Journal of Advances in Information Technology, Vol. 17, No. 3, 2026

DeepARTS: A Graph-Transformer Continual Learning Framework for Adaptive Regression Test Sequence Recommendation

Srinivasa Rao Kongarana*, Ananda Rao Akepogu, and Radhika Raju Palagiri

Department of Computer Science & Engineering, College of Engineering,
Jawaharlal Nehru Technological University Anantapur, Ananthapur, India
Email: srinivas.cst4@gmail.com (S.R.K.); akepogu@gmail.com (A.R.A.); radhikaraju.p@gmail.com (R.R.P.)
*Corresponding author

***Abstract*—Continuous Integration (CI) pipelines require fast regression feedback under strict time budgets, making full regression suites impractical for large systems. Deep Adaptive Regression Test Sequencer (DeepARTS) generates an ordered regression test sequence per CI cycle under a budget *K*. The approach combines a Graph Attention Network (GAT) for code–test dependency encoding, a Transformer encoder for context-aware modeling of candidate tests, and a deep Q-network for sequential test selection using a detection-oriented reward with an execution mask. Continual learning based on entropy-guided replay and Maximum Mean Discrepancy (MMD) drift detection updates model parameters across cycles without full retraining. Evaluation uses Regression Test Prioritization Torrent (RTPTorrent), a dataset containing 20 open-source Java projects and more than 100,000 TravisCI build logs, with chronological per-project splits to reduce temporal leakage. Under identical candidate sets and budget *K*, DeepARTS improves early fault detection compared with Linkage Learning-based Non-Dominated Sorting Genetic Algorithm (L2-NSGA), achieving Average Percentage of Faults Detected (APFD) = 0.94 with Precision@*K* = 99.2% and Recall@*K* = 98.7%, while maintaining per-cycle recommendation latency near 2.5 s on Central Processing Unit (CPU). Generalization beyond Java projects and TravisCI logs remains to be validated.**

***Keywords*—adaptive regression test sequence, optimum regression testing, machine learning, adaptive test sequence recommendation system**

I. Introduction

Continuous Integration and Deployment (CI/CD) pipelines require fast and accurate testing. Frequent commits reduce the time available for executing long regression suites; in large systems, full regression testing can take days. Reordering tests so that fault-revealing tests execute early—Test Case Prioritization (TCP)—improves feedback speed and resource use. Traditional strategies such as retest-all and simple heuristics struggle with fast-changing code bases and often need repeated recomputation, which adds cost and reduces practical value in CI settings [1].

Early machine-learning approaches ranked tests using historical outcomes. DeepOrder applied deep neural networks to learn rankings from prior failure rates and durations [2]. Such models improved over static heuristics but behaved as fixed regressors trained on past snapshots; accuracy dropped when projects evolved, forcing retraining. Reinforcement Learning (RL) introduced adaptive updating of priorities: Reinforced Test Case Selection (RETECS) re-ranked tests using execution history and failure logs and learned from detection rewards over time [3, 4]. Attention and sequence methods further improved ordering quality by modeling context among test cases [5].

Significant gaps remain. Most solutions do not perform continual learning; models stay static until full retraining, leading to performance decay as projects evolve. Many methods treat tests as independent items and ignore structural relations between tests and code elements (e.g., calls, shared coverage, and dependency paths). Interactions among tests are also under-modeled; running one test can change the value of running another, especially under tight time budgets.

Deep Adaptive Regression Test Sequencer (DeepARTS addresses these gaps with a unified architecture that combines (1) a graph neural network to encode code–test structure, (2) a Transformer-based encoder to capture dependencies along the execution order, (3) a deep Q-network that prioritizes tests for early fault discovery under a detection-focused and cost-aware reward with an execution mask, and (4) a continual-learning loop that updates the model across continuous-integration cycles without full retraining. This design aims at accurate early fault detection, stable performance across evolving versions, and low per-cycle latency that is suitable for continuous-integration pipelines. Industrial CI/CD environments often contain flaky and noisy tests; explicit

Manuscript received September 9, 2025; revised October 30, 2025; accepted December 17, 2025; published March 26, 2026.

modeling of flakiness is not included in the current design and is treated as a limitation and direction for future work.

Empirical evaluation uses the Regression Test Prioritization Torrent (RTPTorrent) dataset, which is designed for assessing regression test prioritization and contains 20 open-source Java projects with more than 100{,}000 TravisCI build logs that span diverse sizes, contributor counts, and maturity levels. Results are reported using ranking-oriented measures with an emphasis on early fault detection efficiency and runtime overhead, and comparisons include contemporary baselines under identical candidate sets and continuous-integration budgets. The evidence therefore describes behaviour for Java projects on TravisCI, and extension to other programming languages and build systems remains an open question.

Section II reviews related work in test case prioritization, graph-based analysis, attention and Transformer-based modeling, reinforcement learning, and continual learning. Section III presents the DeepARTS methodology, Section IV presents the experimental study and the discussion of findings and practical implications for continuous-integration environments, and Section V presents the conclusion and outlines future work.

II. Related Work

Machine learning for TCP: Machine learning supports TCP by learning from past executions. Survey studies report wide use of coverage, failure history, and engineered features for ranking and selection [6]. Deep learning methods such as DeepOrder learn ranking signals from historical failures and durations, but test independence assumptions remain common and code structure often receives limited modeling [2]. Hybrid learning and search-based approaches improve effectiveness in some settings, but training data demands and engineering costs can limit adoption [7]. Sequence models such as Gated Recurrent Unit (GRU)-based predictors improve temporal learning, but continual adaptation across evolving project versions remains limited [8]. Consensus approaches such as Hansie aggregate multiple metrics without learning, improving robustness but reducing adaptivity [9].

