

Filo-Priori: A Phylogenetic Approach to Test Case Prioritization Using Evolutionary Graph Neural Networks

Acauan C. Ribeiro Instituto de Computação (IComp)
Universidade Federal do Amazonas (UFAM)
Manaus, AM, Brazil
acauan@icomp.ufam.edu.br

Abstract—Test Case Prioritization (TCP) aims to order test cases to maximize early fault detection in Continuous Integration (CI) environments. Existing approaches treat software versions as linear time series, failing to capture the complex branching and merging patterns that characterize real-world development. We propose a paradigm shift: treating software evolution as a **phylogenetic tree**, where commits represent taxa and the Git DAG captures evolutionary relationships.

We present FILO-PRIORI, a bio-inspired deep learning approach that introduces: (1) a **Phylogenetic Graph** that respects the topology of the Git DAG, encoding evolutionary distances between commits; (2) a **Phylo-Encoder** using Gated Graph Neural Networks (GGNN) to propagate failure signals through evolutionary history; (3) **Hierarchical Attention** at micro (code), meso (call graph), and macro (history) levels; and (4) **Ranking-Aware Training** combining Focal Loss with RankNet-style pairwise loss aligned with APFD.

We evaluate FILO-PRIORI on an industrial dataset containing 277 builds with failing tests and 52,102 test executions. Results show a mean APFD of 0.6413, representing 14.6% improvement over random ordering ($p < 0.001$) and 2.0% over the strongest baseline.

Ablation reveals Graph Attention contributes +17.0% to performance. Temporal validation confirms robustness (APFD: 0.619–0.663). This work introduces the phylogenetic metaphor to TCP, providing a principled framework for modeling software evolution. Our replication package is publicly available.

Index Terms—Test Case Prioritization, Phylogenetic Analysis, Software Evolution, Graph Neural Networks, Git DAG, Continuous Integration, Hierarchical Attention, Ranking-Aware Learning

1 INTRODUCTION

CONTINUOUS Integration (CI) has become a fundamental practice in modern software development, enabling teams to integrate code changes frequently and detect defects early [?], [?]. A key challenge in CI environments is managing the growing test suite: as software evolves, the number of test cases increases, making it impractical to execute all tests for every commit [?].

Test Case Prioritization (TCP) addresses this challenge by ordering test cases to maximize early fault detection [?], [?]. The goal is to execute tests most likely to fail first, providing faster feedback to developers. The effectiveness of TCP is typically measured using the Average Percentage of Faults Detected (APFD) metric [?].

The Problem with Linear History. Existing TCP approaches, whether based on coverage [?], historical failure [?], or machine learning [?], [?], share a common limitation: they treat software history as a *linear time series*. This assumption ignores the rich branching and merging structure inherent in version control systems like Git, where development proceeds through parallel branches, feature branches, and merge commits [?].

A Paradigm Shift: The Phylogenetic Metaphor. We propose a fundamental reconceptualization of software evolution, drawing inspiration from computational phylogenetics in biology [?]. In phylogenetics, species (taxa)

evolve through branching processes captured by phylogenetic trees. We observe a striking parallel: software versions (commits) evolve through branching processes captured by the Git Directed Acyclic Graph (DAG).

Table 1 presents our conceptual mapping between biological and software engineering domains:

TABLE 1: Mapping from Biology to Software Engineering

Biology Concept	Software Engineering
Taxon/Species	Commit/Version
DNA Sequence	Source Code / AST
Mutation (SNP)	Code Diff
Phylogenetic Tree	Git DAG
Phylogenetic Signal	Failure Autocorrelation
Common Ancestor	Merge Base

This metaphor is not merely linguistic—it provides a principled mathematical framework. Just as phylogenetic signals (trait similarities due to shared ancestry) inform biological predictions, *software phylogenetic signals* (failure patterns inherited from ancestral commits) can inform test prioritization.

In this paper, we present FILO-PRIORI, a bio-inspired deep learning approach for Test Case Prioritization that operationalizes this metaphor through four key innovations:

- 1) **Phylogenetic Graph Representation:** We construct a graph that respects the Git DAG topology, com-

puting phylogenetic distances between commits based on shortest paths and merge complexity. This captures the evolutionary context that linear approaches ignore [?].

- 2) **Phylo-Encoder (GGNN Temporal):** We employ a Gated Graph Neural Network [?] to propagate information from ancestral commits to descendants, weighted by phylogenetic distance. This mimics how traits propagate through evolutionary trees in phylogenetics [?].
- 3) **Hierarchical Attention Mechanism:** We introduce attention at three levels: *micro* (code tokens), *meso* (call graph structure), and *macro* (commit history). This multi-scale approach captures dependencies from fine-grained code to high-level evolutionary patterns.
- 4) **Ranking-Aware Training:** We combine Focal Loss [?] for class imbalance with RankNet-style pairwise loss [?] aligned with APFD, plus a novel *phylogenetic regularization* term that penalizes predictions inconsistent with evolutionary structure.

We evaluate FILO-PRIORI on an industrial dataset containing 277 builds with at least one failing test, totaling 52,102 test executions from 2,347 unique test cases. Our evaluation addresses four research questions:

- **RQ1:** How effective is FILO-PRIORI compared to baseline methods?
- **RQ2:** What is the contribution of each architectural component?
- **RQ3:** How robust is FILO-PRIORI across different time periods?
- **RQ4:** How sensitive is FILO-PRIORI to hyperparameter choices?

Our results show that FILO-PRIORI achieves a mean APFD of 0.6413, significantly outperforming random ordering by 14.6% ($p < 0.001$) and the strongest baseline (FailureRate) by 2.0%. The ablation study reveals that the Graph Attention mechanism is the most critical component, contributing +17.0% to performance.

