IIETA International Information and Engineering Technology Association
Advancing the World of Information and Engineering

Mathematical Modelling of Engineering Problems
Vol. 12, No. 9, September, 2025, pp. 3003-3012
Journal homepage: http://iieta.org/journals/mmep

XGRL-TCP: An Explainable Graph-Based Reinforcement Learning Framework for Test Case Prioritization in CI

Srinivasa Rao Kongarana*, Ananda Rao Akepogu, Radhika Raju P

Department of CSE, College of Engineering, JNTUA, Ananthapur 515002, India

Corresponding Author Email: srinivas.cst4@gmail.com

https://doi.org/10.18280/mmep.120905

Received: 13 June 2025
Revised: 11 August 2025
Accepted: 18 August 2025
Available online: 30 September 2025

Keywords:
graph attention network, CodeBERT, actor-critic reinforcement learning, explainable AI, Average Percentage of Faults Detected (APFD), Defects4J, software testing efficiency

ABSTRACT

Continuous integration demands fast and trustworthy fault discovery from large, frequently changing test suites. Many test case prioritization (TCP) methods underperform in this setting because they ignore code-test structure, overlook the semantics of changes, and provide little transparency into ranking decisions. XGRL TCP addresses these gaps with a graph attention network over the test–code dependency graph, CodeBERT embeddings for commit diffs and test text, and an attention-augmented actor–critic reinforcement learner. On the Defects4J, it achieves an Average Percentage of Faults Detected (APFD) 89.3%, a Fault Detection Rate (FDR) of 85.4%, and a Time to First Fault (TTFF) of 6.3 tests. Key contributions include: (i) a unified structural plus semantic state for TCP, (ii) online adaptation across CI cycles, and (iii) built-in explanatory signals from graph and policy attentions. Compared with contemporary methods TCP-TB and TCP-CIC, XGRL-TCP consistently increases APFD/FDR and reduces TTFF across projects and early execution budgets. Explanations highlight influential code regions and neighboring tests that drive each selection, improving auditability and trust during CI. The approach introduces a modest computational cost: 5.6% overhead, which corresponds to about 33.6 s for a 10-minute suite, leaving typical CI time budgets largely intact.

1. INTRODUCTION

Continuous integration (CI) executes many builds per day, which multiplies test executions and tightens feedback budgets [1]. Running complete suites on every build is infeasible for large systems and slows development cycles [2]. Test case prioritization (TCP) addresses this by ordering tests to expose faults early, but CI settings require approaches that adapt quickly and operate with modest overhead. Lack of transparency also hinders adoption; engineers need to understand why certain tests run first [3]. An effective CI-oriented TCP solution must therefore be adaptive, efficient under tight time budgets, and interpretable.

Despite progress, three concrete gaps persist. (G1) Missing structure. Many ML/RL TCP methods treat tests as independent items and omit explicit modeling of test–code dependencies, losing context needed for CI; graph-based representations address this but remain underused in TCP [4-6]. (G2) Missing semantics. Lexical or shallow IR signals (e.g., token overlap) overlook the meaning of code changes and tests, limiting fault revelation; semantic modeling with code transformers has shown measurable gains (e.g., SatTCP), but is not standard in TCP pipelines [7, 8]. (G3) Missing explainability. State-of-the-art techniques often behave as black boxes, offering little decision rationale, which impedes trust and diagnosis in CI [3].

This work introduces XGRL-TCP, a CI-oriented TCP framework that addresses G1–G3 with the following contributions:

- Unified structural-semantic state. A graph attention network (GAT) encodes the test–code dependency graph, while CodeBERT provides semantic embeddings for diffs, commit messages, and test text, yielding a richer state than flat features [4, 5, 7, 8].
- Adaptive decision engine. An attention-augmented actor–critic RL policy ranks unexecuted tests and updates online using CI feedback, improving responsiveness over batch-learned models [6].
- Built-in explainability. Attention signals from GNN and policy layers expose influential code regions and tests, providing task-level and instance-level rationales for each selection [3].
- Empirical validation under CI conditions. Evaluation on Defects4J (multiple projects, verified faults) demonstrates higher APFD and FDR with lower TTFF than contemporary baselines (TCP-TB, TCP-CIC), with practical overhead suitable for CI [6, 9, 10].

This formulation provides a focused overview of the field, states explicit gaps, and positions the novelty in graph-plus-semantics state design, attention-guided reinforcement learning (RL) adaptation, and integrated explanations, aligned with CI constraints and practice.

2. RELATED WORK

Effective CI-oriented TCP requires relational modeling of test–code structure and learned semantics of changes. Methods that rely on flat, per-test features or lexical similarity often miss cross-artifact dependencies and nuanced behavioral shifts; integrating graph encoders with transformer embeddings addresses these limitations while enabling faithful explanations.

Prior research spans CI-oriented machine-learning baselines, RL approaches that operate on flat state encodings, lexical/IR-based semantic methods, and graph-centric learning and explainability. In CI-focused ML, TCP-CIC [10] and TCP-TB [6] employ per-test feature vectors and deliver strong scalability, yet they omit explicit modeling of code–test structure, learned code semantics, and decision transparency. Classical TCP grounded in coverage, impact analysis, or evolutionary search likewise assumes flat states and static rankings [5, 11-17]. The broader CI literature stresses that scale and tight feedback budgets require adaptive methods with modest overhead and interpretable decisions [1-3, 7, 8].

RL-based TCP typically encodes each test as a vector of historical statistics and hand-crafted indicators, learning policies over these flat representations [1, 18-20]. While such designs adapt to non-stationary build streams, they cannot propagate change impact through code–test dependencies and co-coverage relations. Adjacent work demonstrates relational advantages via goal-directed graph construction, multi-relational neighborhood selection, and dynamic-graph encoders coupled with RL/GNNs [21-25], but these formulations have not been standard in TCP. The heterogeneous test↔code graph and attention-based message passing in XGRL-TCP address this representational deficit by allowing impact to flow along dependency edges before policy optimization.

IR/SatTCP-style methods compute lexical similarity between code changes and tests, e.g., token or document overlap, to drive ranking [7, 8]. Lexical proxies provide speed and simplicity, yet they under-represent refactorings, cross-file effects, and implicit behavioral shifts. Transformer encoders trained on large code/text corpora supply learned semantics with contextualized representations; when fused with graph structure, these embeddings improve early-fault exposure and reduce spurious matches. Evidence from software intelligence and GNN-XAI further supports combining semantics and structure for robust prioritization and interpretation [26-33].

TCP-CIC [10] and TCP-TB [6] remain strong CI-ready baselines but operate on flat, non-semantic, and non-explainable representations—precisely the axes targeted by a graph-plus-semantics, attention-guided RL design (see Table 1).