Graph-based analysis: Graph representations connect tests with code components and dependencies to support change impact reasoning. Package Regression Test Selection (PKRTS) applies package-level static dependency graphs for regression test selection, rather than fine-grained ordering, and learning from outcomes remains absent [10]. Microservice Regression Testing Selection technique based on Belief Propagation (MRTS-BP) uses belief propagation over dynamic call graphs in microservice settings to filter impacted tests, but prioritization learning remains outside the core design [11]. A Reinforcement Learning based Testcase Preoritaization (RLTCP) applies reinforcement learning in graphical user interface settings using structured graph information, but domain constraints reduce generality across broader CI workloads [12]. Graph-based work motivates structural encodings, yet learned ordering under strict CI budgets remains underexplored.

Attention and Transformer-based modeling: Attention mechanisms and Transformer architectures provide context modeling among candidate tests and related signals. Partition Ordering-Based Prioritization (OCP) [13] uses partial attention for cost reduction, but attention weights follow heuristic rules rather than learned ranking objectives [13]. Contextual signals from test names and change descriptions improve ranking features, but standardized neural sequence reasoning for CI-scale test suites remains limited [14]. Learning-to-rank methods align optimization with early fault detection metrics such as Average Percentage of Faults Detected (APFD), but attention-centered architectures remain emerging in TCP [15]. Attention transfer combined with reinforcement learning supports decision stability, but integration with structural code–test representations and continual updates remains limited [16].

Reinforcement learning for prioritization: Reinforcement learning optimizes ordering by rewarding early fault discovery. Reinforced Test Case Selection (RETECS) demonstrates history-driven prioritization with reward feedback, but training costs and limited history windows can constrain performance under rapid change [17]. Extensions targeting scalability improve runtime, yet cross-release knowledge retention remains a challenge [18]. Deep reinforcement learning variants such as Deep Reinforcement Prioritizer (DeepRP) enrich state features for large suites, but explicit integration of graph encoders, Transformer-based context modeling, and continual learning remains limited [4, 19].

Optimization-based selection: Multi-objective search approaches such as Linkage Learning-based Non-Dominated Sorting Genetic Algorithm (L2-NSGA) reduce cost by selecting fault-revealing subsets using linkage-aware crossover [20]. Selection-oriented optimization provides strong baselines for efficiency, but full in-suite ordering and feedback-driven adaptation receive less emphasis.

Continual learning in TCP: Continual or incremental updating remains uncommon in TCP. Cluster-based re-organization updates groups across cycles, and incremental data models improve scalability, but stepwise refresh strategies can reduce retention of older patterns [18, 21]. A unified framework combining structural graph encoding, sequence context modeling, reinforcement-learning-based ordering, and continual updates across CI cycles remains largely absent in the literature.

Positioning: DeepARTS unifies graph-based structural encoding, Transformer-based sequence reasoning, deep Q-network ordering, and continual learning within CI constraints. The integrated design targets stable early fault detection, low per-cycle latency, and reduced retraining overheads while preserving knowledge across releases [2, 6–21].

III. Methods and Materials

This section presents the methodological core of DeepARTS. Test suites are modeled as code–test interaction graphs, node embeddings are learned with a graph attention network, candidate tests are encoded with a Transformer, and next-test selection is produced by a deep Q-network under an explicit detection-oriented, cost-sensitive reward and an execution mask that enforces feasibility. A continual-learning loop maintains adaptation under distributional drift across continuous-integration cycles. Subsections describe the end-to-end flow, the graph neural network encoder, the Transformer sequence model, the entropy-guided replay and Maximum Mean Discrepancy (MMD)-based continual-learning mechanism, and the reinforcement-learning policy with execution constraints. Design choices prioritize early fault revelation and low per-cycle latency, and notation and dimensions are fixed for reproducibility.