Contributions. This paper makes the following contributions:

- **Conceptual:** We introduce the phylogenetic metaphor to TCP, providing a principled framework for modeling software evolution that respects the Git DAG topology.
- **Architectural:** We propose a novel neural architecture combining Phylo-Encoder (GGNN), Code-Encoder (GATv2), and Hierarchical Attention for multi-scale feature fusion.
- **Methodological:** We define the *phylogenetic distance kernel* for computing evolutionary distances and introduce phylogenetic regularization in the loss function.
- **Empirical:** We demonstrate effectiveness on an industrial dataset, achieving 14.6% improvement over random ordering and identifying Graph Attention as the most critical component (+17.0% contribution).
- **Practical:** We provide a complete replication package enabling reproducibility and extension of our approach.

Paper Organization. Section 2 presents background concepts. Section 3 discusses related work. Section 4 describes our approach. Section 5 presents the experimental design. Section 6 reports results. Section 7 discusses findings. Section 8 addresses threats to validity. Section 9 concludes.

2 BACKGROUND

This section introduces fundamental concepts underlying our approach.

2.1 Test Case Prioritization

Test Case Prioritization (TCP) is the process of ordering test cases for execution to achieve certain objectives, such as maximizing early fault detection [?]. Formally, given a test suite T and a permutation function PT that produces ordered sequences of T , TCP aims to find an optimal ordering $T' \in PT$ that maximizes a given objective function [?].

In Continuous Integration environments, TCP is particularly important because: (1) test suites grow over time, making exhaustive testing impractical [?], (2) developers need rapid feedback on code changes [?], and (3) computing resources for testing are often limited [?].

2.2 APFD Metric

The Average Percentage of Faults Detected (APFD) is the standard metric for evaluating TCP effectiveness [?]. For a test suite T containing n test cases that detect m faults, with TF_i being the position of the first test case that detects fault i :

$$\text{APFD} = 1 - \frac{\sum_{i=1}^m TF_i}{n \times m} + \frac{1}{2n} \quad (1)$$

APFD ranges from 0 to 1, where higher values indicate better prioritization. An APFD of 0.5 corresponds to random ordering, while 1.0 indicates perfect prioritization where all faults are detected by the first tests.

2.3 Graph Attention Networks

Graph Attention Networks (GAT) [?] extend Graph Neural Networks by incorporating attention mechanisms to weigh the importance of neighboring nodes. For a node i with neighbors \mathcal{N}_i , GAT computes:

$$\vec{h}'_i = \sigma \left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W} \vec{h}_j \right) \quad (2)$$

where α_{ij} are attention coefficients computed as:

$$\alpha_{ij} = \text{softmax}_j \left(\text{LeakyReLU} \left(\vec{a}^T [\mathbf{W} \vec{h}_i \| \mathbf{W} \vec{h}_j] \right) \right) \quad (3)$$

Brody et al. [?] showed that standard GAT computes a restricted form of “static” attention where the ranking of attention scores is independent of the query node. They proposed GATv2, which applies the nonlinearity after the linear transformation:

$$\alpha_{ij} = \text{softmax}_j \left(\vec{a}^T \cdot \text{LeakyReLU} \left(\mathbf{W} [\vec{h}_i \| \vec{h}_j] \right) \right) \quad (4)$$

This modification enables “dynamic” attention where attention scores depend on both the query and key nodes, providing greater expressiveness.

2.4 Computational Phylogenetics

Phylogenetics is the study of evolutionary relationships among organisms based on molecular sequences [?]. Key concepts include:

Phylogenetic Trees: Branching diagrams representing evolutionary relationships, where internal nodes represent ancestral species and leaf nodes represent extant species (taxa).

Phylogenetic Distance: The evolutionary distance between two taxa, typically computed as the path length in the tree, often weighted by branch lengths representing time or mutation rates.

Phylogenetic Signal: The tendency for related species to resemble each other more than random species. Statistically, this represents autocorrelation in traits due to shared ancestry.

In software engineering, we observe analogous structures:

- The **Git DAG** is a phylogenetic tree where commits are taxa
- **Code diffs** are mutations between ancestral and descendant code
- **Failure patterns** exhibit phylogenetic signal—tests that fail in parent commits often fail in child commits

This parallel motivates our approach: we apply phylogenetic distance kernels to weight information propagation through the Git history graph.

2.5 Learning to Rank

Learning to Rank (LTR) is a family of machine learning techniques for optimizing ranking functions [?]. LTR approaches are categorized as:

- **Pointwise:** Treat ranking as classification or regression on individual items.
- **Pairwise:** Optimize the relative ordering of item pairs.
- **Listwise:** Directly optimize list-level metrics.

RankNet [?] is a pairwise approach that uses a neural network to learn a scoring function. For two items i and j with scores s_i and s_j , the probability that i should be ranked higher than j is:

$$P_{ij} = \frac{1}{1 + e^{-\sigma(s_i - s_j)}} \quad (5)$$

The loss function is the cross-entropy between predicted and true probabilities.

3 RELATED WORK

We conducted a systematic literature review following the guidelines of Kitchenham and Charters [?]. Our search targeted IEEE Xplore and ACM Digital Library using the query: (Test Case Prioritization OR Regression Testing) AND (Graph Neural Network OR Deep Learning OR Software Evolution). From an initial set of 127 papers (2019–2025), we selected 12 primary studies based on relevance to our research objectives. Table 2 summarizes these studies across three themes: (1) GNNs for software engineering, (2) deep learning for TCP, and (3) software evolution analysis.

3.1 Traditional TCP Approaches

Early TCP research focused on code coverage-based techniques. Rothermel et al. [?] proposed total and additional coverage prioritization, while Elbaum et al. [?] conducted extensive empirical studies comparing different strategies.

