Program Structure-aware Language Models: Targeted Software Testing beyond Textual Semantics

Khang Tran, Khoa Nguyen, Cristian Borcea, NhatHai Phan
New Jersey Institute of Technology, Newark, NJ, USA
{kt36, nk569, borcea, phan}@njit.edu

Abstract

Recent advances in large language models for test case generation have improved branch coverage via prompt-engineered mutations. However, they still lack principled mechanisms for steering models toward specific high-risk execution branches, limiting their effectiveness for discovering subtle bugs and security vulnerabilities. We propose `GLMTest`, the first program structure-aware LLM framework for targeted test case generation that seamlessly integrates code property graphs and code semantics using a graph neural network and a language model to condition test case generation on execution branches. This structured conditioning enables controllable and branch-targeted test case generation, thereby potentially enhancing bug and security risk discovery. Experiments on real-world projects show that `GLMTest` built on a Qwen2.5-Coder-7B-Instruct model improves branch accuracy from 27.4% to 50.2% on TestGenEval benchmark compared with state-of-the-art LLMs, i.e., Sonnet-4.5 from Claude and 4o-mini from OpenAI.

1 Introduction

Testing is a cornerstone of modern software development, serving to validate program correctness and uncover functional defects before deployment (Battina, 2019; Wang et al., 2024, 2025). It is equally critical for software security, as systematically generated test cases can reveal crashes, anomalous behaviors, and exploitable vulnerabilities (Liang et al., 2018; Zhu et al., 2022). Consequently, testing accounts for a substantial portion of software engineering effort, as reflected in industry reports (KPMG, 2024). The rising cost and complexity of software systems have therefore heightened the demand for automated test generation techniques that improve testing efficiency, effectiveness, and coverage (Brunetto et al., 2021; Baqar and Khanda, 2025).

Motivation. Recent advances in large language models (LLMs) have enabled new approaches to automated test-case generation (Harman et al., 2025; Lemieux et al., 2023; Pan et al., 2025). In particular, LLMs have been used to mutate test cases to expand execution coverage (Harman et al., 2025; Lemieux et al., 2023; Pan et al., 2025). However, most existing methods rely on prompt-engineering heuristics, making the mutation process difficult to control due to LLM stochasticity and lacking principled optimization to target specific execution branches or high-risk code regions. Consequently, prompt-engineered test cases often fail to exercise security-critical paths, limiting their effectiveness in bug discovery (Weissberg et al., 2024). This highlights the need for explicitly optimized test-generation techniques that target high-risk execution branches.

Challenges. Developing LLM-based mechanisms for generating test cases targeting specific execution branches is challenging. Even with greedy sampling, the inherent stochasticity of LLMs (Astekin et al., 2024; Song et al., 2025) makes outputs difficult to control, often failing to reach the intended branches (Huang et al., 2025; Feng et al., 2025). Moreover, purely textual representations do not adequately capture dependencies among code objects, limiting the model's understanding of program structure and execution behavior and hindering precise execution guidance.

Our Solution. We propose **`GLMTest`** to encode the program (under test) by transforming its graph representation and developer-provided textual information into a shared high-dimensional embedding space using a heterogeneous graph neural network (GNN) and an LLM. Unlike prior works (Chen et al., 2025; Liu et al., 2025a) that typically feed graph features into an encoder and then query an LLM only at the sequence level, `GLMTest` jointly trains a heterogeneous GNN and an LLM to learn node-level embeddings that are directly

Findings of the Association for Computational Linguistics: ACL 2026, pages 11113–11126
July 2-7, 2026

aligned with branch-specific execution masks and injected into the LLM as branch-conditioned inputs, explicitly tailoring the graph representation to targeted test case generation rather than generic code understanding. Thus, GLMTest provides a controllable mechanism for generating test cases that execute targeted program locations.

At inference time, GLMTest can generate test cases oriented toward specific targeted locations in the code, providing a practical way to exercise high-risk branches and potentially expose underlying defects or security risks. Furthermore, GLMTest can be applied in a coverage-oriented setting to systematically expand coverage for regression and coverage-driven testing pipelines. In both cases, GLMTest offers finer-grained control over which execution paths are exercised than prior prompt-engineered LLM approaches, enabling more precise and interpretable test case generation.

Contributions. Our contributions are as follows: **(1)** We present GLMTest, the first graph-enhanced language modeling framework for branch-targeted test case generation. By jointly modeling program structure and textual semantics, GLMTest enables focused testing of high-risk branches and systematic exploration to improve branch coverage. **(2)** We also introduce a new dataset derived from real-world repositories and a training strategy that learns fine-grained, branch-oriented embeddings for targeted test generation[1]. **(3)** Experiments on Python programs from the TestGenEval benchmark show that GLMTest significantly outperforms enterprise LLMs (e.g., Claude-Sonnet-4.5), improving branch accuracy from 27.4% to 50.2% while achieving high branch coverage.

2 Background & Related Work

LLMs for Test Case Generation. LLMs have become central to automated code generation, improving software development workflows (Parvez et al., 2018). Trained on large-scale open-source code and fine-tuned for instruction following (Roziere et al., 2023), they have recently been applied to software testing to generate readable, correct test suites with improved coverage (Tufano et al., 2021). Existing approaches fall into two categories: fine-tuning and prompt engineering. Fine-tuning methods specialize LLMs for test generation using curated code–test pairs (Tufano et al., 2021; Alagarsamy et al., 2024; Rao et al., 2024), while prompt-based methods guide frozen LLMs with structured program features (e.g., signatures and control flow) to achieve coverage-oriented test generation (Siddiq et al., 2024; Chen et al., 2024; Dakhel et al., 2024).