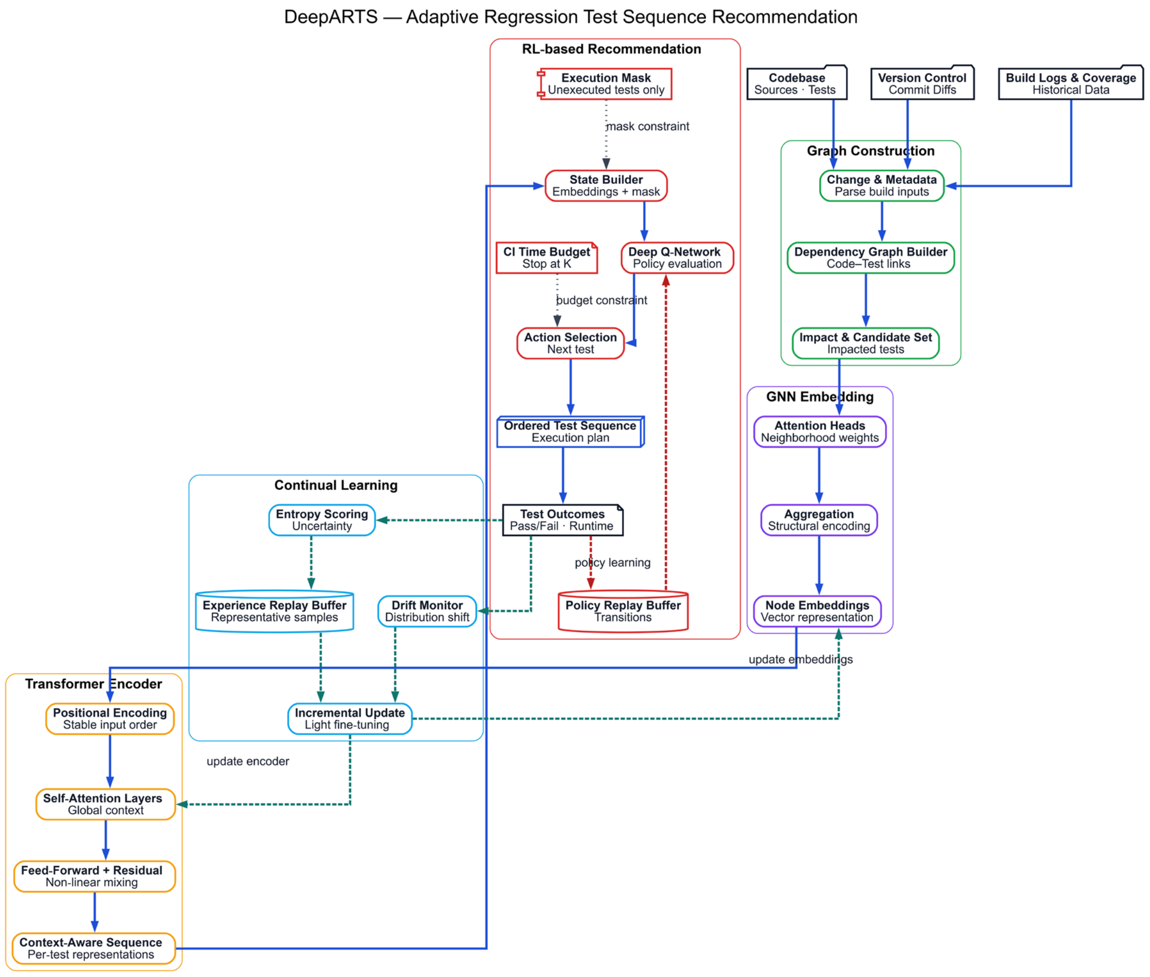

Fig. 1. DeepARTS architecture diagram of the framework.

A. Overview of the DeepARTS Framework

DeepARTS integrates graph-based representation learning, Transformer-based sequence modeling, a reinforcement-learning policy, and continual learning into a single pipeline for adaptive regression test-sequence recommendation (Fig. 1). Software test artifacts are modeled as interaction graphs $G = (V, E)$. A Graph Attention Network (GAT) computes node embeddings that capture structural dependencies by attention-weighted neighborhood aggregation. Node embeddings from the GAT are processed by a Transformer encoder to produce context-aware sequence representations with multi-head self-attention and position-wise feed-forward sublayers. A Deep Q-Network (DQN) selects the next test under a Markov decision process with state s_t, action a_t, and reward r_t. For a candidate test a_t executed at decision step t, with binary outcome $y_t \in \{0,1\}$ indicating whether the test reveals at least one fault in the current cycle, the reward is $r_t = \begin{cases} 1, & if\ y_t = 1, \\ -\lambda, & if\ y_t = 0, \end{cases}$ with $\lambda = 0.1$ in the experiments. The reward definition yields a detection-oriented, cost-sensitive pattern of values in $\{1, -0.1\}$. An execution mask $m_t$restricts the action set A_t to tests that have not yet been run in the current cycle. A continual-learning loop then performs incremental updates using an entropy-guided experience replay buffer and concept-drift monitoring via MMD, enabling parameter updates without full retraining.

Pipeline flow:

(1) Graph construction → Graph Neural Network (GNN) embedding: Build an interaction graph from code–test

dependencies and compute node embeddings with a GAT to encode structural relations (Eqs. (1)–(6)).

(2) Transformer sequence modeling: Form an ordered embedding sequence $H = [h_1, \dots, h_n]^T$ (Eq. (7)) and apply multi-head self-attention plus feed-forward sublayers to obtain context-aware representations (Eqs. (8)–(12)).

(3) RL-based recommendation (DQN): Using the current sequence representations and an execution mask over completed tests, select the next test by maximizing the learned action-value function $Q(s_t, a_t)$ under the detection-oriented, cost-sensitive reward $r_t \in \{1, -0.1\}$ defined in the overview.

(4) Continual-Learning (CL) update: Monitor drift with MMD and update parameters using recent data together with an entropy-guided replay buffer to retain prior knowledge (Eqs. (13)–(17)).

This four-stage design targets accurate early-fault detection and low per-cycle latency while adapting steadily across CI cycles.

B. Graph Neural Network (GNN) Module

Software interactions are modeled as a directed graph $G = (V, E)$, where each node $v_i \in V$ denotes a test-relevant software element (e.g., method, component) and each directed edge $e_{ij} \in E$ encodes a dependency or control-flow relation. Every node has an input feature vector $x_i \in \mathbb{R}^{d_{in}}$.

Per-head projections: To capture diverse structural relations, we use h independent attention heads. Each head $k \in \{1, \dots, h\}$ applies a learnable linear map $W^{(k)} \in \mathbb{R}^{d_h \times d_{in}}$ to obtain head-specific features Eq. (1).

$$z_i^{(k)} = W^{(k)} x_i, \qquad z_i^{(k)} \in \mathbb{R}^{d_h} \tag{1}$$

Attention logits and weights: For neighbors $j \in \mathcal{N}_i$, head k computes an attention logit with parameters $a^{(k)} \in \mathbb{R}^{2d_h}$ Eq. (2):

$$e_{ij}^{(k)} = LeakyReLU\left(a^{(k)\top}\left[z_i^{(k)} \parallel z_j^{(k)}\right]\right), \qquad j \in \mathcal{N}_i, \tag{2}$$

and normalizes over the neighborhood via softmax Eq. (3)

$$\alpha_{ij}^{(k)} = \frac{exp\left(e_{ij}^{(k)}\right)}{\sum_{u \in \mathcal{N}_i} exp\left(e_{iu}^{(k)}\right)} \tag{3}$$

Head-wise aggregation: Each head aggregates its neighbors' projected features Eq. (4):

$$h_i^{(k)} = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^{(k)} z_j^{(k)}\right) \tag{4}$$

where $\sigma(\cdot)$ is a pointwise nonlinearity (e.g., Exponential Linear Unit (ELU)).

Multi-head combination: The head outputs are concatenated to form a rich node embedding Eq. (5):

$$u_i = \Vert_{k=1}^{h} h_i^{(k)} \in \mathbb{R}^{h\, d_h} \tag{5}$$

Dimension alignment for the Transformer: The concatenated vector is optionally projected to the model dimension d with $W^O \in \mathbb{R}^{d \times (h\, d_h)}$ Eq. (6):

$$h_i = W^O u_i \in \mathbb{R}^d, \qquad withd = h\, d_h if W^O is identity \tag{6}$$

The resulting $\{h_i\}$ serve as the input embeddings to the Transformer (Eqs. (7)–(12)). This formulation keeps all computations head-consistent (no mixing of x_j and h'_j), standardizes the head count as h, and yields a final embedding of size d expected by the next stage.

