrate explainable AI (XAI) techniques to enhance interpretability, as it is a crucial factor for adoption in the industry.

Limitations and Generalizability

Given the evaluation of open-source datasets, which cannot be compared to an enterprise environment, the evaluation has its limitations. Large organizations tend to utilize monorepos that contain numerous interdependent services, thereby requiring the iteration of the similarity matrix to encapsulate inter-module dependencies. Proprietary build systems like Bazel or Buck may require specialized log parsing. In certain cases, employing regulated and sensitive metadata and potentially requiring differential privacy measures will be necessary when deploying systems in environments that are subject to regulatory frameworks. Additionally, enterprise repositories frequently contain far more than a million tests, thereby requiring distributed versions of the VSO to implement systems beyond the validated threshold of 100,000 tests. Although these factors do not negate the results, substantial adjustments are needed to the architectural structure, and the use of closed-source datasets will be needed to validate industrial datasets.

Conclusion

This study proposed a novel approach to improve CI/CD pipelines through the combination of the VSO, the BiLSTM–SVM hybrid, and the ensemble prediction module. The study integrates the test case prioritization and failing test case prediction. Most previous studies analyzed these problems in isolation. With the proposed

approach, there is no need to compromise system scalability or fully allocate system resources. The experimental evaluation of the proposed framework on the TravisTorrent and CI-Datasets data sets resulted in a substantial increase in the system's technical and economic efficiency. The framework achieved a fault detection rate of 89.6%, an APFD of 0.798, and an average cloud cost savings of 18%. The execution time was also reduced by more than 20%. The statistical analysis of the values above the economic and technical efficiency defined the values as practically and statistically significant. The ablation studies demonstrate that the values of each system component are significant and applicable. The operational benefits of the framework, beyond predictive accuracy, are reliable and cost-effective system deployment, reduced system resources, and improved CI/CD pipelines of modern DevOps systems. These benefits increase the potential of the system to be realistically utilized in large-scale CI/CD systems.

Next steps in this research will include validating enterprise-scale closed-source datasets, examining transfer learning and domain adaptation for more generalized use, and applying explainable AI (XAI) methodologies for improved interpretability. This body of work scientifically and practically balances the optimizations of CI/CD pipeline advancements and the more efficient, reliable, and sustainable software delivery systems.

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive comments that improved this manuscript.

Funding Information

The authors received no financial support for this research.

Authors Contributions

Dileepkumar S R: Conceptualization, methodology, software development, data analysis, write original draft.

Juby Mathew: Supervision, validation, write review and edited.

Ethics

This study used only publicly available datasets; therefore, ethical approval and informed consent were not required.

Conflict of Interest

The author declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article

Ai-Assistance Statement

The authors used ChatGPT to support language editing, text polishing, and formatting checks. All intellectual content, analyses, and conclusions are solely the work of the authors.

Reference

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A Next-generation Hyperparameter Optimization Framework. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2623–2631. https://doi.org/10.1145/3292500.3330701

Bagherzadeh, M., Kahani, N., & Briand, L. (2022). Reinforcement Learning for Test Case Prioritization. *IEEE Transactions on Software Engineering*, *48*(8), 2836–2856. https://doi.org/10.1109/tse.2021.3070549

Benjamin, J., & Mathew, J. (2025). Enhancing continuous integration predictions: a hybrid LSTM-GRU deep learning framework with evolved DBSO algorithm. *Computing*, *107*(1), 9. https://doi.org/10.1007/s00607-024-01370-2

Burns, B., Beda, J., Hightower, K., & Evenson, L. (2022). *Kubernetes: Up and Running*.

Busjaeger, B., & Xie, T. (2016). Learning for test prioritization: an industrial case study. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 975–980. https://doi.org/10.1145/2950290.2983954

DileepKumar, S. R., & Mathew, J. (2023). Ebola optimization search algorithm for the enhancement of devops and cycle time reduction. *International Journal of Information Technology*, *15*(3), 1309–1317. https://doi.org/10.1007/s41870-023-01217-7

Dileepkumar, S. R., & Mathew, J. (2025). Optimizing continuous integration and continuous deployment pipelines with machine learning: Enhancing performance and predicting failures. *Advances in Science and Technology Research Journal*, *19*(3), 108–120. https://doi.org/10.12913/22998624/197406

Dorigo, M., & Gambardella, L. M. (1997). Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, *1*(1), 53–66. https://doi.org/10.1109/4235.585892

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBert: A Pre-Trained Model for Programming and Natural Languages. *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Fraser, G., & Arcuri, A. (2011). EvoSuite: automatic test suite generation for object-oriented software. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 416–419. https://doi.org/10.1145/2025113.2025179

Gousios, G., Zaidman, A., Storey, M.-A., & Deursen, A. van. (2015). Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 358–368. https://doi.org/10.1109/icse.2015.55

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Shujie, Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., & Zhou, M. (2021). GraphCodeBERT: Pre-training code representations with data flow. *Proceedings of the 9th International Conference on Learning Representations*, 1–7. https://doi.org/10.48550/arXiv.2009.08366

He, P., Zhu, J., Zheng, Z., & Lyu, M. R. (2017). Drain: An Online Log Parsing Approach with Fixed Depth Tree. *2017 IEEE International Conference on Web Services (ICWS)*, 33–40. https://doi.org/10.1109/icws.2017.13

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*.

Leite, L., Rocha, C., Kon, F., Milojicic, D., & Meirelles, P. (2019). A Survey of DevOps Concepts and Challenges. *ACM Computing Surveys*, *52*(6), 1–35. https://doi.org/10.1145/3359981

Lin, T.-Y., Goyal, P., Girshick, R., He, K., & Dollar, P. (2017). Focal Loss for Dense Object Detection. *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017 IEEE International Conference on Computer Vision (ICCV), Venice. https://doi.org/10.1109/iccv.2017.324

Liang, J., Elbaum, S., & Rothermel, G. (2018). The Rails Dataset of Testing Results from Travis CI. *GitHub Repository*.

Najmi, A., & El-Dosuky, M. (2025). Intelligent Software Testing for Test Case Analysis Framework Using ChatGPT with Natural Language Processing and Deep Learning Integration. *Journal of Computer Science*, *21*(5), 1140–1155. https://doi.org/10.3844/jcssp.2025.1140.1155

Pan, C., & Pradel, M. (2021). Continuous test suite failure prediction. *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 553–565. https://doi.org/10.1145/3460319.3464840

Rothermel, G., Untch, R. H., Chu, C., & Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, *27*(10), 929–948. https://doi.org/10.1109/32.962562

Schwendner, D., Jungwirth, M., Gruber, M., Knoche, M., Merget, D., & Fraser, G. (2025). Practical Pipeline-Aware Regression Test Optimization for Continuous Integration. *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 371–381. https://doi.org/10.1109/icst62969.2025.10989002

Spieker, H., Gotlieb, A., Marijan, D., & Mossige, M. (2017). Reinforcement learning for automatic test case prioritization and selection in continuous integration. *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 12–22. https://doi.org/10.1145/3092703.3092709

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Proceedings of the 31st Conference on Neural Information Processing Systems*, 5998–6008.