Code Property Graph (CPG). In software analysis, graph representations encode relationships among program elements and serve as structured inputs for program analysis (Bilot et al., 2024). Common examples include abstract syntax trees for syntactic structure (White et al., 2016), control-flow graphs for execution paths (Zhao and Huang, 2018), and data-flow graphs for data dependencies (Nielson et al., 2010). Recent work (Lekssays et al., 2025; Chen et al., 2025) explores combining code graphs with LLMs, mainly using graphs as auxiliary knowledge to enrich prompts rather than tightly integrating program structure and code semantics into an optimized model for test generation (Ryan et al., 2024a). More information on related work can be found in Appendix B.

3 Problem Formulation

CPG Annotation. The CPG can be defined as $G = (V, E)$, where V is the set of nodes and E is the set of edges. Each node $v \in V$ corresponds to a program element (e.g., a statement, expression, variable, or function) and is associated with a feature vector $x_v \in \mathbb{R}^d$. This vector can encode multiple static attributes, such as the tokenized code snippet, syntactic type (e.g., assignment, call, branch), and program location (e.g., file, line, and column). By stacking all node features, we obtain the node feature matrix $X \in \mathbb{R}^{|V| \times d}$. The edge set E is partitioned into subsets $E = \bigcup_j^r E_j$, where E_j contains edges of type j (e.g., abstract syntax tree) and r is the total number of edge relations. In this way, the CPG represents the program as a heterogeneous, multi-relational graph that captures the program's structural information.

Setting of GLMTest. Given a program S, the test case generator produces a test suite $\tau = \{t_i\}_{i \in [1,n]}$, where n is the number of generated test cases. An *execution branch* is the control-flow path the program takes for a given input, i.e., the ordered sequence of executed decisions and statements executed for a test case (Ammann and Offutt, 2008). Thus, we consider that by executing test case $t_i \in \tau$ on S we can extract an execution branch $\hat{b}_i$ as the ordered sequence of statements executed by S un-

[1]Our implementation and dataset can be found here: https://github.com/khangtran2020/glmtest

```
1  def process_login(is_logged_in, has_subscription):
2      if is_logged_in:
3          login_status = "loggedin"
4      else:
5          login_status = "loggedout"
6      if has_subscription:
7          access_level = "premium"
8      else:
9          access_level = "free"
10     return login_status + "-" + access_level
11
12 assert (process_login(True, False) == "loggedin-free")
13 # Branch: 1 -> 2 -> 3 -> 6 -> 8 -> 9 -> 10
14
15 assert (process_login(True, True) == "loggedin-premium")
16 # Branch: 1 -> 2 -> 3 -> 6 -> 7 -> 10
```

Figure 1: A Python function example annotated with line numbers and branch paths. Two test cases (Lines 12 and 15) are shown with their corresponding execution branches (Lines 13 and 16), illustrating how different input combinations traverse distinct branches.

der test case t_i. Figure 1 illustrates two test cases for `process_login`: the first call in line 12 drives execution along the branch indicated in line 13, while the second call in line 15 follows the branch indicated in line 16. We denote $\hat{b}_i = \texttt{Exec}_S(t_i)$ as the process of executing t_i on S which returns the execution branch $\hat{b}_i$. We note that two test cases $t_i, t_j \in \tau$ can derive the same execution branch, i.e., $\hat{b}_i = \hat{b}_j$ since they can produce the same control-path of the program. Let $B_S = \{b_i\}_{i\in[1,m]}$ denote the set of all possible execution branches of S, where m is the total number of branches. The conventional coverage-driven objective of test case generation is to generate a test suite whose induced branches maximize coverage over possible execution branches B_S.

Goals. In this work, we consider an objective tailored to branch-targeted test case generation while remaining aligned with the conventional coverage-driven objective. Specifically, `GLMTest` trains a model f_θ that takes as input the program S and a target execution branch b indicated by the developers, and outputs a test case $\hat{t}$. The training objective is to maximize the probability that executing the generated test on S realizes the target branch:

$$\theta^* = \arg\max_\theta \sum_{b\in B_S} \Pr\left[\texttt{Exec}_s(\hat{t}) = b | \hat{t} = f_\theta(S, b)\right]. \quad (1)$$

By optimizing this objective, `GLMTest` is tailored to generate test cases whose execution on S will align with the targeted execution branch. Furthermore, this also allows `GLMTest` to iterate over branches in B_S and synthesize a test suite that improves coverage over B_S, thereby maximizing the execution branch coverage.

4 GLMTest Framework

This section describes `GLMTest` in detail with its training and inference processes.

4.1 Overview

Figure 2 illustrates the operation pipeline of `GLMTest` framework, which seamlessly integrates code structural information with code textual information using a GNN and an LLM, as follows. **(1)** The GNN extracts the code structural information of the targeted execution branch. Unlike prior graph–LLMs, our GNN module is optimized along with the LLM to induce structural embeddings aligned with targeted branches. **(2)** The LLM combines the structural embeddings with the text embedding of the instruction to generate test cases executing targeted branches. By incorporating both structural and textual information, the LLM learns branch-oriented representations more effectively, steering the model generation toward relevant execution paths.

First, the framework extracts a CPG G that captures the control-flow and data-dependency relationships of program S. Then, for a targeted execution branch b, it induces a specific set of nodes $V_b \subseteq V$ in G that are components related to execution statements in b (based on their location in S). We represent this set as a branch mask $m \in \{0,1\}^{|V|}$, where $m_i = 1$ if $i \in V_b$; otherwise, $m_i = 0$. Then, the GNN module f_g takes the CPG G and the branch mask m as input, and derives branch-aware structural embeddings that capture the structural dependencies of b.

Second, `GLMTest` adopts a text-based instruction prompt that specifies the testing objective and format, providing the LLM with high-level guidance on constructing effective test cases to exercise the targeted branch. The prompt is tokenized and encoded into textual embeddings. Then, the structural embeddings are concatenated with the textual embeddings, which are passed to the LLM module f_{lm} to generate executable test cases executing b.