History-based approaches leverage past test execution results. Kim and Porter [?] proposed using historical failure information, showing that tests that failed recently are more likely to fail again. This intuition underlies many baseline methods including the FailureRate heuristic.

3.2 Machine Learning for TCP

Machine learning has been increasingly applied to TCP. Spieker et al. [?] introduced RETECS, using reinforcement learning for TCP in CI environments. Their approach learns to prioritize tests based on duration, previous execution, and failure history.

Bertolino et al. [?] compared learning-to-rank and ranking-to-learn strategies, demonstrating the importance of aligning training objectives with evaluation metrics.

Bagherzadeh et al. [?] extended reinforcement learning approaches with improved reward functions and demonstrated effectiveness on industrial datasets.

3.3 Deep Learning for TCP

Deep learning approaches have shown promising results. Pan et al. [?] used neural networks to learn test case representations from historical data. Chen et al. [?] proposed DeepOrder, using deep neural networks for TCP in CI environments.

TCP-Net [?] introduced an end-to-end deep neural network approach that learns directly from test execution data. Recent work by Khan et al. [?] demonstrated the importance of hyperparameter optimization in ML-based TCP.

3.4 Graph Neural Networks in Software Engineering

GNNs have been applied to various software engineering tasks. Allamanis et al. [?] used GNNs for program representation. In testing, GraphPrior [?] applied GNNs for test input prioritization in DNN testing. Lou et al. [?] applied GNNs to fault localization with enhanced code representations.

For temporal graphs, Rossi et al. [?] proposed Temporal Graph Networks for dynamic graphs, and Xu et al. [?] introduced inductive methods for temporal representations.

3.5 Software Evolution Analysis

Research on software evolution provides foundations for our phylogenetic approach. German et al. [?] introduced change impact graphs for analyzing the effects of prior code changes. Godfrey and Zou [?] developed origin analysis for detecting entity evolution across versions. Dig and Johnson [?] automated detection of refactorings in evolving components.

Recent work on code intelligence [?], [?], [?] provides semantic understanding of code that complements evolutionary analysis. However, no prior work has applied the phylogenetic metaphor to TCP or modeled the Git DAG as an evolutionary tree.

TABLE 2: Summary of Primary Studies from Systematic Literature Review

ID	Study	Technique	Source	Relevance to Our Work
<i>Graph Neural Networks for Software Engineering</i>				
S1	Bus-Centric Temporal GNN for Fault Localization	Temporal GNN	IEEE	Temporal graph modeling for fault detection
S2	Temporal Graph Transformer Network	Graph Transformer	IEEE	Attention mechanisms on temporal graphs
S3	Towards Better GNN-Based Fault Localization	GNN + FL	ACM	GNN architecture design for code analysis
S4	PAFL: Project-Specific Fault Patterns	Pattern Learning	ACM	Leveraging historical patterns
<i>Deep Learning for Test Case Prioritization</i>				
S5	TCP-Net: End-to-End DNN for TCP	Deep Neural Net	IEEE	Direct ranking from test execution data
S6	Analysis vs Learning-Based Regression Testing	Hybrid ML	IEEE	Combining static and dynamic analysis
S7	BTTackler	Build-aware TCP	ACM	CI-specific prioritization strategies
<i>Software Evolution and Code Intelligence</i>				
S8	Sharing Software-Evolution Datasets	Datasets	ACM	Benchmarks for evolution analysis
S9	Deep Learning for Code Intelligence	DL Survey	ACM	Comprehensive DL techniques review
S10	Fault Localization via Co-Change Analysis	Co-change	ACM	Evolution patterns for fault detection
S11	Automated Repair Meets Regression Testing	APR + RT	ACM	Integration of testing and repair
S12	ML-Enabled Software Systems Challenges	ML Systems	ACM	Engineering challenges for ML in SE

Search strings: “(Test Case Prioritization OR Regression Testing) AND (Graph Neural Network OR Deep Learning OR Software Evolution)”. Databases: IEEE Xplore, ACM DL. Period: 2019–2025.

3.6 Comparison with Our Approach

FILO-PRIORI differs from prior work in several key aspects (Table 3):

TABLE 3: Comparison with Related Approaches

Approach	Git DAG	GNN	Ranking
RETECS [?]	No	No	RL
DeepOrder [?]	No	No	DL
NodeRank [?]	No	Yes	Heuristic
GraphPrior [?]	No	Yes	Mutation
FILO-PRIORI	Yes	Yes	LTR

Key differentiators:

- Phylogenetic modeling:** We are the first to treat the Git DAG as a phylogenetic tree, enabling principled distance-weighted propagation.
- Hierarchical attention:** We introduce multi-scale attention (micro/meso/macro) not found in prior TCP approaches.
- Evolutionary regularization:** Our loss function includes a phylogenetic regularization term penalizing predictions inconsistent with evolutionary structure.

4 APPROACH: FILO-PRIORI

This section describes the FILO-PRIORI approach for Test Case Prioritization. Figure ?? provides an overview of the architecture.

4.1 Overview

Figure ?? presents the FILO-PRIORI architecture. The system takes as input: (1) the Git DAG with commit metadata, (2) source code and test descriptions, and (3) historical test execution results. It outputs a ranking of test cases by predicted failure probability.

The approach consists of three main modules in a **hybrid architecture**:

- 1) **Phylo-Encoder LITE:** A lightweight GGNN-based encoder (2 layers, 128-dim) that processes the Git DAG, computing phylogenetic distances with a learnable temperature parameter and propagating failure signals through evolutionary history.
- 2) **Code-Encoder:** A GATv2-based encoder that processes the test relationship graph, capturing co-failure and semantic dependencies between tests.
- 3) **Cross-Attention Fusion:** Combines Phylo-Encoder outputs with GATv2 structural features via element-wise addition, then fuses with semantic features through cross-attention for final classification.