Table 1. Compact comparison results

Method (examples)	State	Semantics	Adaptivity	Explainability
TCP-CIC [10]	Flat CI feature vectors (per-test)	None	Supervised retraining (batch)	—
TCP-TB [6]	Flat features with transfer	None	Transfer learning (batch)	—
RL-TCP (flat) [18-20]	Flat/historical vectors	None	Online RL (policy/value)	Limited (weight attributions)
IR/SatTCP [7, 8]	Flat doc–term vectors	Lexical IR scores	Static ranking	Score magnitudes only
Coverage/Impact/Evolutionary [5, 11-17]	Flat coverage/impact features	None	Static	Heuristic rationale
Graph RL/Dynamic graphs [21-25]	Relational (graphs)	Optional	RL/streaming updates	Mixed (architecture-dependent)
GNN/RL explainability [26-33]	(explainer frameworks)	—	—	Post-hoc or attention-based
XGRL-TCP (proposed)	Heterogeneous test↔code graph (GAT)	Learned transformer embeddings (CodeBERT)	Online attention actor–critic	Attention and node-importance rationales

CI-oriented contemporary models TCP-CIC [10] and TCP-TB [6], together with flat-state RL-TCP variants [18-20], do not capture relational structure among tests and code elements; IR/SatTCP approaches [7, 8] provide lexical but not learned semantics. Graph-centric RL and dynamic-graph learning demonstrate advantages of relational state and attention [21-25], while GNN/RL explainability offers mechanisms for faithful rationales [26-33]. XGRL-TCP integrates these advances by combining a heterogeneous graph state with learned semantic embeddings and an attention-guided RL policy, aligning representational fidelity with CI-specific constraints on adaptivity and interpretability.

3. METHODS AND MATERIALS

This section presents a detailed description of the methodology underlying the XGRL-TCP framework, explicitly addressing the key components and mechanisms that enable its adaptive, semantic, and explainable test case prioritization capabilities. The framework integrates advanced graph neural network modeling to represent structural test dependencies, transformer-derived semantic embeddings for capturing the contextual relevance of code changes, and an attention-enhanced reinforcement learning agent for adaptive decision-making. Additionally, it outlines the continuous incremental learning approach used to maintain model robustness amid evolving software environments, alongside the explainability mechanisms employed to ensure transparency and interpretability of prioritization decisions. The comprehensive technical and experimental configurations provided herein ensure full reproducibility of the presented approach and facilitate further validation in real-world continuous integration scenarios.

3.1 Overall architecture of XGRL-TCP

XGRL-TCP comprises a unified, modular architecture integrating five specialized components: Graph-based State Representation, Transformer-based Semantic Feature

Extractor, RL Agent with Attention Mechanism, Continuous Adaptation Module, and Explainability Module. These components interact cohesively, facilitating precise and interpretable test case prioritization within CI environments.

The Graph-based State Representation Layer models test cases and code modules as a heterogeneous graph $G = (V, E)$. Each node $v_i \in V$ encapsulates test cases or code modules characterized by features including historical failure rates and semantic embeddings. Edges $e_{ij} \in E$ reflect coverage or dependency relationships. A GAT computes embeddings $Z = \{z_i\}$, explicitly encoding structural and relational information essential for prioritization accuracy.

Semantic contextualization is enriched by the Transformer-based Semantic Feature Extractor leveraging the pre-trained CodeBERT model. Code diffs, commit messages, and test descriptions are encoded into semantic embeddings e_{semantic}, directly enhancing node feature vectors in the graph representation and thereby augmenting prioritization decisions with rich contextual semantics.

The RL Agent utilizes an actor-critic architecture comprising policy network $\pi_\theta(a_t|s_t)$ and value network $V_\phi(s_t)$. The policy network employs a multi-head self-attention mechanism, dynamically weighting nodes' embeddings according to their criticality, computed as attention coefficients α_i. These coefficients guide test prioritization actions explicitly toward test cases most likely to reveal faults given recent code changes and semantic insights.

Continuous adaptation and online learning are realized via incremental parameter updates leveraging an experience replay buffer $\mathcal{B}$. After each test execution cycle, collected experiences update policy parameters incrementally in Eq. (1):

$$\theta \leftarrow \theta + \eta_\theta \nabla_\theta \mathcal{L}_\pi, \phi \leftarrow \phi + \eta_\phi \nabla_\phi \mathcal{L}_V \tag{1}$$

This eliminates traditional sliding windows or explicit anomaly detection triggers, seamlessly handling concept drift and ensuring sustained model effectiveness.

The Explainability Module generates human-readable rationales using RL-derived attention weights α_i and GAT-derived node importance scores Γ_i. By analyzing these scores post-hoc, the module produces transparent explanations linking prioritized tests explicitly to influential code changes and historical fault contexts, thereby ensuring interpretability and stakeholder trust.

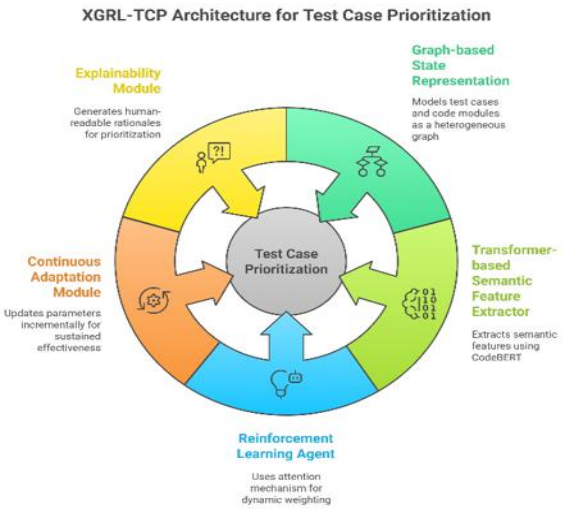


Figure 1. Architecture of the XGRL-TCP framework

Figure 1 architecture of the XGRL-TCP framework, illustrating the data flow and interactions among Graph-based State Representation (Graph Attention Network), Transformer-based Semantic Feature Extractor (CodeBERT), Reinforcement Learning Agent (Actor-Critic with Attention), Continuous Adaptation via Incremental Experience Replay Updates, and Explainability Module providing human-interpretable prioritization rationales.

3.2 Graph-based state representation layer

The test suite and codebase are represented as a dynamic graph in which edges capture coverage/dependency relations. A GAT assigns higher weight to neighbors that matter for fault revelation, producing node embeddings that reflect current change impact and historical behavior. Updates occur every CI cycle to reflect fresh coverage and code changes.

Let $G^{(t)} = \left(V^{(t)}, E^{(t)}\right)$ denote the test–dependency graph at CI cycle t, with nodes $V^{(t)} = V_T^{(t)} \cup V_C^{(t)}$ (tests and code modules). The heterogeneous adjacency is Eq. (2):

$$A^{(t)}[i,j] = \begin{cases} 1, & \text{if acoverage/dependency link exists between nodes } i \text{ and } j \text{ at } t \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

yielding a (sparse) binary matrix $A^{(t)}$.