We train `GLMTest` end-to-end on a high-quality dataset curated from real-world projects with human-written test cases. For program S, we extract its human-written test cases and their associated execution branches. Then, f_{lm} and f_g are jointly trained with a supervised learning method to generate test cases from the instruction prompt text embedding and the branch-aware structural embedding. `GLMTest`'s flexible structure allows both

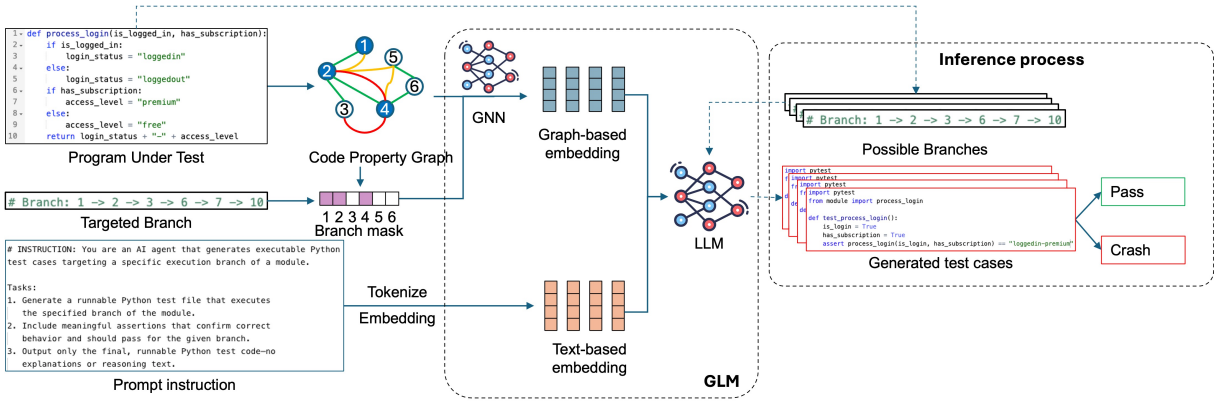

Figure 2: GLMTest pipeline.

f_{lm} and f_g to simultaneously train under conventional gradient-based optimizers, optimizing the same objective for targeted test case generation.

At inference time, developers can specify execution branches to test, and GLMTest generates test cases explicitly aimed at exercising those branches. In practice, developers can specify the execution locations (e.g., code lines, code blocks, or functions) from which GLMTest will automatically extract and derive execution branches. In addition, GLMTest can adapt to coverage-driven settings by enumerating feasible branches that are detectable by static analyzers and generating test cases across them. The resulting test cases are executed on the program, and their outcomes are used assess the program's execution (Song et al., 2019).

4.2 Model Structure of GLMTest

Let us describe the graph language–modeling module, the core component of GLMTest, to integrate structural and textual information from the program S for test case generation.

Execution Branch Embeddings. We first introduce the GNN module of GLMTest, which extracts branch-aware structural embeddings from the CPG G. We employ a K-layer heterogeneous GNN (Schlichtkrull et al., 2018) to capture different types of relations among nodes in G. Each layer $k \in [1, K]$ takes the node embeddings h_v^{k-1} for $v \in V$ from the previous layer $k-1$ and updates them as follows:

$$h_{N_j(v)}^k = \mathrm{AGG}\left(h_v^{k-1} \cup \{h_u^{k-1} : u \in N_j(v)\}\right),$$
$$h_{v,j}^k = \sigma\left(h_{N_j(v)}^k, W_j^k\right),$$

where $N_j(v)$ is a set of neighborhood nodes of v under relation type j with edge set E_j, $h_v^0 = x_v$ is the initial node feature, $\mathrm{AGG}(\cdot)$ is an aggregation function, W_j^k is the trainable parameter matrix of layer k for relation j, and $\sigma(\cdot)$ is a message-passing function (e.g., graph attention (Veličković et al., 2017) or GraphSAGE (Hamilton et al., 2017)).

This heterogeneous GNN propagates information along different relations in the CPG so that each node embedding captures its local program structural information. Also, the GNN backbone is modular, allowing GLMTest to benefit from future advanced GNN structures.

At the last layer K, the GNN component aggregates relation-specific embeddings into an overall embedding using a pooling function $\mathtt{pool}(\cdot)$, e.g., a summation or average pooling operator, as follows: $h_v^K = \mathtt{pool}\left(\{h_{v,j}^K\}_{j=1}^r\right)$. This step yields a unified embedding $h_v^K \in \mathbb{R}^{d_h}$ integrating information propagated through all edge relations r, where d_h is the hidden dimension. Then, the embeddings of the targeted branch b are derived by stacking the set of node embeddings related to the targeted branch b: $e_b = \mathtt{stack}\left(\{h_v^K\}_{v \in V_b}\right) \in \mathbb{R}^{|V_b| \times d_h}$. This set of branch embeddings encodes the structural context of all nodes participating in the targeted branch, containing fine-grained information along the execution path.

LLM Module. GLMTest adopts a prompt template tailored to the TestGenEval benchmark (Jain et al.) (Figure 7, Appendix D) explicitly defining the model's role and the prompt's inputs, and it constrains the output to a runnable and valid test case. The prompt's inputs include: **(i)** the program's source code, **(ii)** the execution-branch information (line executed), **(iii)** the importable program's path, **(iv)** the branch embeddings, and **(v)** a code snippet showing how to import the program.

To combine the structural embeddings with textual embeddings, we introduce a *graph token*

<|graph_pad|>, which is included in place of item (iv) as a placeholder for the branch embeddings. The branch embeddings e_b are then integrated by replacing the embeddings of the graph tokens with e_b, yielding an input embeddings e_{inp}, which is forwarded through the LLM f_{lm} to produce the token-level logits for next-token prediction. Thus, the structural signal influences all subsequent decoding steps, guiding the model toward generating test cases that exercise the targeted branch.