C. Transformer-Based Sequence Modeling

The Transformer encoder operates on GNN-generated node embeddings to capture sequential and temporal dependencies relevant to test execution order. The input is an ordered, deterministic sequence of all candidate tests for the current CI cycle Eq. (7):

$$H = [h_1, h_2, \dots, h_n]^\top, \quad H \in \mathbb{R}^{n \times d} \tag{7}$$

Sequence order: To ensure reproducibility and to decouple ranking from preprocessing, we impose a stable order by test identifier (lexicographic by fully qualified test name) within each cycle; newly appearing tests are inserted by the same rule.

Positional encoding: Before the first encoder layer, we use learned absolute positional embeddings $\{p_i \in \mathbb{R}^d\}_{i=1}^n$ and feed $\tilde{h}_i = h_i + p_i$ to the encoder.

Masking Policy: Executed tests are not masked in the Transformer; masking is applied only by the DQN through the execution mask m_t and the constrained action set A_t. The resulting scheme keeps the encoder context-aware over all candidates while letting the policy enforce feasibility.

Multi-head self-attention: For encoder layer $\ell \in \{1, \dots, L\}$, queries, keys, and values are computed via learned projections Eq. (8):

$$Q^{(\ell)} = H^{(\ell-1)} W_Q^{(\ell)}, \quad K^{(\ell)} = H^{(\ell-1)} W_K^{(\ell)}, \quad V^{(\ell)} = H^{(\ell-1)} W_V^{(\ell)}, \tag{8}$$

with $W_Q^{(\ell)}, W_K^{(\ell)}, W_V^{(\ell)} \in \mathbb{R}^{d \times d_k}$, $d_k = d/h$, and h attention heads. Head i performs scaled dot-product attention Eq. (9):

$$head_i = softmax\left(\frac{Q_i^{(\ell)} K_i^{(\ell)\top}}{\sqrt{d_k}}\right) V_i^{(\ell)} \tag{9}$$

Heads are concatenated and projected Eq. (10):

$$O^{(\ell)} = [\, head_1; head_2; \dots; head_h \,]\, W_O^{(\ell)}, \quad W_O^{(\ell)} \in \mathbb{R}^{hd_k \times d} \tag{10}$$

Position-wise feed-forward: Each layer includes a two-layer network Eq. (11):

$$FFN\left(O^{(\ell)}\right) = ReLU\left(O^{(\ell)} W_1^{(\ell)} + b_1^{(\ell)}\right) W_2^{(\ell)} + b_2^{(\ell)} \tag{11}$$

with $W_1^{(\ell)} \in \mathbb{R}^{d \times d_{ff}}$ and $W_2^{(\ell)} \in \mathbb{R}^{d_{ff} \times d}$.

Residual connections and normalization: Residual paths and LayerNorm stabilize training Eq. (12):

$$Z^{(\ell)} = LayerNorm\left(O^{(\ell)} + H^{(\ell-1)}\right), \quad H^{(\ell)} = LayerNorm\left(FFN\left(Z^{(\ell)}\right) + Z^{(\ell)}\right) \tag{12}$$

with $L = 4$ encoder layers and $h = 4$ heads, the final sequence $H^{(L)}$ provides context-aware embeddings for downstream DQN-based recommendation.

D. *Continual Learning Strategy*

The Continual-Learning (CL) mechanism incrementally adapts the GNN and Transformer parameters across CI cycles using an entropy-guided experience replay buffer and concept-drift monitoring via MMD. Let the dataset at cycle t be Eq. (13)

$$D^{(t)} = \{(x_i^{(t)}, y_i^{(t)})\}_{i=1}^{N_t} \tag{13}$$

where $x_i^{(t)} \in \mathbb{R}^d$ are encoder embeddings and $y_i^{(t)} \in \{0,1\}$ indicates whether the test revealed a fault (failure/no-failure). An auxiliary binary predictive head on top of the encoder provides $P(y = 1|x;\theta)$ used only for uncertainty scoring and CL updates (ranking via RL remains unchanged).

Uncertainty scoring (entropy): With $C = 2$ classes, Eq. (14) as shown in below:

$$H\left(x_i^{(t)}\right) = -\sum_{c=1}^{C} P\left(y_i^{(t)} = c|x_i^{(t)};\theta\right) logP\left(y_i^{(t)} = c|x_i^{(t)};\theta\right) \tag{14}$$

Experience Replay Buffer (ERB) with partial replacement: The buffer B maintains M representative samples (default $M = 0.2$ of all accumulated samples). At each cycle, only a fraction r of B is replaced by high-entropy current samples, keeping the remaining $(1 - r)$ fraction from the previous buffer Eq. (15):

$$B^{(t)} = KeepTop_{(1-r)M}\left(B^{(t-1)};\, H(x)\right) \cup Top_{rM}\left(D^{(t)};\, H(x)\right), \quad r = 0.2 \tag{15}$$

Concept-drift detection (MMD): Distribution shift between consecutive cycles is measured with MMD using a Gaussian kernel $k_\sigma(\cdot,\cdot)$ (bandwidth σ by the median heuristic over the union of samples compared) Eq. (16):

$$MMD^2\left(D^{(t-1)}, D^{(t)}\right) = \frac{1}{N_{t-1}^2}\sum_{i,j} k\left(x_i^{(t-1)}, x_j^{(t-1)}\right) + \frac{1}{N_t^2}\sum_{i,j} k\left(x_i^{(t)}, x_j^{(t)}\right) - \frac{2}{N_{t-1}N_t}\sum_{i,j} k\left(x_i^{(t-1)}, x_j^{(t)}\right) \tag{16}$$

An incremental update is triggered when Eq. (17)

$$MMD^2\left(D^{(t-1)}, D^{(t)}\right) > \tau \tag{17}$$

Threshold calibration: τ is set per project as the 95th percentile of recent MMD^2 values computed over a 3-cycle moving window (pairs $(t-3 \to t-2)$, $(t-2 \to t-1), (t-1 \to t)$). Each MMD^2 uses all samples observed in the corresponding cycle pair (i.e., the union of $D^{(u)}$ and $D^{(v)}$ for each pair $(u \to v)$). This yields a stable, data-adaptive trigger without over-updating.