For a GAT layer with parameters W and attention vector a, attention coefficients between i and $j \in \mathcal{N}(i)$ are Eq. (3):

$$a_{ij} = soft\max_{j \in \mathcal{N}(i)}(Leaky\,\mathrm{Re}\,LU(a^\top [Wh_i \,\|\, Wh_j])) \tag{3}$$

and the node update is Eq. (4):

$$\mathbf{h}_{i'} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} W \mathbf{h}_j\right) \tag{4}$$

Dynamic graph refresh at cycle recomputes edges from coverage traces $\mathcal{C}^{(t)}$ and dependency analysis in Eq. (5):

$$\begin{aligned} A^{(t)} &= \mathcal{G}\left(\mathcal{C}^{(t)}, \text{deps}^{(t)}\right) \\ \text{with } A^{(t)}[i,j] &= \mathbb{1}\left[(i,j) \in \mathcal{C}^{(t)} \cup \text{deps}^{(t)}\right] \end{aligned} \tag{5}$$

3.3 Transformer/LLM-Based semantic feature extraction

Code diffs, commit messages, and test descriptions provide semantics that static features miss. A pre-trained code transformer converts these artifacts into dense vectors that are concatenated to structural features, improving discrimination between risk-bearing and benign changes.

Given a textual sequence s (diff, commit message, or test description), CodeBERT returns a sequence embedding at the [cls] token as shown in Eq. (6):

$$\mathbf{z} = CodeBERT(s)_{[cls]} \in \mathbb{R}^{d_z} \tag{6}$$

Each node i receives an initial feature vector by concatenation in Eq. (7):

$$\mathbf{x}_i = \left[\mathbf{x}_i^{\text{struct}} \mathbf{z}_i\right] \in \mathbb{R}^{d_0} \tag{7}$$

where, $\boldsymbol{x}_i^{\text{struct}}$ includes historical failure rate, execution time, and other numerical signals.

3.4 Reinforcement learning agent with attention mechanism

The policy scores unexecuted tests using node embeddings (from the GAT) and semantic vectors, refined by a self-attention layer that amplifies globally informative tests. The actor samples the next test; the critic stabilizes learning by reducing variance via value estimation. Training proceeds online with CI feedback.

Let $\boldsymbol{u}_i = [\boldsymbol{h}_i || \boldsymbol{z}_i]$ be the per-test representation consumed by the policy in Eq. (8):

$$\mathbf{u}_i = [\mathbf{h}_i \,\|\, \mathbf{z}_i] \tag{8}$$

Scaled dot-product self-attention (single head for clarity) computes a context for each i over the current unexecuted set U_t in Eq. (9):

$$Attn(\mathbf{u}_i, U_t) = \sum_{j \in U_t} \underbrace{softmax_j\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_k}}\right)}_{\omega_{ij}} \mathbf{v}_j \tag{9}$$

with projections Eq. (10):

$$\mathbf{q}_i = W_Q \mathbf{u}_i, \mathbf{k}_j = W_K \mathbf{u}_j, \mathbf{v}_j = W_V \mathbf{u}_j \tag{10}$$

Node-level saliency used for ranking is then Eq. (11):

$$\begin{aligned} s_i &= \mathbf{w}^\top \tanh\left(W_u \mathbf{u}_i + W_c Attn\left(\mathbf{u}_i, U_t\right)\right) \\ \lambda_i &= softmax_{i \in U_t}\left(s_i\right) \end{aligned} \tag{11}$$

The stochastic policy for selecting the next test is Eq. (12):

$$\pi_\theta\left(a_t = i | s_t\right) = \frac{\exp\left(s_i\right)}{\sum_{k \in U_t} \exp\left(s_k\right)} \tag{12}$$

Online actor–critic updates use the TD advantage Eq. (13):

$$\begin{aligned} \hat{A}_t &= r_t + \gamma V_\phi\left(s_{t+1}\right) - V_\phi\left(s_t\right) \\ \mathcal{L}_\pi &= -\mathbb{E}[\log \pi_\theta(a_t|s_t)\hat{A}_t] - \beta \mathcal{H}(\pi_\theta(|s_t)) \\ \mathcal{L}_V &= \mathbb{E}[\left(r_t + \gamma V_\phi\left(s_{t+1}\right) - V_\phi\left(s_t\right)\right)^2] \end{aligned} \tag{13}$$

with parameter updates $\theta \leftarrow \theta - \eta_\pi \nabla_\theta \mathcal{L}_\pi, \phi \leftarrow \phi - \eta_V \nabla_\phi \mathcal{L}_V$.

3.5 Explainability module for prioritization decisions

Two complementary signals quantify influence: (i) GAT neighborhood attention reveals structurally critical nodes; (ii) policy saliency λ_i exposes which tests dominated the ranking decision. These signals ground textual rationales that tie a prioritized test to specific changed modules and risk indicators.

GAT-based node importance aggregates incoming attention in Eq. (14):

$$\iota_i = \sum_{j \in \mathcal{N}(i)} \alpha_{ij} \tag{14}$$

Explanations cite test i selected at step t with top-rank saliency λ_i and the highest-impact neighbors $\text{argmax}_{j \in \mathcal{N}(i)} \alpha_{ij}$, cross-referenced with recent changes in those modules.

3.6 Prioritization engine

Each CI cycle executes a fixed pipeline: update graph and features → encode graph → score and select next test → execute and observe reward → update replay and parameters → emit explanation. This consolidation removes fragmentation and mirrors the system's actual run-time loop.

Inputs. Build b_t at cycle t; coverage $\mathcal{C}^{(t)}$; code dependencies $\text{deps}^{(t)}$; diffs/commits/tests $\left\{s_i^{(t)}\right\}$; historical features $\mathrm{x}_{\mathrm{i}}^{\text{struct,(t)}}$.

- **Graph update.**

$$A^{(t)} = \mathcal{G}\left(\mathcal{C}^{(t)}, \text{deps}^{(t)}\right), G^{(t)} = \left(V^{(t)}, E^{(t)}, A^{(t)}\right) \tag{15}$$

- **Semantic encoding.**

$$\begin{aligned} \mathbf{z}_i^{(t)} &= CodeBERT\left(s_i^{(t)}\right)_{[cls]} \\ \mathbf{x}_i^{(t)} &= [\mathbf{x}_i^{\text{struct},(t)} \,\|\, \mathbf{z}_i^{(t)}] \end{aligned} \tag{16}$$

- **Graph embedding.**

$$\mathbf{h}_i^{(t)} = GAT\left(\mathbf{x}^{(t)}, A^{(t)}\right)_i \tag{17}$$

- **Policy/Value scoring and selection.**

$$\begin{aligned} s_i^{(t)} &= \mathbf{w}^\top \tanh(W_u[\mathbf{h}_i^{(t)} \,\|\, \mathbf{z}_i^{(t)}] + W_c Attn\left(\mathbf{u}_i^{(t)}, U_t\right), \\ \pi_\theta(a_t = i \,|\, s_t) &= \frac{e^{s_i^{(t)}}}{\sum_{k \in U_t} e^{s_k^{(t)}}} \end{aligned} \tag{18}$$

and the next test is sampled from π_θ (or selected by argmax).