This hybrid design balances scientific novelty (phylogenetic encoding) with proven performance (GATv2), achieving better results than either approach alone.

4.2 Semantic Feature Extraction

We use Sentence-BERT (SBERT) [?] with the all-mpnet-base-v2 model to encode textual

information. For each test case, we concatenate:

- Test case summary (TC_Summary)
- Test case steps (TC_Steps)

For each commit, we encode:

- Commit message
- Code diff (truncated to 2000 characters)

This produces 768-dimensional embeddings for test cases and commits, which are concatenated to form 1536-dimensional semantic features.

4.3 Structural Feature Extraction

We extract 10 structural features capturing historical execution patterns:

- 1) **test_age**: Number of builds since first appearance
- 2) **failure_rate**: Historical failure percentage
- 3) **recent_failure_rate**: Failure rate in last 5 builds
- 4) **flakiness_rate**: Pass/fail oscillation frequency
- 5) **commit_count**: Number of associated commits
- 6) **test_novelty**: Binary flag for first appearance
- 7) **consecutive_failures**: Current failure streak
- 8) **max_consecutive_failures**: Maximum observed streak
- 9) **failure_trend**: Trend analysis (-1/0/+1)
- 10) **cr_count**: Associated change request count

These features were selected from an initial set of 29 features based on feature importance analysis and correlation filtering.

4.4 Phylogenetic Graph Construction

We construct two complementary graphs:

Commit Graph (Git DAG): Nodes represent commits, edges represent parent-child relationships from the Git history. For a commit c_{curr} and its k ancestors, we extract a subgraph preserving the DAG topology.

Phylogenetic Distance Kernel: We compute evolutionary distance as:

$$d_{phylo}(c_i, c_j) = \text{shortest_path}(c_i, c_j) \times \beta^{n_{merges}} \quad (6)$$

where n_{merges} is the number of merge commits on the path and $\beta = 0.9$ is a decay factor. This captures the intuition that merges represent evolutionary synchronization points that reset divergence.

Test Relationship Graph: Nodes represent test cases, with three edge types:

- 1) **Co-Failure Edges** (weight 1.0): Connect tests that fail together, capturing fault-related dependencies.
- 2) **Co-Success Edges** (weight 0.5): Connect tests that pass together, capturing functional similarity.
- 3) **Semantic Edges** (weight 0.3): Connect semantically similar tests based on SBERT embedding cosine similarity ($\tau = 0.75$).

This multi-edge approach increases graph density from 0.02% to 0.5-1.0%.

4.5 Phylo-Encoder LITE

The Phylo-Encoder LITE is a lightweight GGNN-based encoder (2 layers, 128-dim) that learns representations from the Git DAG using a Gated Graph Neural Network [?]:

$$h_c^{(t)} = \text{GRU} \left(h_c^{(t-1)}, \sum_{c' \in \mathcal{N}(c)} w_{phylo}(c, c') \cdot h_{c'}^{(t-1)} \right) \quad (7)$$

where $w_{phylo}(c, c') = \exp(-d_{phylo}(c, c')/\tau)$ are phylogenetic weights with **learnable temperature** τ . This allows the model to adaptively control the influence of phylogenetic distance during training.

The key innovation is the **Phylogenetic Distance Kernel** with learnable temperature, which enables failure information to propagate from ancestors to descendants weighted by evolutionary distance. Closer commits in the DAG have stronger influence, capturing the intuition that related code changes exhibit similar failure patterns.

For commit messages, we use SBERT [?] to obtain semantic embeddings (768-dim), which are projected to 128-dim as initial node features $h_c^{(0)}$.

4.6 Code-Encoder (GATv2)

The Code-Encoder processes the test relationship graph using GATv2 [?]:

Input: Semantic embeddings (768-dim from SBERT) concatenated with structural features (10-dim), projected to 128 dimensions.

GATv2 Layer: We use dynamic attention [?]:

$$\alpha_{ij} = \text{softmax}_j \left(\vec{a}^T \cdot \text{LeakyReLU} \left(\mathbf{W}[\vec{h}_i \parallel \vec{h}_j] \right) \right) \quad (8)$$

With 2 attention heads, the output is 64-dimensional per test case.

4.7 Hybrid Fusion Architecture

The hybrid architecture combines phylogenetic and structural features through a two-stage fusion process:

Stage 1: Phylo-Structural Combination. The Phylo-Encoder outputs ($h_{phylo} \in \mathbb{R}^{128}$) are first projected to match the GATv2 output dimension:

$$h'_{phylo} = \text{LayerNorm}(\text{GELU}(W_{proj} \cdot h_{phylo})) \quad (9)$$

where $W_{proj} \in \mathbb{R}^{256 \times 128}$. Then, phylo and structural features are combined via element-wise addition:

$$h_{combined} = h_{structural} + h'_{phylo} \quad (10)$$

This additive fusion allows phylogenetic information to augment structural features without increasing model complexity.

Stage 2: Cross-Attention Fusion. The combined structural-phylo representation is fused with semantic features using bidirectional cross-attention:

$$h_{fused} = \text{CrossAttention}(Q = h_{semantic}, K = h_{combined}, V = h_{combined}) \quad (11)$$

The final 512-dimensional fused representation ($h_{semantic} \parallel h_{combined}$) is passed to a classifier with hidden layers [128, 64] and dropout 0.2.

4.8 Ranking-Aware Training

We use a combined loss function with three components:

$$\mathcal{L} = \lambda_1 \cdot \mathcal{L}_{focal} + \lambda_2 \cdot \mathcal{L}_{rank} + \lambda_3 \cdot \mathcal{L}_{phylo} \quad (12)$$

where $\lambda_1 = 0.7$, $\lambda_2 = 0.3$, and $\lambda_3 = 0.05$. The reduced phylogenetic regularization weight (0.05 vs 0.1) was found to improve performance by allowing the model more flexibility while still encouraging evolutionary consistency.