4.3 Training

Data Curation. Since no existing dataset is tailored to branch-targeted test case generation, we curate supervision signals directly from real-world projects and their developer-written test suites. For each program S, we first collect its existing test suite τ written by developers and decompose it into individual test cases $\{t_i\}_{i=1}^{|\tau|}$. We process each test case t_i to achieve high-quality requirements by removing unnecessary imports and dead code from each test set, ensuring t_i focuses only on the targeted branch and reducing hallucination.

We then execute each test case t_i and record the corresponding branch $\hat{b}_i = \texttt{Exec}_s(t_i)$ executed in S. This branch information is used to build the input prompt and CPG-based structural features, while the original test case serves as the ground-truth target output. This automatic procedure yields realistic (program, branch, test case) triples aligned with the training objective in Eq. (1). Training on this dataset guides the model to generate accurate test cases for the targeted execution branch, resulting in executable, branch-aware test cases.

For each executed test case, we construct a training sample by extracting the CPG G of S, deriving the branch mask m_i associated with b_i, and instantiating the corresponding instruction prompt p_i, yielding a dataset $D = \{(G, p_i, m_i, t_i)\}_{i=1}^{|D|}$. We release this branch-annotated dataset to support and encourage future research on structure-aware, branch-targeted test case generation.

Training Objectives. The GLMTest model is then trained to optimize the following objective:

$$\hat{t}_i = f_{\text{lm}}\big(e_{p_i}, f_g(G, m_i)\big), \tag{2}$$

$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{|D|} \ell(t_i, \hat{t}_i) + \lambda\|\theta\|_2, \tag{3}$$

where θ denotes all model parameters (including those of f_g and f_{lm}), λ is an ℓ_2 regularization coefficient, and $\ell(\cdot,\cdot)$ is the token-level training loss (e.g., cross-entropy).

Because the branch embedding is concatenated into the embedding e_{p_i}, this operation remains fully differentiable, allowing gradients to backpropagate through the GNN f_g. As a result, the entire GLMTest pipeline can be trained end-to-end via gradient-based optimization (e.g., Adam). In practice, this pipeline can be instantiated under different training paradigms, such as supervised fine-tuning (SFT) on developer-written test cases or reinforcement learning from human feedback (RLHF) (Patil and Gudivada, 2024) to further align generations with preferred testing behaviors. In our experiments, we focus on SFT due to its stability for large-scale training and leave RLHF-based refinement for future work.

5 Experimental Results

We conduct an extensive experiment to evaluate GLMTest around three research questions:

RQ1: Can GLMTest effectively generate test cases that exercise a targeted branch compared to state-of-the-art LLMs?

RQ2: Does GLMTest generate high-quality test suites that achieve competitive coverage?

RQ3: What is the contribution of GLMTest's components to its overall performance?

5.1 Experiment Setup

Datasets. We base our experiments on the TestGenEval dataset (Jain et al.), a large-scale dataset for evaluating unit test case generation and completion. TestGenEval is constructed from SWEBench and comprises 68,647 test cases paired with 1,210 modules with executable Docker environments. In our setting, we treat individual Python modules within a repository as programs under test, and the associated developer-written test cases as ground-truth test cases for these programs. We decompose each test cases and spurious dependencies. We then execute each test case and collect branch information. Each test case and its associated set of executed branches form a data point, yielding a triplet {program, branch, test case} as described in Section 4.3. Our processing results in 40,868 data points illustrated in Table 2 (Appendix D). We reserve 1,344 instances for evaluation, sampled uniformly across projects, ensuring that programs in the test set do not appear in the training set. Full details of our dataset are in Appendix A.1.

Implementation. To construct a CPG, we use Joern (Yamaguchi et al., 2014) for each program, and represent each node with a 772-dimensional feature vector obtained by concatenating a 768-dimensional `codet5p-110m-embedding` code embedding with a 4-dimensional encoding of categorical attributes (node type, order, and location). For branch masks, we build a binary branch mask by aligning line ranges with nodes. If no node aligns with a branch, we fall back to a special *"not available"* structural input. This design provides an interpretable mapping from dynamic execution to static structure while remaining compatible with standard coverage tools. We employ `GLMTest` instantiated with a 3-layer graph attention network (GAT) using 8 attention heads per layer and Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) as the backbone LLM. Full details of our implementation can be found in Appendix A.2.

Metrics. We evaluate `GLMTest` using three complementary metrics: (i) **`Pass@1`** (Jain et al.) reflects the basic *functional correctness* and executability of the generated test cases; (ii) **Branch Coverage** (`BranchCov`) (Wang et al., 2024) measures how many feasible execution branches we can explore, and thus reflects the *"testing utility"* of the generated suite; (iii) **Branch Accuracy** (`BranchAcc`) measures the success in executing the targeted execution branches; and (iv) **Branch Overlap** (`BranchOverlap`) measures the percentage of targeted branches that are covered by the generated test cases. It is worth noting that we modified the TestGenEval (Jain et al.) pipeline so that branch coverage is computed even if a generated test case fails due to incorrect assertions, and any executed branch is still recorded for coverage statistics.

`BranchAcc` captures whether the ground-truth targeted branch b_i is exercised by the generated test (with executed branch set $\hat{b}_i$), and `BranchOverlap` measures how much of b_i is actually covered, defined on a dataset D:

$$\texttt{BranchAcc} = \frac{1}{|D|}\sum_{i=1}^{|D|}\mathbb{I}[b_i \in \hat{b}_i],$$

$$\texttt{BranchOverlap} = \frac{1}{|D|}\sum_{i=1}^{|D|}\frac{|b_i \cap \hat{b}_i|}{|b_i|},$$

where $|b_i|$ is the number of statements in b_i and $|D|$ is the size of the dataset D.

As in TestGenEval (Jain et al.), `Pass@1` measures the percentage of test cases in the generated test suite that pass when run on the program. `BranchCov` measures how much branch coverage we obtain from these generated test cases. Specifically, for each program, we run the subset of generated test cases whose execution and assertions succeed, compute branch coverage, and report the average value across the programs.