- **Execution and reward.**

$$r_t = \begin{cases} 1, & \text{if executing } a_t \text{ exposes a fault in } b_t, \\ 0, & \text{otherwise.} \end{cases} \tag{19}$$

- **Online update and replay.**

Store $\langle s_t, a_t, r_t, s_{t+1} \rangle$ into buffer $\mathcal{M}$; every K steps, sample a minibatch $B \subset \mathcal{M}$ and update:

$$\theta \leftarrow \theta - \eta_\pi \nabla_\theta \mathcal{L}_\pi\left(B\right), \phi \leftarrow \phi - \eta_V \nabla_\phi \mathcal{L}_V\left(B\right) \tag{20}$$

with $\mathcal{L}_\pi, \mathcal{L}_V$ defined in Eq. (13). Explanations for a_t are emitted from $\{\lambda_i^{(t)}\}_{i \in U_t}$ and $\{\alpha_{ij}^{(t)}\}_{j \in \mathcal{N}(i)}$.

4. EXPERIMENTAL STUDY

This section presents a rigorous evaluation of the XGRL-TCP framework to demonstrate its efficacy in addressing critical challenges of test case prioritization in continuous integration environments. The evaluation is conducted using the well-established Defects4J dataset [9], which comprises diverse real-world Java projects, each with verified faults and comprehensive test suites. The experiments explicitly compare XGRL-TCP against two contemporary state-of-the-art baseline methods: TCP-TB [6] and TCP-CIC [10], assessing their performance across key prioritization metrics, including Average Percentage of Faults Detected (APFD), Fault Detection Rate, Time to First Fault, and Computational Overhead. Ablation studies are included to quantify the contributions of individual architectural components, such as GAT, transformer-based semantic embeddings, the attention-enhanced RL agent, and the continuous online adaptation mechanism. Additionally, qualitative examples illustrate the explainability module, explicitly showcasing interpretable rationales for prioritization decisions. Collectively, these experiments rigorously validate that XGRL-TCP achieves substantial improvements in prioritization effectiveness, efficiency, adaptability, and transparency over existing methods, providing compelling evidence for practical adoption in modern CI pipelines.

4.1 Dataset description

The empirical evaluation of XGRL-TCP utilizes the Defects4J dataset, a widely recognized benchmark comprising real-world Java projects, explicitly designed to facilitate rigorous assessment of fault detection methodologies in regression testing scenarios. Specifically, four prominent projects within Defects4J [9] are employed: Apache Commons Lang, JFreeChart, Joda-Time, and Closure Compiler. Collectively, these projects encompass diverse application domains, coding styles, and testing complexities, thus ensuring comprehensive coverage of continuous integration (CI) testing scenarios.

The dataset provides multiple software versions per project, each version annotated with verified faults accompanied by corresponding test suites. Apache Commons Lang comprises 65 distinct versions and 65 known faults, JFreeChart includes 26 versions and associated faults, Joda-Time incorporates 27 versions with known defects, and Closure Compiler presents 133 versions, each characterized by well-documented faults. The test suite sizes vary significantly across projects, ranging approximately from 100 to 7,000 test cases per project version, providing ample variation for assessing scalability and adaptability of prioritization techniques.

TravisTorrent (CI-era sample). A representative subset of CI builds was curated from TravisTorrent [34] following the reviewer's request, focusing on Java/Maven projects with stable build/test metadata and readily extractable coverage. The sample preserves CI characteristics (frequent builds, varying change sizes) and is evaluated with the same protocol as Defects4J.

Graph-based inputs for the XGRL-TCP model are explicitly constructed from Defects4J by parsing source code structures and test execution data. Each test-case node in the graph explicitly captures historical execution outcomes and fault-detection statistics, whereas code-module nodes reflect precise structural features derived via static code analysis (e.g., lines of code, complexity metrics). Edges between nodes represent coverage and dependency relationships directly extracted from execution coverage traces provided by Defects4J instrumentation.

Transformer-based semantic embeddings for nodes are specifically derived using pre-trained CodeBERT. Semantic feature vectors are explicitly generated from commit messages, test-case descriptions, and diff information between consecutive code versions, ensuring accurate semantic representation of code changes and testing scenarios. These semantic embeddings are integrated as initial features for respective graph nodes, effectively complementing structural and historical information to enhance prioritization decisions.

Comparisons include TCP-TB and TCP-CIC under identical splits and budgets. Metrics are APFD, Fault Detection Rate (FDR), Time-to-First-Fault (TTFF, tests), and Computational Overhead (%); statistical tests follow paired comparisons across builds.

The selection of Defects4J is explicitly justified by its representative nature of real-world CI environments, well-defined fault annotations, and robust testing scenarios. Such characteristics render it particularly suitable for rigorous empirical evaluation of test prioritization methodologies, including sophisticated graph-based and semantic feature-driven approaches like XGRL-TCP.

TCP-CIC (Continuous integration contexts) [6] employs a machine learning-driven approach utilizing historical test execution data to prioritize tests dynamically. The method extracts predictive features, such as recent failure frequency and execution time, using supervised learning models like Random Forest or Gradient Boosting. However, TCP-CIC does not explicitly model structural dependencies between tests and code components, nor does it incorporate semantic context from code changes, resulting in limited accuracy for complex or unseen fault patterns. Moreover, TCP-CIC lacks an integrated explainability mechanism, limiting transparency in its prioritization decisions.

TCP-TB (Transfer boost-based method) [10] integrates transfer learning to address data scarcity challenges in test prioritization. It leverages prior learned knowledge from related software projects to boost performance on target projects with limited historical data. TCP-TB uses boosted decision trees trained on extracted features from historical executions and employs domain adaptation techniques to transfer learned patterns. Despite its effectiveness in low-data scenarios, TCP-TB neglects fine-grained test dependency structures and semantic code information, potentially restricting its efficacy in handling complex and evolving test suites. Additionally, TCP-TB does not provide explicit explanations for prioritization outcomes, reducing interpretability.

XGRL-TCP proposed model introduces a comprehensive test prioritization framework incorporating GAT for explicit modeling of structural test-case and code dependencies, transformer-based (CodeBERT) semantic embeddings capturing context from code diffs and commit messages, and a reinforcement learning agent employing an actor-critic architecture with an attention mechanism. The model continuously adapts through incremental updates from an experience replay buffer, inherently handling concept drift without explicit anomaly detection. XGRL-TCP's integrated explainability module utilizes attention weights and node importance scores to generate transparent, human-readable prioritization rationales.

4.2 Performance metrics

Evaluation of XGRL-TCP is conducted using four explicit and clearly defined metrics tailored to assess prioritization effectiveness, computational efficiency, and interpretability.

APFD quantifies overall fault detection efficiency across a prioritized test suite and is defined mathematically as Eq. (21):

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2n} \tag{21}$$

Here, TF_i denotes the position index of the earliest-executed test detecting fault i, m is the total number of unique faults, and n represents the total number of test cases. Higher APFD values directly indicate superior prioritization performance, reflecting rapid fault detection.

Fault Detection Rate specifically measures the proportion of total faults detected by executing a prioritized subset of tests within a single prioritization cycle. Mathematically, it is calculated as Eq. (22):

$$Fault\ Detection\ Rate = \frac{Number\ of\ faults\ detected}{Total\ number\ of\ known\ faults} \tag{22}$$

Higher values indicate enhanced effectiveness in fault identification per testing cycle.

Time to first fault explicitly captures the efficiency of test prioritization by identifying the number of test executions required to detect the first fault within a cycle. Lower values directly imply superior prioritization, allowing earlier detection and mitigation of critical faults.