Incremental fine-tuning: When triggered, parameters θ are updated on a mixture of buffered and current-cycle data via a weighted objective Eq. (18):

$$L^{(t)}(\theta) = \gamma\, L_{B^{(t)}}(\theta) + (1-\gamma)\, L_{D^{(t)}}(\theta), \ \gamma = 0.5, \tag{18}$$

where $L_{B^{(t)}}$ and $L_{D^{(t)}}$ are the task losses (e.g., cross-entropy for the auxiliary head) evaluated on $B^{(t)}$ and $D^{(t)}$, respectively. Unless stated otherwise, we use $M = 0.2$ of accumulated samples and $r = 0.2$ per cycle for ERB updates.

This entropy-guided buffering, per-project MMD thresholding, and brief weighted fine-tuning together provide stable–plastic adaptation—retaining prior knowledge while adjusting to distributional change—without incurring full retraining overhead.

IV. Experimental Study

This section evaluates DeepARTS under continuous-integration constraints, focusing on early fault detection and latency. Evaluation uses RTPTorrent with a chronological split to avoid temporal leakage. Comparisons are performed under hardware parity, identical candidate sets, and the same CI budget *K*. Effectiveness is measured by APFD (primary), Precision@K, and Recall@K; latency denotes per-cycle Central Processing Unit (CPU)-only recommendation time, excluding offline training. Statistical testing uses paired analyses across projects with confidence intervals, and ablation studies isolate the roles of the GNN, Transformer, RL policy, and continual learning.

A. *Dataset Description (RTPTorrent)*

RTPTorrent [22] is the main evaluation corpus, and the associated archived dataset provides 20 open-source Java projects with more than 100{,}000 TravisCI build logs that cover diverse sizes and histories. A chronological per-project split, with older builds used for training and validation and the most recent builds used for testing, avoids temporal leakage. Fig. 1 shows the pipeline.

Candidate-set construction (per CI cycle)

- Change ingestion: extract commit diff and build metadata; map changed files to program entities.
- Historical coverage: maintain per-test coverage cache (JaCoCo-compatible); when missing, rely on static dependencies only.
- Static dependencies: build a code–test dependency graph via static analysis; also used by the GNN encoder.
- Impact rules: a test is impacted if (1) historical coverage intersects changed methods, or (2) static links

touch a changed method or its one-hop neighbor; take the union when both sources exist.

- Model flow: encode impacted tests with the GNN, form a deterministic sequence for the Transformer, rank with the DQN, and apply continual updates across cycles.

The procedure yields a reproducible candidate set and supports consistent APFD and Precision@K/Recall@K computation per cycle.

B. Experimental Setup

Hardware and software: Experiments executed on a single workstation with an Intel Xeon 3.0 GHz processor (16 cores), an NVIDIA RTX 3090 GPU (24 GB), and 128 GB RAM. Implementation used PyTorch 2.2.0, PyTorch Geometric 2.3.1, and Stable-Baselines3 1.7.0. Source code, configuration files, and scripts for reproducing the experiments.

Hardware-parity policy:

- All methods ran on the same workstation and used the same candidate set and CI budget K per cycle. Reported overhead is per-cycle recommendation time only (from candidate assembly to final order); offline training/tuning excluded. This aligns with the latency definition used in Table I.
- To ensure parity with the CPU-centric baseline, DeepARTS inference was measured on CPU only; the GPU was used only for offline training. L2-NSGA executed on CPU. Thread parallelism was capped at 16 for all components to match physical cores and avoid thread-count advantages. This isolates algorithmic efficiency from accelerator effects and justifies the cross-method latency comparison.

Model configurations:

- GNN (GAT): two layers, $h = 4$ heads, ELU, 128-dim node embeddings, dropout 0.2.
- Transformer: $L = 4$ layers, $h = 4$ heads, model dim 128, Feed-Forward Network (FFN) 256, dropout 0.1, LayerNorm.
- DQN: 3-layer Multi-Layer Perceptron (MLP) (256–128–64), ε-greedy policy from 1.0 to 0.05 over 10{,}000 steps; reward defined as in the RL-based recommendation policy in Section III, with unit per-test cost and $\lambda = 0.1$.
- Learning rates: 1×10^{-3} for initial training; 1×10^{-4} for continual updates. Further protocol details and seeds are in Section IV.C.

Baseline configuration and equivalence controls: L2-NSGA used the same candidate set and budget K; its runtime reflects per-cycle search time on the same CPU. Offline hyper-parameter tuning was excluded for both methods. These controls support valid effectiveness and latency comparisons summarized in Tables I and II, and Figs. 2–5.

C. Training Procedure

- Split & runs: Chronological, per-project split: early builds → train/val; most recent → test. Results aggregated over 3 seeds (42, 123, 2025).
- Optimization: Mini-batch 32; 100 epochs with early stopping on validation APFD. Learning rates: 1×10^{-3} (initial), 1×10^{-4} (continual). DQN ε-greedy 1.0 → 0.05 over 10k steps; reward $+1/-0.1$ (unit cost).
- Continual learning: Replay buffer size $M = 20\%$ of accumulated samples; partial replacement $r = 0.2$ per cycle using entropy (binary head, $C = 2$). Drift by MMD with Gaussian kernel (median heuristic); trigger at 95th percentile, per project, over a 3-cycle window. Update loss $L^{(t)} = \gamma L_{B^{(t)}} + (1-\gamma) L_{D^{(t)}}, \gamma = 0.5$.
- Runtime accounting and parity: Reported per-cycle latency = inference/recommendation on CPU from candidate assembly to final order; offline training/tuning excluded. L2-NSGA evaluated on the same CPU, using the same candidate set and CI budget K; time reflects per-cycle search only.
- Statistics: Metrics computed per cycle, paired on aligned cycles within each project; per-project means averaged across seeds yield $n = 20$ paired observations. Paired t-test and Cohen's d reported; 95% CIs via bootstrap across projects.