Baselines. There is a growing body of work on LLM-based test case generation (Harman et al., 2025; Ryan et al., 2024b), among which only ASTER (Pan et al., 2025) and CodaMOSA (Lemieux et al., 2023) directly target Python test case generation. However, both mechanisms are tightly coupled to Pynguin and project-specific import configurations, requiring modules to be directly importable from the local filesystem. This setting is incompatible with the containerized, repository-level setup of the TestGenEval dataset, where programs are executed in pre-built Docker environments. Our efforts to adapt and reproduce these methods in TestGenEval ultimately proved inapplicable to TestGenEval's containerized setting. Therefore, we focus on two strong and controllable LLM-based baselines that are fully compatible with TestGenEval. (i) **Prompt Engineering** (PE): we follow the TestGenEval protocol and prompt templates to query state-of-the-art commercial LLMs (Claude-Sonnet-4.5 and GPT-4o-mini). (ii) **Fine-tuning** (FT): we fine-tune the same backbone LLM used in `GLMTest` on the {program, targeted branch, test case} triples, but remove the GNN component and feed only textual instruction. We use the same LLM as `GLMTest` without the GNN component and mark all branch embeddings as *Not Available*, so the model receives only textual inputs. More details are in Appendix A.3

5.2 RQ1: Can `GLMTest` effectively generate test cases that exercise a targeted branch compared to state-of-the-art LLMs?

Compared to the prompt-engineering (PE) baselines built on Claude-Sonnet-4.5 and GPT-4o-mini, `GLMTest` substantially improves the ability to exercise the targeted branches (Figure 3): overall branch accuracy increases from 0.274 (GPT-4o-mini) and 0.292 (Claude-Sonnet-4.5) to 0.502 (`GLMTest`), registering a relative performance improvement of 71.9%. It is worth noting that `GLMTest` uses a small, open-source model (Qwen2.5-Coder-7B-Instruct) augmented with our branch-structured conditioning rather than relying on massively scaled LLMs such as Claude-Sonnet-

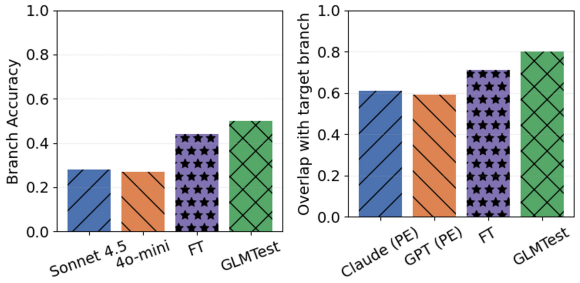

Figure 3: Branch accuracy and branch overlap with the targeted branches of GLMTest and baselines.

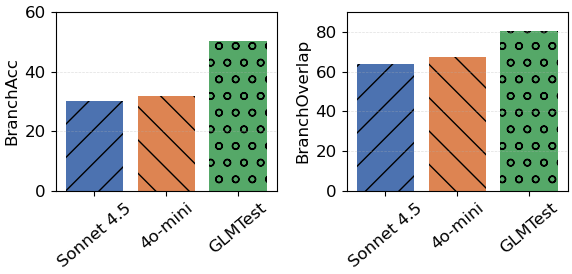

Figure 5: Branch accuracy and branch overlap with the targeted branches of GLMTest vs. RAG augmentation baselines.

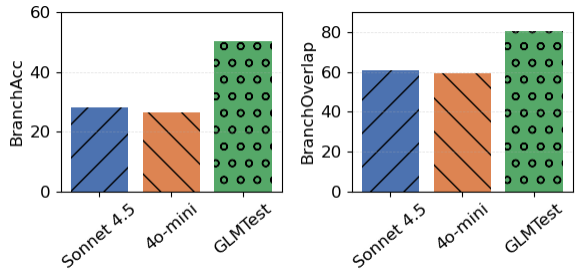

Figure 4: Branch accuracy and branch overlap with the targeted branches of GLMTest vs. baselines with execution feedback.

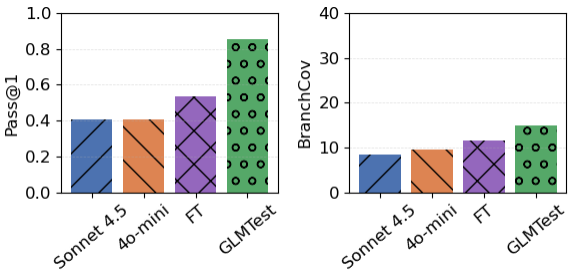

Figure 6: Pass@1 and BranchCov when using branch-targeted inference.

4.5 and GPT-4o-mini. Similar results are observed for BranchOverlap. Specifically, BranchOverlap increases on average from 0.615 of the PE baselines to 0.794 of GLMTest, indicating that GLMTest is more likely and more consistent to reach the targeted branch across tasks. Per-repository results mirror this overall performance. For instance, on django repository, compared with test cases generated by Claude-Sonnet-4.5, branch accuracy improves from 0.46 to 0.68 with GLMTest, and the fraction of targeted branches covered by the generated test cases increases from 0.63 to 0.91.

Finally, comparing GLMTest with FT isolates the effect of the GNN component. GLMTest improves BranchAcc from 0.44 (FT) to 0.50 and BranchOverlap from 0.71 to 0.80. Similar patterns appear in complex repositories, such as xarray, where GLMTest improves BranchAcc from 0.00 (FT) to 0.61 and BranchOverlap from 0.15 to 0.85, suggesting that code graph-based structural embeddings provide a rich signal that helps the model localize and execute the targeted execution branches.