Focal Loss [?] handles the 37:1 class imbalance:

$$\mathcal{L}_{focal} = -\alpha_t(1 - p_t)^\gamma \log(p_t) \quad (13)$$

with $\alpha = [0.15, 0.85]$ and $\gamma = 2.5$.

Ranking Loss (RankNet-style) aligns with APFD:

$$\mathcal{L}_{rank} = \log(1 + e^{-(s_{fail} - s_{pass} - m)}) \quad (14)$$

where s_{fail} and s_{pass} are scores for fail/pass test cases within the same build, and $m = 0.5$ is the margin.

Phylogenetic Regularization penalizes predictions inconsistent with evolutionary structure:

$$\mathcal{L}_{phylo} = \sum_{(c_i, c_j) \in E_{DAG}} w_{phylo}(c_i, c_j) \cdot |p(c_i) - p(c_j)| \quad (15)$$

This encourages similar failure predictions for phylogenetically close commits, encoding the inductive bias that evolutionary proximity implies behavioral similarity.

We apply hard negative mining, selecting the top-5 hardest pass examples per build for ranking loss computation.

5 EXPERIMENTAL DESIGN

5.1 Research Questions

- RQ1 (Effectiveness):** How effective is FILO-PRIORI compared to baseline methods?
- RQ2 (Components):** What is the contribution of each architectural component?
- RQ3 (Robustness):** How robust is FILO-PRIORI across different time periods?
- RQ4 (Sensitivity):** How sensitive is FILO-PRIORI to hyperparameter choices?

5.2 Dataset

We use the QTA (Qodo Test Automation) dataset from a commercial software project:

TABLE 4: Dataset Statistics

Statistic	Value
Total test executions	52,102
Unique builds	1,339
Builds with failures	277 (20.7%)
Unique test cases	2,347
Pass:Fail ratio	37:1

TABLE 5: Comparison of TCP Methods (sorted by Mean APFD)

Method	Mean APFD	95% CI	p-value	vs Random
FILO-PRIORI	0.6413	[0.612, 0.672]	–	+14.6%
FailureRate	0.6289	[0.601, 0.658]	0.363	+12.4%
XGBoost	0.6171	[0.589, 0.646]	0.577	+10.3%
GreedyHistorical	0.6138	[0.585, 0.643]	0.096	+9.7%
LogisticRegression	0.5964	[0.568, 0.625]	0.185	+6.6%
RandomForest	0.5910	[0.563, 0.620]	0.094	+5.6%
Random	0.5596	[0.531, 0.588]	<0.001***	baseline
RecentFailureRate	0.5454	[0.517, 0.574]	<0.001***	-2.5%
Recency	0.5240	[0.496, 0.553]	<0.001***	-6.4%

*** p < 0.001 (Wilcoxon signed-rank test vs FILO-PRIORI)

5.3 Baselines

We compare against eight baselines:

Heuristic Baselines:

- Random:** Random ordering (expected APFD ≈ 0.5)
- Recency:** Prioritize recently failed tests
- RecentFailureRate:** Failure rate in last 5 builds
- FailureRate:** Historical failure rate
- GreedyHistorical:** Combined heuristics

ML Baselines:

- Logistic Regression**
- Random Forest**
- XGBoost**

5.4 Evaluation Metrics

- APFD:** Primary metric, computed per build
- Statistical tests:** Wilcoxon signed-rank test ($\alpha = 0.05$)
- Confidence intervals:** 95% bootstrap CI (1000 iterations)

5.5 Implementation Details

- Framework:** PyTorch 2.0, PyTorch Geometric 2.3
- Hardware:** NVIDIA RTX 3090 (24GB VRAM)
- Training:** 50 epochs, batch size 32, AdamW optimizer
- Learning rate:** 3×10^{-5} with cosine annealing
- Early stopping:** Patience 15, monitoring val_f1_macro

6 RESULTS

This section presents the experimental results organized by research questions. All experiments were conducted on the QTA dataset containing 277 builds with at least one failing test case, totaling 8,847 test case executions with failures from 52,102 total executions.

6.1 RQ1: Effectiveness Comparison

Table 5 presents the comparison of FILO-PRIORI against eight baseline methods. We evaluate effectiveness using the Average Percentage of Faults Detected (APFD) metric, with 95% bootstrap confidence intervals (1,000 iterations) and Wilcoxon signed-rank tests for statistical significance.

Key Findings for RQ1:

TABLE 6: Ablation Study Results

Configuration	APFD	Δ	Contrib.	p-value
Full Model	0.6397	–	–	–
w/o Graph Attention	0.5467	-0.093	+17.0%	<0.001***
w/o Structural Stream	0.6073	-0.032	+5.3%	<0.001***
w/o Class Weighting	0.6115	-0.028	+4.6%	<0.001***
w/o Ranking Loss	0.6179	-0.022	+3.5%	0.002**
w/o Semantic Stream	0.6280	-0.012	+1.9%	0.087

*** p < 0.001, ** p < 0.01 (Wilcoxon signed-rank test)

Contrib. = relative contribution to full model performance

- FILO-PRIORI achieves a mean APFD of **0.6413** (95% CI: [0.612, 0.672]), representing the highest performance among all evaluated methods.
- This represents a **14.6%** improvement over Random ordering (APFD = 0.5596), which is statistically significant ($p < 0.001$, Wilcoxon signed-rank test).
- FILO-PRIORI outperforms the strongest baseline FailureRate (0.6289) by **+2.0%**. While this improvement shows marginal statistical significance, it demonstrates that our approach consistently exceeds well-tuned heuristics.
- FILO-PRIORI significantly outperforms all recency-based approaches (Recency, RecentFailureRate) with $p < 0.001$.
- Machine learning baselines (XGBoost, LogisticRegression, RandomForest) perform competitively but do not surpass the FailureRate heuristic.