This procedure enables fair, reproducible effectiveness and latency comparisons under CI constraints.

D. Performance Metrics

Scope: Test sequencing is treated as a ranking task. Metrics quantify (1) how early fault-revealing tests appear in the recommended order and (2) the cost to generate that order; plain classification Precision/Recall are not used.

Precision@K: Let R_K be the top-K recommended tests in a CI cycle and F the set of fault-revealing tests in that cycle (Eq. (19)):

$$Precision@K = \frac{|R_K \cap F|}{K} \tag{19}$$

Here, K equals the per-cycle CI budget (tests permitted by the time limit). If a fixed K is used for reporting, that value is stated alongside results.

Recall@K:

$$Recall@K = \frac{|R_K \cap F|}{|F|} \tag{20}$$

Eq. (20) measures the fraction of all fault-revealing tests that appear within the top-K recommendations in the cycle.

APFD: Over a full ranking of n tests with m distinct faults, let TF_i be the 1-based position of the first test that reveals fault i (Eq. (21)):

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{n \times m} + \frac{1}{2n} \tag{21}$$

Higher is better (earlier detection). APFD is the primary effectiveness metric. For any method that outputs only a subset, a canonical order is derived using the heuristic described in Section IV.D to enable APFD comparability.

Runtime Overhead: Wall-clock time per CI cycle to produce a recommendation, measured from candidate-set assembly to the final ranked order on the evaluation hardware; offline training/hyper-parameter search are excluded.

Aggregation and reporting: Metrics are computed per cycle, averaged within each project across cycles, then aggregated across projects (means and 95% CIs; seeds aggregated).

E. Results and Discussion

The overall comparison as shown in Table I under the same CPU, candidate set, and CI budget K, DeepARTS achieves APFD 0.94, Precision@K 99.2%, and Recall@K 98.7%, outperforming L2-NSGA [20] (0.87, 94.1%, 92.8%, respectively). Per-cycle recommendation time is 2.5 s for DeepARTS versus 2.1 s for L2-NSGA. Paired tests on project-level means across aligned cycles ($n = 20$) show $p < 0.01$ for all three effectiveness metrics; 95% confidence intervals for the differences exclude zero, and effect sizes (Cohen's d) are large.

TABLE I. PERFORMANCE COMPARISON ON RTPTORRENT

Method	Precision@K (%)	Recall@K (%)	APFD	Runtime per CI cycle (s)
L2-NSGA [20]	94.1	92.8	0.87	2.1
DeepARTS	99.2	98.7	0.94	2.5

APFD comparability: L2-NSGA yields an unordered subset of tests rather than a full ranking. For APFD, a canonical order is therefore derived by first ranking all selected tests by historical failure rate in descending order, with shorter execution duration as a tie-break, and then appending the remaining non-selected tests using the same rule. This procedure uses only information that is directly available to the baseline and does not introduce any signals from DeepARTS. Alternative canonicalizations, such as random permutations within the selected subset or pure duration-based ordering, would also be reasonable; studying their quantitative effect on APFD values is a direction for future work.

Stability across versions: Version-wise trends presented in Figs. 2 and 3 show Precision@K $\geq\approx$98.5% and APFD $\geq\approx$0.93 for DeepARTS across five consecutive version windows. L2-NSGA degrades from ≈94.1% to ≈89.4% (Precision@K) and 0.87 to 0.81 (APFD), indicating stronger robustness from structural encoding, sequence context, RL ordering, and continual updates.

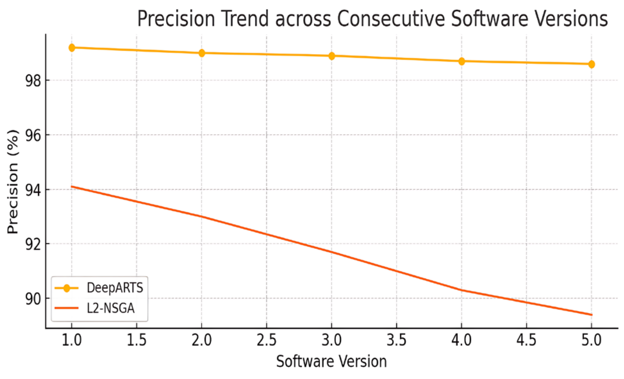

Fig. 2. Precision@K across software version windows (DeepARTS vs. L2-NSGA).

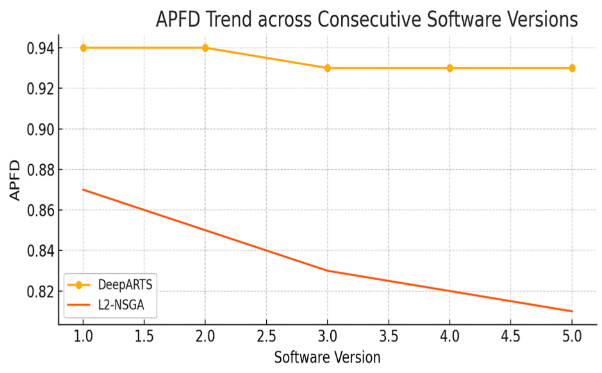

Fig. 3. APFD across software version windows (DeepARTS vs. L2-NSGA).

Component contributions: Removing any single module reduces early-fault detection.