Advanced Prompting Techniques. We compare GLMTest with an execution feedback baseline under the same branch-targeted evaluation protocol (Figure 4). Specifically, we provide the generated test case and its associated execution branch, and ask the models to revise their outputs accordingly. Under this setting, 4o-mini and Sonnet-4.5 achieve 26.5% and 28.5% BranchAcc with 59.5% and 60.2% BranchOverlap, respectively, which are lower than GLMTest's 50.2% BranchAcc and 80.2% BranchOverlap. These results indicate that iterative execution feedback, while providing some guidance, remains significantly less effective than explicit structural conditioning at reliably satisfying branch-specific execution constraints, highlighting GLMTest's advantage.

In addition, we compare GLMTest with a retrieval-augmented generation (RAG) baseline (Figure 5), constructing the retrieval corpus from the GLMTest training data. Specifically, for each training instance, we encode the prompt using OpenAI's text-embedding-3-small. At inference time, the RAG baseline retrieves the top-3 most similar samples via cosine similarity and provides them, along with their associated human-written test cases, as in-context examples. The baseline is then prompted to generate test cases targeting the specified branches. Under this setting, 4o-mini and Sonnet-4.5 achieve 31.8% and 30.1% BranchAcc with 67.5% and 64.0% BranchOverlap, respectively. These results further confirm that in-context retrieval augmentation remains insufficient for re-

liably satisfying branch-specific execution constraints compared to the explicit structural conditioning employed by GLMTest.

5.3 RQ2: Does GLMTest generate high-quality test suites that achieve competitive coverage?

We evaluate the test suite's quality and coverage under the GLMTest inference procedure. We use coverage.py to enumerate feasible execution branches. Then, for each mechanism, we generate one test per branch and aggregate the resulting test cases into a single test suite. We consider the number of processed branches at $\delta = 1,000$, which is sufficient to cover *all* branches for 41 out of 42 modules in the test set, providing ample branch-level coverage while keeping generation cost manageable. Then, we do the comparision under this shared pipeline in terms of Pass@1 and BranchCov.

In this branch-targeted setting (Figure 6), GLMTest attains substantially higher Pass@1 than the prompt-engineering baselines built on Claude-Sonnet-4.5 and GPT-4o-mini (~ 0.85 vs. 0.41), indicating that GLMTest's test suites are substantially higher quality and more reliably executable. Also, GLMTest achieves the highest BranchCov, which strengthens its superior branch accuracy, indicating that a higher number of the targeted execution branches are actually exercised.

In addition, we consider a complementary setting in which GLMTest follows its inference pipeline while Claude-Sonnet-4.5 and GPT-4o-mini are prompted with the original TestGenEval prompt template, which does *not* impose explicit branch targets and allows them to generate and explore freely. We include this setting for a fair comparison with the TestGenEval benchmark (Jain et al.), evaluating the functionality of generated test cases. In this setting, GLMTest still achieves the highest Pass@1 (i.e., 0.85 vs. 0.71 and 0.67 for Claude-Sonnet-4.5 and GPT-4o-mini, correspondingly), indicating that GLMTest consistently produces more reliable and executable test suites.

5.4 RQ3: Ablation studies

We conduct extensive ablation experiments to shed light on understanding the effects of each component of GLMTest on its performance.

GNN Structures. We vary the GNN structure while keeping the backbone LLM and training data fixed, comparing our default GAT encoder against a GraphSAGE-based alternative. This experiment evaluates the effect of the message-passing architecture on the quality and contribution of the branch embeddings. Comparing GLMTest with its GraphSAGE-based alternative (Table 1) indicates that replacing the default GAT encoder with GraphSAGE leads to a drop in branch accuracy from 0.502 to 0.465, suggesting that the multi-head attention mechanism of GAT is better suited to capturing the heterogeneity in the code graphs.

Factor	Model Variant	BranchAcc ↑
GNN structure	GAT	**0.502**
	SAGE	0.465
	None (FT)	0.442
Branch emb	Node emb	**0.502**
	Graph emb	0.399

Table 1: Ablation on GNN structure and branch embedding mechanism GLMTest.

Aggregation Mechanisms. We examine how the branch embedding is constructed by changing the aggregation mechanism, as follows: *(1) node-emb:* the embeddings of the nodes related to the targeted branch are concatenated; and *(2) graph-emb*: node embeddings are first pooled (mean operator) into a single branch vector before being injected into the LLM. As in Table 1, the node-level masking variant, which exposes all masked node embeddings directly to the LLM, achieves 0.502 branch accuracy, whereas pooling these nodes into a single branch vector reduces performance to 0.399. This indicates that preserving fine-grained structural information at the node level is vital for precise branch targeting, and motivates our choice of GAT with node-level masking as the default configuration for GLMTest.

Model Size. To assess the impact of model scale, we evaluate a larger backbone, Qwen2.5-Coder-14B-Instruct, under the same GLMTest training and inference pipeline. The 14B model achieves 49.3% BranchAcc and 17.63% BranchCov, compared to 50.2% and 16.91% for our default 7B model. These results suggest that explicit structural conditioning already provides strong branch-targeted reasoning at a moderate scale, while parameter scaling primarily benefits coverage breadth by enabling more diverse exploration of execution behaviors.

Training Cost. We train GLMTest for $2{,}048$ optimization steps with a per-device batch size of 8 and gradient accumulation of 32, using a maximum sequence length of $8{,}192$. To optimize training speed, we leverage LoRA with rank 8 and Deep-

Speed on 4× A100 GB GPUs. Each GPU occupies only 50GB of memory, runs for roughly 48 wall-clock hours (about 192 GPU-hours in total), and processes approximately 5.37×10^8 tokens. This fine-tuning budget, i.e., $119.56 on Vast.ai[2], is modest, making it easy to adopt in practice.

6 Discussion

Practical Use Cases. GLMTest is well suited for practical use cases such as security analysis, where it can target high-risk code paths (e.g., input validation flagged by static analyzers) by generating concrete test suites that exercise these regions. More generally, GLMTest integrates naturally into fuzzing pipelines, supplying high-quality seeds to coverage-guided or mutation-based fuzzers and improving analysis depth and efficiency.