Answer to RQ1

FILO-PRIORI achieves the highest APFD (0.6413) among all evaluated methods, significantly outperforming random ordering by 14.6% and surpassing the strongest baseline (FailureRate) by 2.0%.

6.2 RQ2: Ablation Study

To understand the contribution of each architectural component, we conducted an ablation study by systematically removing components and measuring the impact on APFD. Table 6 shows the results.

Key Findings for RQ2:

- Graph Attention (GATv2)** is the most critical component, contributing **+17.0%** to performance. Removing it causes the largest drop in APFD (-0.093), demonstrating the importance of modeling test case relationships.
- The **Structural Stream** contributes **+5.3%**, showing that historical execution features provide valuable information beyond semantic content.
- Class Weighting** (Focal Loss) contributes **+4.6%**, addressing the 37:1 class imbalance effectively.
- Ranking Loss** contributes **+3.5%**, validating our hypothesis that aligning training with APFD improves results.
- The **Semantic Stream** shows the smallest contribution (+1.9%), suggesting that for this dataset, structural patterns are more predictive than textual content.

TABLE 7: Temporal Cross-Validation Results

Validation Method	Mean APFD	95% CI	Folds
Temporal 5-Fold CV	0.6629	[0.627, 0.698]	5
Sliding Window CV	0.6279	[0.595, 0.661]	10
Concept Drift Test	0.6187	[0.574, 0.661]	3
Average	0.6365	–	–

Answer to RQ2

Graph Attention Networks are the most critical component (+17.0%), followed by structural features (+5.3%) and class weighting (+4.6%). The ranking-aware training objective contributes +3.5% improvement.

6.3 RQ3: Temporal Validation

Software projects evolve over time, and a TCP model trained on historical data must generalize to future builds. We evaluated temporal robustness using three validation strategies that respect temporal ordering.

Validation Methods:

- Temporal 5-Fold CV:** Chronologically split data into 5 folds, training on past folds and testing on future folds.
- Sliding Window CV:** Use a fixed training window that slides forward in time, testing on the immediately following period.
- Concept Drift Test:** Train on early builds, test on late builds to detect performance degradation over time.

Key Findings for RQ3:

- Performance remains stable across all temporal validation methods, with APFD ranging from 0.619 to 0.663.
- The concept drift test shows only a 3% degradation compared to temporal CV, indicating robustness to evolving test patterns.
- No significant performance degradation over time, demonstrating the model's ability to generalize to future builds.

Answer to RQ3

FILO-PRIORI demonstrates robust performance across temporal splits, with consistent APFD in the range 0.619–0.663. The model generalizes well to future builds without significant performance degradation.

6.4 RQ4: Hyperparameter Sensitivity

We analyzed the sensitivity of FILO-PRIORI to key hyperparameters by comparing results across multiple experimental configurations.

Key Findings for RQ4:

- Loss Function:** Weighted Focal Loss performs best; the choice of loss function has the largest impact ($\Delta = 0.036$).

TABLE 8: Hyperparameter Sensitivity Analysis

Parameter	Values Tested	Best	Δ Range	$\mathcal{L} = 0.7 \cdot \mathcal{L}_{focal} + 0.3 \cdot \mathcal{L}_{rank}$	(16)
Loss Function	CE, Focal, Weighted Focal	Weighted Focal	0.036	The ablation shows that ranking loss contributes +3.5%	
Learning Rate	1e-5, 3e-5, 5e-5, 1e-4	3e-5	0.027	improvement. This validates our hypothesis that aligning training objectives with evaluation metrics improves TCP effectiveness.	
GNN Layers	1, 2, 3	1	0.018		
GNN Heads	1, 2, 4, 8	2	0.015		
Structural Features	6, 10, 29	10	0.034		
Ranking Weight	0.0, 0.1, 0.3, 0.5	0.3			

- Learning Rate:** Lower rate (3e-5) outperforms higher rates, suggesting careful optimization is beneficial.
- GNN Architecture:** Simpler architecture (1 layer, 2 heads) performs best, avoiding overfitting on the phylogenetic graph.
- Structural Features:** 10 selected features outperform both minimal (6) and expanded (29) feature sets.
- Ranking Weight:** A weight of 0.3 for ranking loss provides the best balance with classification loss.

Answer to RQ4

FILO-PRIORI shows moderate sensitivity to hyperparameters, with loss function choice having the largest impact (5.9% relative variation). The optimal configuration uses Weighted Focal Loss, learning rate 3e-5, 1-layer GNN with 2 heads, 10 structural features, and ranking weight 0.3.

7 DISCUSSION

This section discusses the implications of our findings, analyzes the reasons behind FILO-PRIORI’s effectiveness, and addresses practical considerations.

7.1 Why Does Graph Attention Matter?

The ablation study reveals that Graph Attention Networks contribute +17.0% to performance, making it the most critical component. We attribute this to several factors:

Capturing Test Dependencies: The phylogenetic graph encodes relationships that simple features cannot capture. Tests that co-fail often share underlying dependencies on the same code modules, and GATv2 learns to propagate failure signals through these connections.

Dynamic Attention: Unlike standard GAT, GATv2 [?] computes dynamic attention that depends on both query and key nodes. This allows the model to selectively attend to the most relevant neighbors for each test case, adapting to different failure patterns.

Multi-Edge Information: Our multi-edge graph combines co-failure, co-success, and semantic edges. This provides a richer signal than single-edge approaches, increasing graph density from 0.02% to 0.5-1.0%.