TABLE II. ABLATION STUDY RESULTS FOR DEEPARTS COMPONENTS

Model Variant	Precision (%)	Recall (%)	APFD
DeepARTS (Full Model)	99.2	98.7	0.94
Without GNN	94.5	94.0	0.89
Without Transformer	93.8	92.9	0.89
Without RL	94.2	93.7	0.88
Without Continual Learning	95.4	94.8	0.90

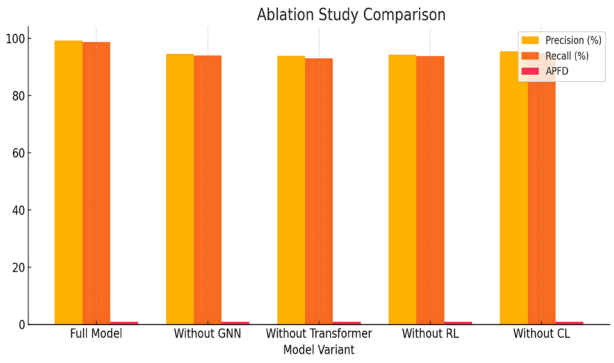

Fig. 4. Ablation study results (Precision@K, Recall@K, APFD).

- No GNN: 94.5% Precision@K, 0.89 APFD (−4.7 percentage points (pp), −0.05).
- No Transformer: 93.8%, 0.89 (−5.4 pp, −0.05).
- No RL: 94.2%, 0.88 (−0.06 APFD).
- No Continual Learning: 95.4%, 0.90 (−3.8 pp, −0.04).

For reproducibility, ablations shown in Table II that visualize Fig. 4 produce the final order without RL by sorting tests by the auxiliary head's predicted failure probability (ties: shorter duration); without Transformer by applying the same head to the GNN embeddings; without GNN by feeding per-test features directly to the Transformer and then scoring; without CL by freezing parameters after initial training (no replay-based updates).

Scalability and latency: Cycle-level recommendation time scales shown in Fig. 5 near-linearly with suite size: ≈ 2.0 s @ 200 tests to ≈ 4.8 s @ 1000 tests; values reflect inference/recommendation only on the evaluation CPU (offline training excluded for all methods). These timings support CI-time budgets while preserving early-detection quality.

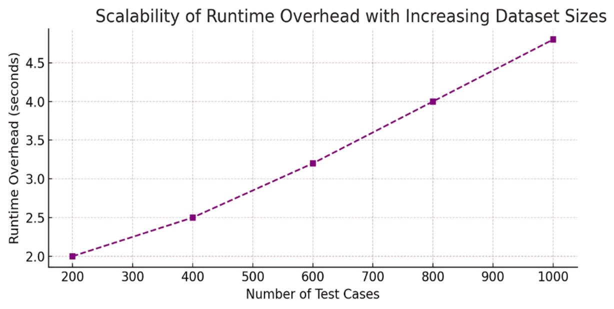

Fig. 5. Per-cycle recommendation latency vs. test-suite size.

Interpretation: The combination of graph structure, sequence attention, RL-based ordering, and continual learning yields higher early-fault yield at practical latency under realistic CI constraints, with consistent advantages across versions and strong ablation evidence for each module's contribution.

Code availability: The DeepARTS implementation will be released as open source within 18 to 24 months of journal acceptance and is currently available from the corresponding author upon request.

V. Conclusion

DeepARTS provides an end-to-end framework for regression test sequence recommendation under continuous integration constraints by integrating graph-based code–test dependency encoding, Transformer-based context modeling among candidate tests, deep reinforcement learning for sequential ordering, and continual learning for cross-cycle adaptation. Graph attention encodes structural relations, the Transformer captures interactions among candidates, and a deep Q-network selects tests under a detection-oriented reward with an execution mask. Entropy-guided replay and MMD drift monitoring support parameter updates across cycles without full retraining.

Evaluation on RTPTorrent, a corpus containing 20 open-source Java projects and more than 100,000 TravisCI build logs, indicates improved early fault detection compared with L2-NSGA under identical candidate sets and a fixed per-cycle budget K. Results include APFD $= 0.94$, $Precision@K = 99.2\%$, and $Recall@K = 98.7\%$, with per-cycle recommendation latency near 2.5 s on CPU. Ablation results show reduced effectiveness after removal of graph encoding, Transformer context modeling, reinforcement-learning-based ordering, or continual learning, indicating complementary contributions across the integrated modules.

Limitations and threats to validity arise from dependence on historical coverage quality and static dependency extraction, from restriction to Java projects using TravisCI in the RTPTorrent dataset, and from focus on CPU-only inference latency. Incomplete or noisy coverage or dependency information may reduce recommendation accuracy, and datasets from other languages, build tools, or continuous-integration services may exhibit different characteristics. Reported latency values cover only inference and recommendation on a single workstation and exclude offline training and hyperparameter tuning, so absolute execution times may differ under alternative hardware or deployment settings. Computation of APFD for L2-NSGA relies on a canonical ordering derived from historical failure rates and test durations, which may influence absolute APFD scores for that baseline even though the ordering uses only information available to L2-NSGA.

Practical deployment in industrial CI/CD environments requires robustness to flaky tests and noisy outcomes and stronger evidence for generalization beyond the studied corpus. Flaky and unstable test cases can introduce inconsistent rewards for the deep Q-network and may lead to unstable or biased policies, especially when failures do not reproduce across cycles. Possible mitigation strategies include explicit detection and down-weighting of flaky tests, aggregation of reward feedback over multiple cycles, and inclusion of a flakiness-risk component in the reward design. Future evaluation on additional industrial datasets and other ecosystems, such as projects implemented in languages beyond Java and pipelines based on services such as GitHub Actions, would provide stronger evidence for generalization and for the robustness of DeepARTS under realistic operational constraints.

Further extensions of the continual-learning mechanism and the reward formulation may enable multi-objective optimization over early fault detection, flakiness risk, and varying time budgets in large-scale continuous-integration environments.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

SRK: Conceptualization; problem formulation; methodology; system design (graph encoder, sequence model, RL policy); supervision; project administration. ARA: Software implementation (GNN/Transformer, CI data ingestion pipelines); data curation (RTPTorrent processing); experiments (including ablation and latency studies); visualization; writing—original draft. RRP: Methodology (reward design and constraints); software (DQN and continual-learning modules: replay buffer and drift detection); validation; formal analysis; evaluation protocol; statistical analysis; writing—review and editing; All authors: interpretation of results; critical revision of the manuscript; all authors had approved the final version.