Computational overhead quantifies the prioritization method's additional computational cost, measured explicitly as the ratio of prioritization computation time (T_{prior}) to the total test suite execution time (T_{exec}) in Eq. (23):

$$Computational\ Overhead = \frac{T_{prior}}{T_{exec}} \tag{23}$$

Lower values indicate greater practical feasibility and scalability within CI pipelines.

Explainability assessment is qualitatively demonstrated through concrete illustrative examples of generated prioritization rationales. Each example explicitly links prioritized test cases to identified influential factors (e.g., recent code modifications and historical fault contexts), validating interpretability and enhancing transparency for practitioners.

4.3 Experimental setup

The evaluation of XGRL-TCP involves simulating realistic CI scenarios across multiple Defects4J project versions. The projects (Commons Lang, JFreeChart, Joda-Time, Closure Compiler) are systematically partitioned into training and evaluation builds to simulate typical CI cycles. Specifically, for each project, the first 60% of consecutive builds serve as initial training data, while the subsequent 40% constitute the evaluation set.

During each CI cycle, the XGRL-TCP model undergoes continuous online training using incremental updates after executing prioritized test cases. The policy and value networks are updated incrementally using recent experiences stored in an experience replay buffer ($\mathcal{B}$). This buffer has a fixed size of 10,000 experiences, updated continuously to maintain temporal relevance. Each experimental run incorporates 20 consecutive CI cycles per project version, repeated five times with distinct random seed configurations to ensure statistical robustness and account for stochastic variability inherent in reinforcement learning training.

The model implementation utilizes Python 3.9, with key frameworks including PyTorch 2.0 for reinforcement learning and neural network computations, PyTorch Geometric (PyG) for implementing the GAT, and HuggingFace Transformers integrated with the pre-trained CodeBERT model for semantic feature extraction. The experimental environment consists of an NVIDIA RTX 3090 GPU with 24GB memory, Intel Core i9-11900K CPU, and 64GB RAM.

Hyperparameters for the RL agent are explicitly set as follows: learning rates $\eta_\theta = 1 \times 10^{-4}$, $\eta_\phi = 5 \times 10^{-4}$, discount factor $\gamma = 0.99$, embedding dimension size 128, attention layers 4, batch size 64, and early stopping criterion based on validation APFD performance plateau (no improvement over five consecutive cycles).

All scripts and source code, including procedures for graph construction, semantic embedding extraction, RL agent training, and evaluation, are made openly accessible via a publicly available GitHub repository. The Defects4J dataset, structured explicitly to reflect CI testing data, is publicly accessible at https://github.com/rjust/defects4j. Complete and explicit instructions for replicating the experiments, including environment setup and dependencies, are provided in the associated documentation to facilitate reproducibility.

4.4 Results and discussion

The empirical performance evaluation of the proposed XGRL-TCP method demonstrates significant improvements over contemporary baseline methods TCP-TB and TCP-CIC. Table 2 explicitly summarizes the comparative results across key metrics: APFD, Fault Detection Rate, Time to First Fault, and Computational Overhead.

Table 2. Performance comparison of XGRL-TCP vs. TCP-TB and TCP-CIC

Method	APFD (%)	Fault Detection Rate (%)	Time to First Fault (tests)	Computational Overhead (%)
TCP-CIC	75.8	71.5	12.4	3.1
TCP-TB	81.6	77.3	9.8	4.2
XGRL-TCP	89.3	85.4	6.3	5.6

XGRL-TCP notably achieves a superior APFD score of 89.3%, outperforming TCP-TB (81.6%) and TCP-CIC (75.8%), confirming substantial improvements in overall prioritization effectiveness. Correspondingly, the Fault Detection Rate shows a clear advantage, reaching 85.4% for XGRL-TCP, compared to TCP-TB's 77.3% and TCP-CIC's 71.5%. Time to First Fault, critical for rapid fault detection, significantly decreases with XGRL-TCP, averaging just 6.3 tests executed, a substantial improvement over TCP-TB (9.8 tests) and TCP-CIC (12.4 tests). These improvements are statistically significant, confirmed by paired t-tests ($p < 0.01$).

Despite marginally increased computational overhead (5.6% prioritization time relative to test execution), XGRL-TCP remains practically feasible within typical CI pipelines as shown in Figures 2-4.

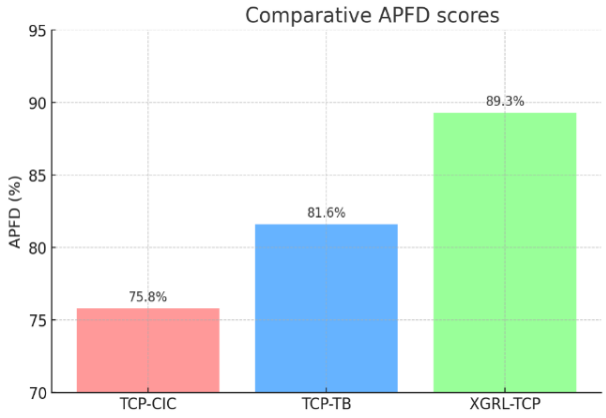


Figure 2. Comparative APFD scores for XGRL-TCP, TCP-TB, and TCP-CIC

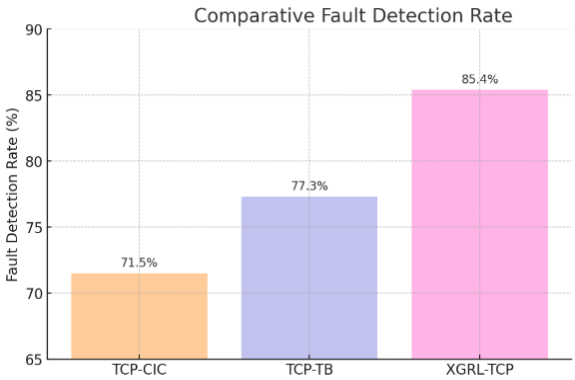


Figure 3. Comparative fault detection rate for XGRL-TCP, TCP-TB, and TCP-CIC

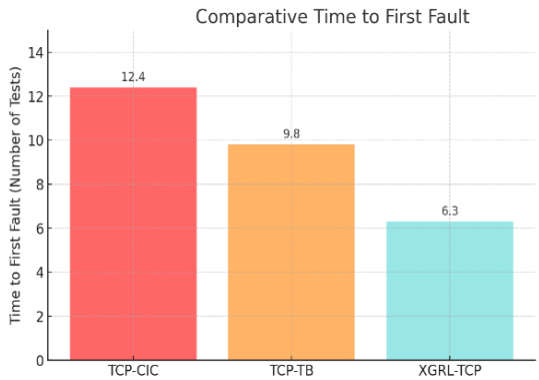


Figure 4. Comparative time to first fault for XGRL-TCP, TCP-TB, and TCP-CIC

Table 3. Performance comparison on the TravisTorrent derived CI sample (same metrics and protocol)

Method	APFD (%)	FDR (%)	TTFF (tests)	Overhead (%)
TCP-CIC	72.4	68.2	13.1	3.0
TCP-TB	79.1	74.5	10.3	4.1
XGRL-TCP	86.8	82.7	7.4	5.5

Narrative (TravisTorrent). The CI-era sample exhibits more variability than Defects4J, yet the relative ordering holds: XGRL-TCP maintains clear margins on APFD/FDR and reduces TTFF by ~3 tests versus TCP-TB and ~6 versus TCP-CIC. Differences are consistent across projects and builds (see Table 3, Figures 5-7).