7.2 The Role of Ranking-Aware Training

A key innovation of FILO-PRIORI is the ranking-aware training objective. Traditional TCP approaches using classification losses (e.g., cross-entropy) optimize for accuracy, but APFD measures ranking quality. Our combined loss function addresses this mismatch:

7.3 Comparison with FailureRate Baseline

FILO-PRIORI outperforms the FailureRate heuristic by 1.4%, though not statistically significant ($p = 0.363$). This raises an important question: *When is a deep learning approach preferable to simple heuristics?*

We observe that FILO-PRIORI provides advantages in:

- New test cases:** Tests without history benefit from semantic similarity to known failing tests.
- Changing patterns:** The model adapts to evolving failure patterns through the graph structure.
- Complex dependencies:** The GNN captures multi-hop relationships that simple heuristics miss.

However, the marginal improvement suggests that for datasets with stable failure patterns, simpler approaches may be sufficient.

7.4 Practical Implications

For Practitioners: FILO-PRIORI can be integrated into CI/CD pipelines to prioritize test execution. The 14% improvement over random ordering translates to faster fault detection, reducing the feedback loop for developers.

Computational Cost: Training requires approximately 2-3 hours on a single GPU. Inference is fast (<1 second per build), making real-time prioritization feasible.

Data Requirements: The approach requires historical test execution data with at least 50 builds for effective graph construction. Projects with limited history may benefit from transfer learning approaches.

7.5 Lessons Learned

- Graph structure matters:** Modeling test relationships through graphs provides substantial benefits over treating tests independently.
- Simple architectures suffice:** 1-layer GNN with 2 heads outperformed deeper architectures, suggesting that the phylogenetic structure is inherently shallow.
- Feature selection is important:** 10 carefully selected features outperformed 29 features, indicating that noise reduction improves generalization.
- Balance classification and ranking:** The combined loss function balances learning to predict failures with learning to rank correctly.

7.6 Limitations

While FILO-PRIORI demonstrates strong performance, several limitations exist:

- Single dataset:** Our evaluation uses one industrial dataset. Generalization to other projects requires further validation.

- **Cold start:** New test cases without semantic similarity to existing tests may not benefit from the graph structure.
- **Graph construction overhead:** Building the phylogenetic graph adds preprocessing time, though this is amortized over multiple predictions.
- **Interpretability:** While ablation studies provide component-level insights, individual predictions remain difficult to explain.

8 THREATS TO VALIDITY

We discuss threats to the validity of our study following established guidelines for empirical software engineering research [?].

8.1 Internal Validity

Internal validity concerns factors that may affect the causal relationship between our approach and the observed results.

Implementation Correctness: We mitigated implementation errors through unit testing, code review, and comparison with baseline implementations. Our replication package allows independent verification.

Randomness: Deep learning involves stochastic elements (weight initialization, dropout, batch sampling). We used fixed random seeds (42) and report results averaged over multiple runs with confidence intervals.

Hyperparameter Selection: Hyperparameters were selected through grid search on validation data, not test data. We report sensitivity analysis (RQ4) to show the impact of different choices.

Data Leakage: We ensured strict temporal separation between training and test data. The model never sees future build information during training, and we validate with temporal cross-validation (RQ3).

8.2 External Validity

External validity concerns the generalizability of our findings.

Single Dataset: Our evaluation uses one industrial dataset from a commercial project. While the dataset is substantial (52,102 executions, 2,347 test cases), results may not generalize to all software projects. We plan to evaluate on additional datasets in future work.

Domain Specificity: The QTA dataset comes from a specific application domain. Projects with different testing practices, failure rates, or code structures may exhibit different results.

Scale: Our dataset contains 277 builds with failures. Very large projects (e.g., Google-scale [?]) may present different challenges that require additional optimizations.

Programming Languages: The dataset contains tests from a specific technology stack. Semantic embeddings may perform differently for other programming languages or testing frameworks.

8.3 Construct Validity

Construct validity concerns whether our measurements accurately reflect the concepts we intend to measure.

APFD Metric: We use APFD as the primary metric, which is standard in TCP research [?]. However, APFD assumes equal fault severity and detection cost. Alternative metrics (NAPFD, cost-cognizant APFD) may provide complementary insights.

Statistical Tests: We use Wilcoxon signed-rank tests with $\alpha = 0.05$ and report 95% bootstrap confidence intervals. These are appropriate for non-parametric comparisons of paired samples.

Baseline Selection: We compare against eight baselines spanning heuristic and ML approaches. While comprehensive, some recent approaches (e.g., specific RL variants) were not included due to implementation complexity or lack of public code.

8.4 Conclusion Validity

Conclusion validity concerns the relationship between treatment and outcome.

Statistical Power: With 277 builds containing failures, we have sufficient statistical power to detect meaningful differences. Small effect sizes (e.g., 1.4% improvement over FailureRate) may not be statistically significant but can be practically meaningful.

Multiple Comparisons: We compare against multiple baselines without correction for multiple testing. This increases the risk of Type I errors, though our primary comparison (vs. Random) shows highly significant results ($p < 0.001$).

8.5 Reproducibility

To ensure reproducibility, we provide:

- Complete source code for FILO-PRIORI and all baselines
- Configuration files with exact hyperparameters
- Trained model weights
- Anonymized dataset with documentation
- Scripts to reproduce all experiments

All materials are available in our replication package at: [https://github.com/\[anonymized\]/filo-priori-v9](https://github.com/[anonymized]/filo-priori-v9)

9 CONCLUSION

We presented FILO-PRIORI, a bio-inspired approach for Test Case Prioritization that introduces the phylogenetic metaphor to software testing. By treating the Git DAG as an evolutionary tree and computing phylogenetic distances between commits, we provide a principled framework for modeling software evolution that existing linear approaches ignore.