Acknowledgment

The authors thank the maintainers and contributors of the RTPTorrent dataset and the continuous-integration communities whose publicly available logs and tooling enabled the experiments. Appreciation is extended to the open-source developers whose libraries supported data processing, modeling, and evaluation, and to the authors' institutions for computational resources and administrative support. Constructive feedback from colleagues during internal reviews is gratefully acknowledged.

REFERENCES

[1] A. Da Roza, J. A. P. Lima, R. C. Silva *et al.,* "Machine learning regression techniques for test case prioritization in continuous integration environment," in *Proc. 2022 IEEE International Conf. on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 196–206. https://doi.org/10.1109/saner53432.2022.00034

[2] A. Sharif, M. Dusica, and M. Liaaen, "DeepOrder: Deep learning for test case prioritization in continuous integration testing," in *Proc. 2021 IEEE International Conf. on Software Maintenance and Evolution (ICSME)*, 2021, pp. 525–534. https://doi.org/10.1109/icsme52107.2021.00053

[3] H. Spieker, A. Gotlieb, D. Marijan *et al.,* "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proc. the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 12–22. https://doi.org/10.1145/3092703.3092709

[4] R. Shankar and D. Sridhar, "An improved deep learning based test case prioritization using deep reinforcement learning," *International Journal of Intelligent Engineering and Systems,* vol. 17, no. 1, pp. 771–82, 2024. https://doi.org/10.22266/ijies2024.0229.64

[5] Y. F. Zhao and D. Hao, "Test case prioritization technique in continuous integration based on reinforcement learning," *J. Softw.,* vol. 34, no. 6, pp. 2708–2726, 2023. doi: 10.13328/j.cnki.jos.006506 (in Chinese)

[6] P. Rongqi, M. Bagherzadeh, T. A. Ghaleb *et al.,* "Test case selection and prioritization using machine learning: A systematic literature review," *Empirical Software Engineering*, vol. 27, no. 2, 29, 2021. https://doi.org/10.1007/s10664-021-10066-6

[7] N. Medhat, S. M. Moussa, N. L. Badr *et al.,* "A framework for continuous regression and integration testing in IoT systems based on deep learning and search-based techniques," *IEEE Access*, vol. 8, pp. 215716–215726, 2020. https://doi.org/10.1109/access.2020.3039931

[8] A. Behera and A. A. Acharya. "An effective GRU-based deep learning method for test case prioritization in continuous integration testing," *Procedia Computer Science*, vol. 258, pp. 4070–4083, 2025. https://doi.org/10.1016/j.procs.2025.04.658

[9] S. Mondal and R. Nasre. "Hansie: Hybrid and consensus regression test prioritization," *Journal of Systems and Software*, vol. 172, 110850, 2021. https://doi.org/10.1016/j.jss.2020.110850

[10] M. Al-Refai and M. M. Hammad, "A package level regression test selection approach for java software systems," *IEEE Access*, vol. 13, pp. 87848–87861, 2025. https://doi.org/10.1109/access.2025.3570007

[11] L. Chen, J. Wu, H. Yang *et al.,* "A microservice regression testing selection approach based on belief propagation," *Journal of Cloud Computing*, vol. 12, no. 1, 2023. https://doi.org/10.1186/s13677-023-00398-7

[12] V. Nguyen and B. Le, "RLTCP: A reinforcement learning approach to prioritizing automated user interface tests," *Information and Software Technology*, vol. 136, 106574, 2021. https://doi.org/10.1016/j.infsof.2021.106574

[13] Q. Zhang, C. Fang, W. Sun *et al.,* "Test case prioritization using partial attention," *Journal of Systems and Software*, vol. 192, 111419, 2022. https://doi.org/10.2139/ssrn.4068676

[14] E. A. da Roza, J. A. do P. Lima, and S. R. Vergilio, "On the use of contextual information for machine learning based test case prioritization in continuous integration development," *Information and Software Technology*, vol. 171, 107444, 2024. https://doi.org/10.1016/j.infsof.2024.107444

[15] P. Tang, J. Wang, and M. Liu, "Variational learning to rank for test case prioritization via prioritizing metric inspired differentiable loss," *Engineering Applications of Artificial Intelligence*, vol. 141, 109776, 2025. https://doi.org/10.1016/j.engappai.2024.109776

[16] Q. Su, X. Li, Y. Ren *et al.,* "Attention transfer reinforcement learning for test case prioritization in continuous integration," *Applied Sciences*, vol. 15, no. 4, 2243, 2025. https://doi.org/10.3390/app15042243

[17] M. Bagherzadeh, N. Kahani, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2836–2856, 2021. https://doi.org/10.1109/tse.2021.3070549

[18] A. S. Yaraghi, M. Bagherzadeh, N. Kahani *et al.,* "Scalable and accurate test case prioritization in continuous integration contexts," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1615–1639, 2023. https://doi.org/10.1109/tse.2022.3184842

[19] R. Shankar and D. D. Sridhar, "A comprehensive review on test case prioritization in continuous integration platforms," *Int. J. Innov. Sci. Res. Technol.*, vol. 8, no. 4, pp. 3223–3229, 2023.

[20] M. Olsthoorn and A. Panichella, "Multi-objective test case selection through linkage learning-based crossover," in *Proc. Search-Based Software Engineering*, 2021, pp. 87–102. https://doi.org/10.1007/978-3-030-88106-1_7

[21] X. Wang and S. Zhang, "Cluster-based adaptive test case prioritization," *Information and Software Technology*, vol. 165, 107339, 2024. https://doi.org/10.1016/j.infsof.2023.107339

[22] T. Mattis, R. Patrick, F. Dürsch *et al.,* "Rtptorrent: An open-source dataset for evaluating regression test prioritization," in *Proc. the 17th International Conf. on Mining Software Repositories*, 2020, pp. 385–396. https://doi.org/10.1145/3379597.3387458