Working with Other Languages. Although our experiments focus on Python, adapting GLMTest to other languages is straightforward. Joern already supports code property graph extraction for multiple languages (e.g., C/C++), enabling the GNN component to operate without architectural changes, and modern code LLMs are multilingual. The primary challenge lies in data curation, which requires running test suites in language-specific environments and recording the branches exercised.

Mitigating the Limitation of CPG. While GLMTest relies on static CPG extraction, syntactically present branches may be unreachable due to dead code or unsatisfiable runtime constraints. As a mitigation, dynamic analysis tools such as PyAnalyzer (Jin et al., 2024) can resolve library dependencies on demand and validate branch feasibility, and combining them with retrieval-based context expansion could further improve runtime condition resolution, which is a promising direction for future work.

7 Conclusion

We presented GLMTest, a novel graph-enhanced language modeling framework that treats feasible execution branches as explicit test-generation targets. By integrating structural and textual information, GLMTest enables structure-aware test case generation. Our experimental results show that GLMTest built on the Qwen2.5-Coder-7B-Instruct model achieves high branch accuracy and executability, while achieving competitive branch coverage compared with state-of-the-art commercialized LLMs (Claude-Sonnet-4.5 and GPT-4o-mini), highlighting the advantages of GLMTest.

Limitations

While GLMTest improves branch-targeted test case generation on our benchmark, it has several limitations. First, our current model is trained on a relatively small set of projects from TestGenEval and does not yet demonstrate strong cross-project generalization. Extending training to a broader and more diverse corpus of repositories is a natural next step. Second, the branch-targeted inference pipeline can become expensive on very large, highly modular systems with thousands of feasible branches. In such settings, applying GLMTest to every branch is impractical. This is a problem for all testing approaches - not just GLMTest- and the method is better viewed as a targeted tool for a subset of critical branches. This limitation also suggests future work on principled branch prioritization, for example, by combining GLMTest with a static security risk detection mechanism (Lekssays et al., 2025; Li et al., 2025).

Acknowledgments

This research was supported by the National Science Foundation (NSF) under Grant No. CNS 2237328 and DGE 2043104, and the Grace Hopper AI Research Institute.

References

Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2024. A3test: Assertion-augmented automated test case generation. *Information and Software Technology*, 176:107565.

Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing*. Cambridge University Press.

Merve Astekin, Max Hort, and Leon Moonen. 2024. An exploratory study on how non-determinism in large language models affects log parsing. In *Proceedings of the ACM/IEEE 2nd International Workshop on Interpretability, Robustness, and Benchmarking in Neural Software Engineering*, pages 13–18.

Mohammad Baqar and Rajat Khanda. 2025. The future of software testing: Ai–powered test case generation and validation. In *Intelligent Computing-Proceedings of the Computing Conference*, pages 276–300. Springer.

Dhaya Sindhu Battina. 2019. Artificial intelligence in software test automation: A systematic literature

[2] https://vast.ai/pricing

review. *International Journal of Emerging Technologies and Innovative Research (www. jetir. org| UGC and issn Approved), ISSN*, pages 2349–5162.

Tristan Bilot, Nour El Madhoun, Khaldoun Al Agha, and Anis Zouaoui. 2024. A survey on malware detection with graph representation learning. *ACM Comput. Surv.*, 56(11).

Matteo Brunetto, Giovanni Denaro, Leonardo Mariani, and Mauro Pezzè. 2021. On introducing automatic test case generation in practice: A success story and lessons learned. *Journal of Systems and Software*, 176:110933.

Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 572–576, New York, NY, USA. Association for Computing Machinery.

Zeqi Chen, Zhaoyang Chu, Yi Gui, Feng Guo, Yao Wan, and Chuan Shi. 2025. Bridging code graphs and large language models for better code understanding. *arXiv preprint arXiv:2512.07666*.

Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, 171:107468.

Xiaotao Feng, Xiaogang Zhu, Kun Hu, Jincheng Wang, Yingjie Cao, Guang Gong, and Jianfeng Pan. 2025. Fuzzing: Randomness? reasoning! efficient directed fuzzing via large language models. *arXiv preprint arXiv:2507.22065*.

Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30.

Mark Harman, Jillian Ritchey, Inna Harper, Shubho Sengupta, Ke Mao, Abhishek Gulati, Christopher Foster, and Hervé Robert. 2025. Mutation-guided llm-based test generation at meta. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 180–191.

Linghan Huang, Peizhou Zhao, Lei Ma, and Huaming Chen. 2025. On the challenges of fuzzing techniques via large language models. In *2025 IEEE International Conference on Software Services Engineering (SSE)*, pages 162–171. IEEE.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Kush Jain, Gabriel Synnaeve, and Baptiste Roziere. Testgeneval: A real world unit test generation and test completion benchmark. In *The Thirteenth International Conference on Learning Representations*.

Wuxia Jin, Shuo Xu, Dawei Chen, Jiajun He, Dinghong Zhong, Ming Fan, Hongxu Chen, Huijia Zhang, and Ting Liu. 2024. Pyanalyzer: An effective and practical approach for dependency extraction from python code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12.

KPMG. 2024. Software testing: Market and insights report 2024. https://assets.kpmg.com/content/dam/kpmgsites/uk/pdf/2024/08/software-testing-market-and-insights-report.pdf. Accessed: 2025-12-01.

Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. 2025. {LLMxCPG}:{Context-Aware} vulnerability detection through code property {Graph-Guided} large language models. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 489–507.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931.

Youpeng Li, Fuxun Yu, and Xinda Wang. 2025. Vulpo: Context-aware vulnerability detection via on-policy llm optimization. *arXiv preprint arXiv:2511.11896*.

Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218.

