# Study of Failures in Continuous Integration Environment

2024

The paper opens by emphasizing the role of **Continuous Integration (CI)** and **Continuous Delivery (CD)** in enabling fast and high-quality software development. The authors highlight a gap: while CI helps detect bugs and quickens releases, it also introduces failures — not only from code issues but from infrastructure, tooling, and process gaps, which are often overlooked but still delay delivery

💡
The paper seeks to raise awareness about all types of CI failures (code-related and non-code) and proposes a model (`Pram Model`) to mitigate them.

**I.Introduction :**
This section defines **Continuous Integration (CI)**, noting that it's viewed as a **practice, process, and technique**. It underscores CI's role in integrating and delivering code fast — especially in **agile environments** — but also mentions complexities, particularly when working with modular systems and dependencies.

TABLE I: MAJOR TYPES OF FAILURES IN CI ENVIRONMENT

| Sl.No. | Failure type | Failure reason |
|---|---|---|
| 1. | Compilation error | Lack of unit testing |
| 2. | Compilation error while integrating code with different module due to missing dependency | Development team Co-ordination error. |
| 3. | Failure in functional testing of software code | Lack of test on device |
| 4. | Failure in iterative testing of the software code | Lack of testing |
| 5. | Random failure in testing (not reproduce in re test | Flaky test case |
| 6. | Failure due to non-software bug | Failed in CI system |

**II. Literature Survey :**
This section reviews existing research into improving CI processes. It covers:

- The rise of Travis CI and open-source CI environments.
- Various research efforts that focus on build time reduction, failure prediction using machine learning, and test case prioritization.
- An analytical table (Table II) that categorizes studies by their focus: Build Optimization, CI Tool Optimization, and Test Optimization.

TABLE II: AREA OF RESEARCH IN CI

| Ref. Paper | Year | Build Optimization | CItool Optimization | Test Optimization | Remarks |
|---|---|---|---|---|---|
| [3] | 2022 | ✔ | ✗ | ✗ | Use machine learning to provide suggestions on suspicious CI builds. |
| [5] | 2023 | ✔ | ✗ | ✗ | proposed to improve the decision making on Pull Request (PR) submissions. |
| [7] | 2021 | ✗ | ✗ | ✔ | Used RiskTopN and RiskBatch model to reduce the number of test cycle in CI by testing in batches thereby provide substantial saving of CI time. |
| [9] | 2021 | ✗ | ✗ | ✔ | Use of optimal Reinforcement Learning (pairwise-ACER) provide a solution for test case prioritization in CI environments. |
| [10] | 2022 | ✔ | ✗ | ✔ | Proposed solution is to move toward more automated,secure, flexible infrastructure. |
| [11] | 2020 | ✔ | ✗ | ✗ | SmartBuildSkip reduced build time by 30% with only 1 extra build in 15% of failures. |
| [13] | 2019 | ✔ | ✗ | ✗ | Employs mixed-effects logistic models to analyze the effect of various factors on build durations. |
| [18] | 2023 | ✔ | ✗ | ✗ | Used machine learning (ML) for software defect prediction (SDP) where a failing build is treated as a defect to fight against. |
| [19] | 2022 | ✔ | ✗ | ✗ | Proposed SKIPCI, an approach to automatically detect commit for CI Skip based on Strength-Pareto Evolutionary Algorithm SPEA-2. |
| [20] | 2022 | ✔ | ✗ | ✗ | KOTINOS improves CI efficiency by caching build environments so that only modified files are built in CI. |
| [21] | 2019 | ✔ | ✗ | ✗ | Proposed to automate the process of identifying which commits can be skipped by CI thereby helped in reducing the number of commits in CI by 18.16%. |
| [22] | 2023 | ✔ | ✔ | ✔ | Improve cost savings and safety by combining predictions from multiple techniques to decide on skipping builds or tests with stronger certainty. Provide cost savings of 11.3% |
| [23] | 2023 | ✗ | ✗ | ✔ | Developed and evaluated feature set for training ML models for test case prioritization (TCP) in CI. Model achieved average accuracy of 85%. |
| [24] | 2022 | ✗ | ✗ | ✔ | COLEMAN technique for test Case Prioritization (TCP) takes second to prioritize a test suite, contributes efficiently and effectively to address the problem of TCP in CI. |
| [25] | 2020 | ✗ | ✗ | ✔ | Prioritize test case based on XCS classifier systems (XCS). |
| [26] | 2020 | ✗ | ✗ | ✔ | Proposed technique to Maximize the number of faults discovered in available time during testing. |
| [27] | 2023 | ✗ | ✗ | ✔ | Proposed Flakify to minimize inconsistent test cases in CI. |
| [28] | 2021 | ✔ | ✗ | ✔ | Proposed FLAST to guide test rerunning and reduce flaky tests in CI. |
| [29] | 2019 | ✗ | ✔ | ✗ | Identifies a significant portion of build breakages as noise, caused by environmental factors, cascading errors from previous builds, or allowed failures |

## III. Discussion

The authors discuss real-world scenarios where failures arise not from bad code, but from:

- Hardware issues (e.g., server downtime, machine crashes),
- Storage limits,
- Resource contention,
- Misconfigurations.