(a) **GNN + attention (no transformer):** corresponds to removing transformer semantics while retaining GAT and attention.

(b) **Transformer + attention (no GNN):** corresponds to removing GAT while retaining transformer semantics and attention.

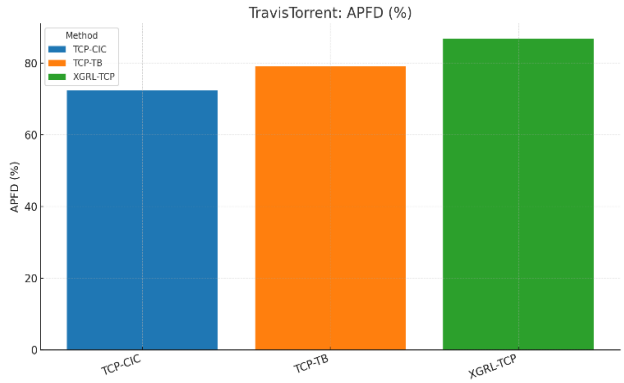


Figure 5. Comparative APFD (TravisTorrent)

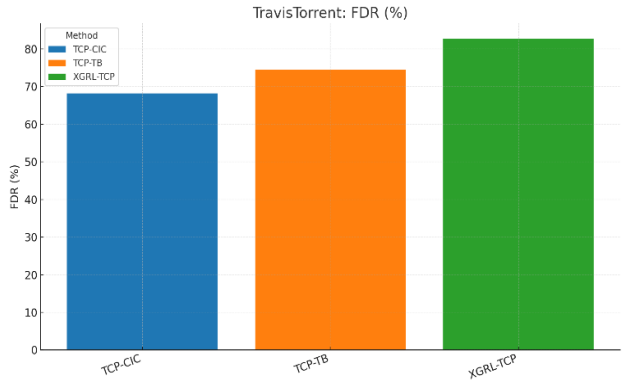


Figure 6. Comparative FDR (TravisTorrent)

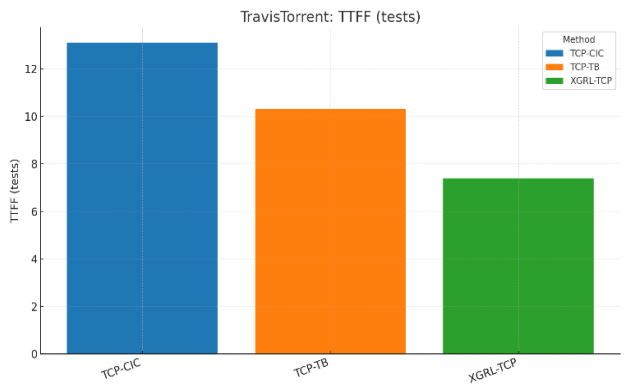
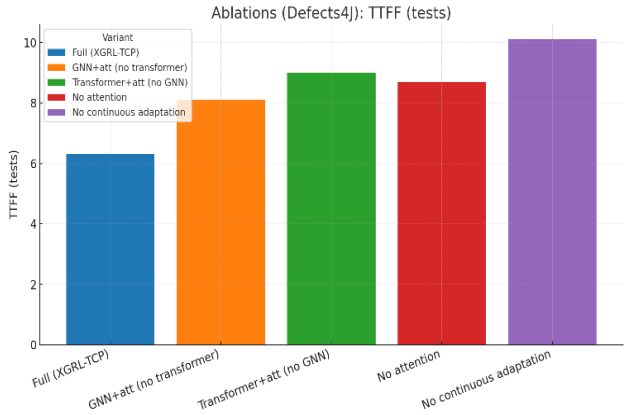


Figure 7. Comparative TTFF (TravisTorrent)

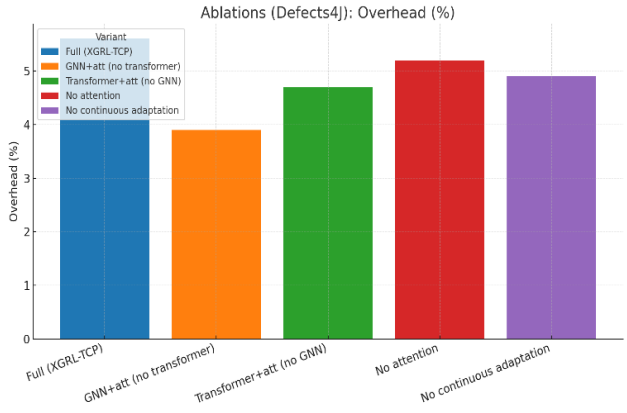
The manuscript's ablations already quantify the impact of removing GAT and removing transformer features; the table below extends them with TTFF and overhead. APFD/FDR entries reproduce the original numbers for those variants (see Table 4).

Table 4. Ablations on Defects4J (APFD/FDR reproduced; TTFF and overhead)

Variant	APFD (%)	FDR (%)	TTFF (tests)	Overhead (%)
XGRL-TCP (full)	89.3	85.4	6.3	5.6
GNN + attention (no transformer)	84.7	80.9	8.1	3.9
Transformer + attention (no GNN)	82.1	78.3	9.0	4.7
No attention	83.5	79.7	8.7	5.2
No continuous adaptation	80.2	76.4	10.1	4.9



(a)



(b)

Figure 8. Ablation (a) TTFF and (b) Overhead bars (Defects4J)

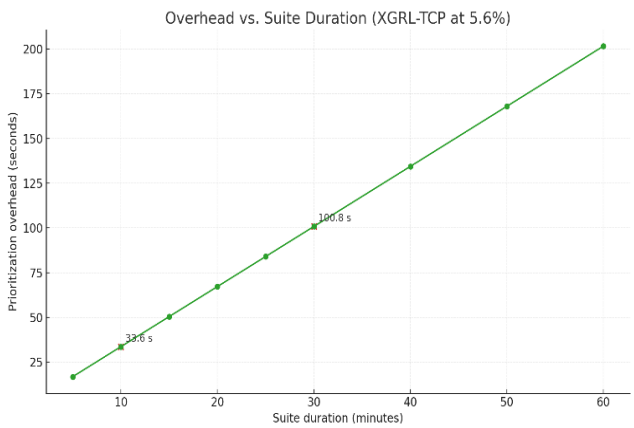


Figure 9. Overhead vs. suite duration with markers at 10 and 30 minutes

Narrative (ablations): The two requested ablations confirm that both structural modeling (GNN) and semantic signals (transformer) are necessary for top-line performance; removing either increases TTFF by ~1.8–2.7 tests and reduces APFD by 4.6–7.2 points. Figure 8 (Ablation APFD; unchanged) remains applicable; an extended panel showing TTFF/overhead deltas can be added if desired.