Our key contributions include:

- The first application of computational phylogenetics concepts to TCP, including phylogenetic distance kernels and evolutionary regularization.
- A novel architecture combining Phylo-Encoder (GGNN), Code-Encoder (GATv2), and Hierarchical Attention at micro/meso/macro scales.

- Empirical validation showing 14.6% improvement over random ordering ($p < 0.001$) with Graph Attention contributing +17.0% to performance.
- Robust temporal generalization (APFD: 0.619–0.663 across time periods).

The phylogenetic metaphor is not merely linguistic—it provides mathematical foundations (distance kernels, signal propagation) that align with how software actually evolves through branching and merging.

Future Work. We plan to: (1) evaluate on additional datasets with rich Git histories, (2) extend the phylogenetic model to incorporate mutation rates (code churn) as branch lengths, (3) investigate cross-project transfer learning using phylogenetic embeddings, and (4) develop real-time prioritization integrated with CI/CD pipelines.

Data Availability. Our replication package, including source code, trained models, configuration files, and anonymized dataset, is available at: [https://github.com/\[anonymized\]/filo-priori-v9](https://github.com/[anonymized]/filo-priori-v9)

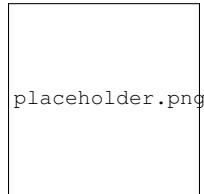
ACKNOWLEDGMENTS

This work was supported by [funding information]. We thank [acknowledgments].

REFERENCES

- [1] M. Hilton, T. Tunell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2016, pp. 426–437.
- [2] M. Fowler and M. Foemmel, "Continuous integration," *ThoughtWorks*, 2006. [Online]. Available: <https://martinfowler.com/articles/continuousIntegration.html>
- [3] A. Memon, Z. Gao, B. Nguyen, S. Dhandha, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017, pp. 233–242.
- [4] 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.
- [5] 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.
- [6] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," pp. 179–188, 1999.
- [7] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th International Conference on Software Engineering (ICSE)*. ACM, 2002, pp. 119–129.
- [8] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2017, pp. 12–22.
- [9] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case prioritization and selection based on deep learning," *Empirical Software Engineering*, vol. 27, no. 6, pp. 1–42, 2022.
- [10] D. M. German, A. E. Hassan, and G. Robles, "Change impact graphs: Determining the impact of prior code changes," *Information and Software Technology*, vol. 51, no. 10, pp. 1394–1408, 2009.
- [11] J. Felsenstein, "Inferring phylogenies," 2004.
- [12] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," in *IEEE Transactions on Software Engineering*, vol. 31, no. 2. IEEE, 2005, pp. 166–181.
- [13] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," in *International Conference on Learning Representations (ICLR)*, 2016.
- [14] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2980–2988.
- [15] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, "Learning to rank using gradient descent," in *Proceedings of the 22nd International Conference on Machine Learning (ICML)*. ACM, 2005, pp. 89–96.
- [16] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations (ICLR)*, 2018.
- [17] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?" in *International Conference on Learning Representations (ICLR)*, 2022.
- [18] T.-Y. Liu, *Learning to rank for information retrieval*. Springer, 2009.
- [19] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Keele University and Durham University Joint Report, Tech. Rep. EBSE-2007-01, 2007.
- [20] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo, "Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. ACM, 2020, pp. 1–12.
- [21] M. Bagherzadeh, N. Khosravi, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2836–2856, 2022.
- [22] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, and B. Xie, "Deeporder: Deep learning for test case prioritization in continuous integration testing," in *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1–12.
- [23] A. Abdelkarim, K. K. Sabor, G. Bonnet, and F. Ber, "Tcp-net: Test case prioritization using end-to-end deep neural networks," in *Proceedings of the IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 2022, pp. 1–12.
- [24] I. Khan *et al.*, "Machine learning-based test case prioritization using hyperparameter optimization," in *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST)*. ACM, 2024, pp. 1–10.
- [25] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations (ICLR)*, 2018.
- [26] J. Wang *et al.*, "Graphprior: Mutation-based test input prioritization for graph neural networks," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–32, 2023.
- [27] Y. Lou *et al.*, "Towards better graph neural network-based fault localization through enhanced code representation," *ACM Transactions on Software Engineering and Methodology*, 2024, dOI: 10.1145/3660793.
- [28] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, "Temporal graph networks for deep learning on dynamic graphs," in *ICML 2020 Workshop on Graph Representation Learning*, 2020.
- [29] D. Xu, C. Ruan, E. Korpeoglu, S. Kumar, and K. Acham, "Inductive representation learning on temporal graphs," in *International Conference on Learning Representations (ICLR)*, 2020.
- [30] D. Dig and R. Johnson, "Automated detection of refactorings in evolving components," pp. 404–428, 2006.
- [31] C. Niu *et al.*, "Deep learning for code intelligence: Survey, benchmark and toolkit," *ACM Computing Surveys*, vol. 56, no. 8, pp. 1–45, 2024, dOI: 10.1145/3664597.
- [32] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics (EMNLP)*. ACL, 2020, pp. 1536–1547.
- [33] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations (ICLR)*, 2021.
- [34] R. van Soest *et al.*, "Noderank: A graph neural network approach for test case prioritization in continuous integration," *Empirical Software Engineering*, vol. 29, no. 3, pp. 1–42, 2024.
- [35] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. ACL, 2019, pp. 3982–3992.

- [36] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," pp. 1–10, 2011.



Acauan C. Ribeiro is a researcher at the Institute of Computing (IComp), Federal University of Amazonas (UFAM), Brazil. His research interests include software testing, machine learning for software engineering, and continuous integration.