The section emphasizes that such "noisy" failures (non-code-related) lead to unnecessary delays, even though developers may have done everything right. Figure 3 and Figure 4 show

the **limited attention this problem gets in literature**. The authors advocate that treating these failures as first-class concerns can save both time and cost.

## IV. Proposed Model: Pram Model

The core contribution:
The **Pram Model** introduces an early warning system for CI environments that consists of:

1. **Continuous Training (CT):** Learning from past failures by training on logs and distinguishing between genuine and non-genuine issues.

2. **Continuous Monitoring (CM):** Live tracking of failures and anomalies even beyond software bugs (e.g., hardware, infrastructure).

The model uses real-world data from Travis Torrent for validation. It shows promising results: by predicting failures early, it can **reduce false positives** and **prevent wasted compute cycles**.

A trade-off exists: the model slightly increases build time **(2-5 minutes)** due to its learning cycle, but the authors argue this is acceptable given the robustness gains.

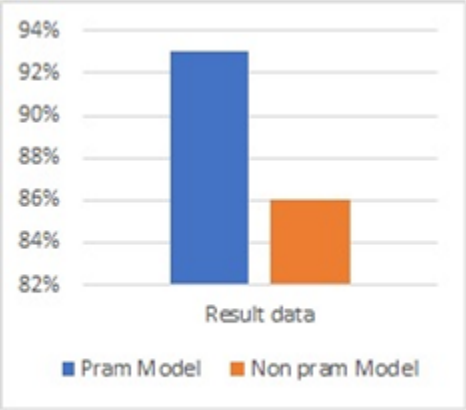


Fig. 5: Model Outcome

We evaluated data from Travis Torrent (Table III) in proposed pram model (Fig.6). The model contains two part, 1) Continuously training the model by analyzing the failure from CI through various logs file on large number of builds. These error are clustered into genuine software and other environment related issues. 2) Continuously monitoring for new failure which are in particular due to infrastructure and updating. Once the model is evaluated, we apply it back on new set of data to make sure CI predict the outcome and take the corrective action before the build is started. The outcome of this model provide us encouraging result (Fig.5).

TABLE III: TRAVIS TORRENT DATA ANALYSIS.

| Total build | Passed build | Failed build |
|---|---|---|
| 1000 | 750 | 250 |

## V. Conclusion

The conclusion reaffirms the paper's central insight: while CI has been widely optimized for software-level failures, **tool- and infrastructure-related failures are equally damaging but underexplored**.

The authors stress the need for future research into machine learning and AI approaches to create **self-healing and predictive CI systems**, and highlight the **Pram Model** as a step in that direction.

## Strengths

1. **Relevant and Timely Topic**
    - The paper addresses a genuinely important but often overlooked problem: **non-code-related failures in CI pipelines**.
    - In modern DevOps, CI/CD is ubiquitous, but much of the focus is on code quality, not on infrastructure health or tool resilience — this paper fills that gap.
2. **Clear Structure and Logical Flow**
    - The paper logically progresses from defining CI, reviewing existing literature, identifying gaps, proposing a model, and offering experimental validation.
    - It shows a solid understanding of the ecosystem by using real-world data (`Travis Torrent`).
3. **Concrete Contribution**

- The **Pram Model** is a notable attempt at adding machine learning for predictive failure detection, which shows promising potential for improving CI robustness.

- Acknowledges trade-offs (2-5 min added build time) which is honest and appreciated.

## Limitations

1. **Shallow Experimental Validation**

    - The experimental validation is brief and lacks depth.

    - The dataset is quite small for robust machine learning claims (only **1000 builds** from TravisTorrent) — this limits statistical significance.

    - No **cross-validation, precision, recall, F1-score** or even baseline comparison is presented.

2. **Lack of Algorithmic Detail**

3. **Narrow Scope**

    - The paper focuses heavily on **CI tool failures** but does not integrate other stages of DevOps, such as:
        - Continuous Deployment (CD).
        - Infrastructure as Code (IaC).

4. **No Consideration of Multi-Cloud or Hybrid Environments**

    - The paper assumes that CI is run in a relatively static environment.

    - In real-world deployments, especially in **multi-cloud or container orchestration setups (like Kubernetes)**, the failure types and detection mechanisms are different and more complex.

5. **Scalability and Generalization**

    - There's no discussion on how the Pram Model would perform at **Google-scale** or **microservices-level CI/CD pipelines**.

    - Model training on small datasets may not generalize to large heterogeneous systems.