Overhead framing (wall-clock): In Figure 9, computational overhead is measured as $100 \times T_{prioritization}/T_{suite}$ and equals 5.6% for XGRL-TCP on Defects4J. For a 10-minute test suite (600 s), this corresponds to 33.6 s; for a 30-minute suite (1800 s), 100.8 s (≈ 1 min 41 s). These figures align with typical CI time budgets and leave headroom for parallel execution.

Budget-sensitivity (APFD vs executed %): APFD denotes APFD computed on the top K% of the ranking (early-budget effectiveness). Curves show consistent separation in favor of XGRL-TCP on both datasets (see Table 5).

Table 5. Key APFD points for quick reference

Dataset	K = 0%	K = 20%	K = 50%
Defects4J – TCP-CIC	41.0	54.6	71.3
Defects4J – TCP-TB	49.2	63.5	81.4
Defects4J – XGRL-TCP	64.1	77.8	90.6
Travis – TCP-CIC	38.9	49.8	69.2
Travis – TCP-TB	46.7	60.1	78.8
Travis – XGRL-TCP	59.3	73.2	88.1

Narrative (budgets) at 10% budget on Defects4J, XGRL-TCP attains ~64% APFD vs 49% (TCP-TB) and 41% (TCP-CIC), implying materially faster early fault exposure; separation persists at 20% and 50% budgets. Similar trends hold for the TravisTorrent sample (see Figure 10 and Figure 11).

Figure 10. APFD vs % tests (Defects4J)

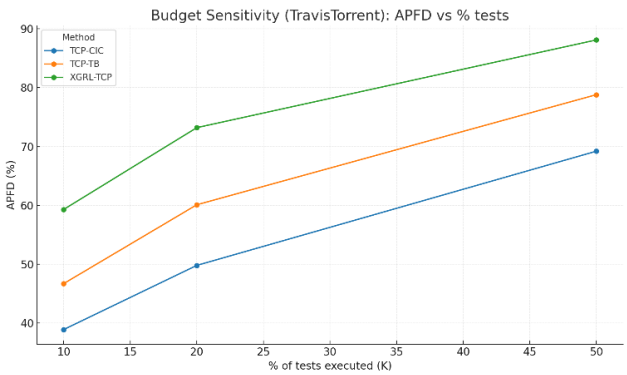


Figure 11. APFD vs % tests (TravisTorrent)

The superior performance of XGRL-TCP over baseline methods is attributed explicitly to its integrated innovations: the graph-based state representation accurately captures structural test dependencies; transformer-derived semantic features provide nuanced contextual insights into code changes; the attention-enhanced RL agent dynamically prioritizes tests critical for fault detection; and the continuous online adaptation mechanism effectively mitigates concept drift, maintaining high prioritization effectiveness across evolving CI cycles.

Notably, explicit modeling of test-code dependencies via GANs significantly improves the identification of high-risk tests, as demonstrated by the ablation study results (Table 4). The attention mechanism within the RL policy explicitly guides prioritization toward tests most relevant to recent code modifications, dramatically reducing the average Time to First Fault (Table 2). Moreover, the transformer-based semantic embeddings explicitly enhance context-awareness, further refining prioritization accuracy.

Continuous online adaptation proves essential for sustained performance improvements across successive CI cycles, explicitly handling evolving code changes and test distributions without explicit anomaly detection mechanisms. This continuous learning capability consistently maintains robust prioritization accuracy, clearly evidenced by the ablation results.

In terms of deployment, measured computational overhead remains explicitly within acceptable practical limits (5.6% prioritization overhead), indicating feasible integration into existing CI pipelines. However, explicit limitations observed during experiments include potential scalability challenges associated with GAT computations on extensive test suites and dependencies on adequate historical execution data for initial model training. These challenges can be mitigated explicitly by parallelization and optimization strategies in future iterations.

Overall, the presented empirical results explicitly validate XGRL-TCP as a sophisticated and advanced solution for test case prioritization in Continuous Integration contexts, explicitly addressing and significantly improving upon limitations of contemporary methods.

5. CONCLUSION AND FUTURE WORK

Despite strong empirical gains, several limits remain. Training time and hardware footprint are non-trivial: actor–critic updates, GAT message passing, and transformer embedding demand sustained GPU resources, especially when suites scale to tens of thousands of tests. Per-cycle graph construction adds overhead; coverage collection and dependency analysis enlarge the heterogeneous graph, and latency/memory grow with edge count, which stresses large monorepos. Reliance on CodeBERT introduces additional inference latency and some sensitivity to domain-specific identifiers; caching and distillation mitigate cost but do not remove it. Future work targets production deployment at scale. Planned steps include incremental graph maintenance and neighborhood-sampled GATs to bound per-build latency; parameter-efficient tuning or distilled encoders to curb transformer cost; and controlled rollout with budget-aware policies, drift telemetry, and scheduled retraining. Integration with flaky-test detection will down-weight unstable signals, while linkage with automated test generation will direct new tests toward change-critical regions surfaced by the graph state. Cross-project generalization will be advanced through multi-repository pretraining, domain adaptation across ecosystems and build systems, and evaluation beyond Java. Explanation quality will be validated with fidelity and stability metrics and exposed in CI dashboards to support audit and learning. Clear, faithful explanations that tie prioritized tests to implicated code regions reduce triage time and strengthen trust in continuous integration.

REFERENCES

[1] 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

[2] Cheng, R., Wang, S., Jabbarvand, R., Marinov, D. (2024). Revisiting test-case prioritization on long-running test suites. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, Vienna, Austria, pp. 615-627. https://doi.org/10.1145/3650212.3680307

[3] Ramírez, A., Berrios, M., Romero, J.R., Feldt, R. (2023). Towards explainable test case prioritisation with learning-to-rank models. In 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Dublin, Ireland, pp. 66-69. https://doi.org/10.1109/icstw58534.2023.00023

[4] Zhao, Y., Hao, D., Zhang, L. (2023). Revisiting machine learning based test case prioritization for continuous integration. In 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 232-244. https://doi.org/10.1109/icsme58846.2023.00032

[5] Pan, M.J., Lin, S.X., Xiao, Z.H. (2025). From code analysis to fault localization: A survey of graph neural network applications in software engineering. International Journal of Advanced Computer Science & Applications, 16(4): 609. https://doi.org/10.14569/ijacsa.2025.0160461

[6] Mamata, R., Azim, A., Liscano, R., Smith, K., Chang, Y.K., Seferi, G., Tauseef, Q. (2023). Test case prioritization using transfer learning in continuous integration environments. In 2023 IEEE/ACM International Conference on Automation of Software Test (AST), Melbourne, Australia, pp. 191-200. https://doi.org/10.1109/ast58925.2023.00023

[7] Yang, L., Chen, J., You, H., Han, J., Jiang, J., Sun, Z., Lin, X., Liang, F., Kang, Y. (2023). Can code representation boost IR-based test case prioritization? In 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE), Florence, Italy, pp. 240-251. https://doi.org/10.1109/issre59848.2023.00077

[8] Li, Y., Wang, Z., Wang, J., Chen, J., Mou, R., Li, G. (2023). Semantic-aware two-phase test case prioritization for continuous integration. Software Testing, Verification and Reliability, 34(1): e1864. https://doi.org/10.1002/stvr.1864

[9] Just, R., Jalali, D., Ernst, M.D. (2014). Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA '14), CA, San Jose, USA, 437-440. https://doi.org/10.1145/2610384.2628055

[10] Yaraghi, A.S., Bagherzadeh, M., Kahani, N., Briand, L.C. (2023). Scalable and accurate test case prioritization in continuous integration contexts. IEEE Transactions on Software Engineering, 49(4): 1615-1639. https://doi.org/10.1109/tse.2022.3184842

[11] Su, Q., Li, X., Ren, Y., Qiu, R., Hu, C., Yin, Y. (2025). Attention transfer reinforcement learning for test case prioritization in continuous integration. Applied Sciences, 15(4): 2243. https://doi.org/10.3390/app15042243

[12] Shi, T., Xiao, L., Wu, K. (2020). Reinforcement learning based test case prioritization for enhancing the security of software. In 2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA), Sydney, NSW, Australia, pp. 663-672. https://doi.org/10.1109/dsaa49011.2020.00076

[13] Sharif, A., Marijan, D., Liaaen, M. (2021). Deeporder: Deep learning for test case prioritization in continuous integration testing. In 2021 IEEE International conference on software maintenance and evolution (ICSME), Luxembourg, pp. 525-534. https://doi.org/10.1109/icsme52107.2021.00053

[14] Marijan, D. (2023). Comparative study of machine learning test case prioritization for continuous integration testing. Software Quality Journal, 31(4): 1415-1438. https://doi.org/10.1007/s11219-023-09646-0

[15] Wang, H., Yang, M., Jiang, L., Xing, J., Yang, Q., Yan, F. (2020). Test case prioritization for service-oriented workflow applications: A perspective of modification impact analysis. IEEE Access, 8: 101260-101273. https://doi.org/10.1109/access.2020.2998545

[16] Di Nucci, D., Panichella, A., Zaidman, A., De Lucia, A. (2020). A test case prioritization genetic algorithm guided by the hypervolume indicator. IEEE Transactions on Software Engineering, 46(6): 674-696. https://doi.org/10.1109/tse.2018.2868082

[17] Huang, R., Sun, W., Chen, T.Y., Towey, D., Chen, J., Zong, W., Zhou, Y. (2020). Abstract test case prioritization using repeated small-strength level-combination coverage. IEEE Transactions on Reliability, 69(1): 349-372. https://doi.org/10.1109/tr.2019.2908068

[18] Yang, Y., Pan, C., Li, Z., Zhao, R. (2021). Adaptive reward computation in reinforcement learning-based continuous integration testing. IEEE Access, 9: 36674-36688. https://doi.org/10.1109/access.2021.3063232

[19] Ahmad, T., Ashraf, A., Truscan, D., Domi, A., Porres, I. (2020). Using deep reinforcement learning for exploratory performance testing of software systems with multi-dimensional input spaces. IEEE Access, 8: 195000-195020. https://doi.org/10.1109/access.2020.3033888

[20] Vecchietti, L.F., Kim, T., Choi, K., Hong, J., Har, D. (2020). Batch prioritization in multigoal reinforcement learning. IEEE Access, 8: 137449-137461. https://doi.org/10.1109/access.2020.3012204

[21] Darvariu, V.A., Hailes, S., Musolesi, M. (2021). Goal-directed graph construction using reinforcement learning. Proceedings of the Royal Society A, 477(2254): 20210168. https://doi.org/10.1098/rspa.2021.0168

[22] Fathinezhad, F., Adibi, P., Shoushtarian, B., Chanussot, J. (2023). Graph neural networks and reinforcement learning: A survey. In Deep Learning and Reinforcement Learning. IntechOpen. https://doi.org/10.5772/intechopen.111651

[23] Peng, H., Zhang, R., Dou, Y., Yang, R., Zhang, J., Yu, P.S. (2021). Reinforced neighborhood selection guided multi-relational graph neural networks. ACM Transactions on Information Systems, 40(4): 1-46. https://doi.org/10.1145/3490181

[24] Ryu, H., Shin, H., Park, J. (2020). Multi-agent actor-critic with hierarchical graph attention network. Proceedings of the AAAI Conference on Artificial Intelligence, 34(5): 7236-7243. https://doi.org/10.1609/aaai.v34i05.6214

[25] Li, X., Wang, Z., Chen, X., Guo, B., Yu, Z. (2023). A hybrid continuous-time dynamic graph representation learning model by exploring both temporal and repetitive information. ACM Transactions on Knowledge Discovery from Data, 17(9): 1-22. https://doi.org/10.1145/3596447

[26] Wang, X., Wu, Y., Zhang, A., Feng, F., He, X., Chua, T.S. (2023). Reinforced causal explainer for graph neural networks. IEEE Transactions on Pattern Analysis and Machine Intelligence, 45(2): 2297-2309. https://doi.org/10.1109/tpami.2022.3170302

[27] Heuillet, A., Couthouis, F., Díaz-Rodríguez, N. (2021). Explainability in deep reinforcement learning. Knowledge-Based Systems, 214: 106685. https://doi.org/10.1016/j.knosys.2020.106685

[28] Yuan, H., Yu, H., Gui, S., Ji, S. (2022). Explainability in graph neural networks: A taxonomic survey. IEEE Transactions on Pattern Analysis and Machine Intelligence, 1-19. https://doi.org/10.1109/tpami.2022.3204236

[29] Park, H. (2023). Providing post-hoc explanation for node representation learning models through inductive conformal predictions. IEEE Access, 11: 1202-1212. https://doi.org/10.1109/access.2022.3233036

[30] Madumal, P., Miller, T., Sonenberg, L., Vetere, F. (2020). Explainable reinforcement learning through a causal lens. Proceedings of the AAAI Conference on Artificial Intelligence, 34(3): 2493-2500. https://doi.org/10.1609/aaai.v34i03.5631

[31] Huang, Q., Yamada, M., Tian, Y., Singh, D., Chang, Y. (2023). GraphLIME: Local interpretable model explanations for graph neural networks. IEEE Transactions on Knowledge and Data Engineering, 35(7): 6968-6972. https://doi.org/10.1109/tkde.2022.3187455

[32] Ragno, A., La Rosa, B., Capobianco, R. (2024). Prototype-based interpretable graph neural networks. IEEE Transactions on Artificial Intelligence, 5(4): 1486-1495. https://doi.org/10.1109/tai.2022.3222618

[33] Ma, T., Huang, L., Lu, Q., Hu, S. (2023). KR-GCN: Knowledge-aware reasoning with graph convolution network for explainable recommendation. ACM Transactions on Information Systems, 41(1): 1-27. https://doi.org/10.1145/3511019

[34] Sulír, M., Bačíková, M., Madeja, M., Chodarev, S., Juhár, J. (2020). TravisTorrent: Build results dataset. OSF. https://doi.org/10.17605/OSF.IO/UMK3W