Ruitong Liu, Yanbin Wang, Haitao Xu, Jianguo Sun, Fan Zhang, Peiyue Li, and Zhenhao Guo. 2025a. Vul-lmgnns: Fusing language models and online-distilled graph neural networks for code vulnerability detection. *Information Fusion*, 115:102748.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. 2025b. Codexgraph: Bridging large language models and code repositories via code graph databases. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 142–160.

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer Publishing Company, Incorporated.

Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. 2025. Aster: Natural and multi-language unit test generation with llms. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 413–424. IEEE.

Md Rizwan Parvez, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2018. Building language models for text with named entities. In *Proceedings of the*

56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 2373–2383, Melbourne, Australia. Association for Computational Linguistics.

Rajvardhan Patil and Venkat Gudivada. 2024. A review of current trends, techniques, and challenges in large language models (llms). *Applied Sciences*, 14(5):2074.

Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J. Hellendoorn. 2024. Cat-lm training language models on aligned code and tests. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ASE '23, page 409–420. IEEE Press.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024a. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proc. ACM Softw. Eng.*, 1(FSE).

Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024b. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering*, 1(FSE):951–971.

Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *European semantic web conference*, pages 593–607. Springer.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105.

Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 2024. Using large language models to generate junit tests: An empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, page 313–322, New York, NY, USA. Association for Computing Machinery.

Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE.

Yifan Song, Guoyin Wang, Sujian Li, and Bill Yuchen Lin. 2025. The good, the bad, and the greedy: Evaluation of llms should not ignore non-determinism. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4195–4206.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit test case generation with transformers and focal context. *Preprint*, arXiv:2009.05617.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903*.

Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4):911–936.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025. TestEval: Benchmarking large language models for test case generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3547–3562, Albuquerque, New Mexico. Association for Computational Linguistics.

Felix Weissberg, Jonas Möller, Tom Ganz, Erik Imgrund, Lukas Pirch, Lukas Seidel, Moritz Schloegel, Thorsten Eisenhofer, and Konrad Rieck. 2024. Sok: Where to fuzz? assessing target selection methods in directed fuzzing. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 1539–1553.

Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE symposium on security and privacy*, pages 590–604. IEEE.

Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 141–151, New York, NY, USA. Association for Computing Machinery.

Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36.

A Experimental details

A.1 Dataset processing details

From each project in TestGenEval, we first decompose the available test suites into individual test cases. For every test case, we statically remove unused imports to simplify the context and reduce opportunities for the model to hallucinate spurious dependencies. We then execute each test case inside its official Docker environment and collect *branch* information using `coverage.py`[3] in branch-coverage mode. Test cases that fail due to environment issues, timeouts exceeding 60 seconds per Jain et al. (Jain et al.) or nondeterministic behavior, are discarded, and we run each remaining test once to obtain a stable branch set. Each qualified test case and its associated set of executed branches form a data point, yielding (program, branch, test case) triples as described in Section 4.3. Projects that yield fewer than 15 valid triples after filtering are removed (4 of the 11 repositories are discarded), as they provide little signal and complicate stratified sampling. After preprocessing, we obtain 7 projects, 45,831 unique triples (program, branch, test case) (see Table 2). For evaluation, we reserve 1,489 test instances, sampled uniformly across the remaining projects to avoid project skew, and use the rest for training and validation; within this split, projects are shared across splits, but individual test cases are disjoint, so our results primarily measure generalization to unseen test cases within the same set of projects.

A.2 Implementation

Additional details of node's features. Each CPG node is annotated with source-location metadata (file path, start line, end line) and a set of categorical attributes (e.g., syntactic type, role in the AST or control/data flow). For node text features, we encode the textual content associated with each node (code snippet and identifier context, excluding comments and docstrings) using the `Salesforce/codet5p-110m-embedding` pretrained model. We use the CodeT5p encoder to obtain a 768-dimensional embedding. We then concatenate this code embedding with a 4-dimensional label-encoded vector of categorical node attributes to obtain a 772-dimensional per-node feature vector, thereby combining rich pre-trained code semantics with lightweight structural metadata.

[3] `https://coverage.readthedocs.io/en/7.13.0/`

Additional details of branch mask construction. To construct branch masks, for each executed branch, we obtain the corresponding set of executed line numbers and align them with CPG nodes via their source-location intervals: a node is marked as relevant (mask value 1) if its line range intersects the executed line set, and irrelevant (mask value 0) otherwise. In rare cases where no CPG node aligns with an executed branch, we set the structural embedding and the prompt input to *"Not available"*. This line-level alignment provides a direct, interpretable mapping from dynamic execution to static structure, enabling `GLMTest` to focus on subgraphs along the targeted execution path while remaining compatible with standard coverage tooling.

A.3 Baseline settings

We compare `GLMTest` against prior work on automated test case generation and two dataset-compatible LLM baselines. Recent systems include ASTER (Pan et al., 2025), CodaMOSA (Lemieux et al., 2023), ACH (Harman et al., 2025), SymPrompt (Ryan et al., 2024b). Among these, only ASTER and CodaMOSA directly target Python unit tests, but both are implemented as Pynguin-based pipelines that assume locally importable modules and direct filesystem access to the project under test. In contrast, TestGenEval executes each repository inside an isolated Docker container with its own entrypoint and dynamically configured `PYTHONPATH`, and does not expose the Pynguin-style project orchestration interface. In our attempts to run ASTER and CodaMOSA on TestGenEval, we were unable to make their Pynguin-based harness discover and import the correct modules inside the official containers without substantial re-engineering of their toolchains, which we consider out of scope for this work.[4]

Within these constraints, we use two reproducible LLM baselines that share the same decoding budget and evaluation protocol as `GLMTest`. (i) *Prompt-only LLM (PE).* Following the TestGenEval setting, we query LLMs with a fixed prompt template provided by TestGenEval that includes the program source and a textual description of the target branch (its line range and correct order of lines executed as in Figure 1), but no CPG-derived features. For each instance, we generate

[4] We therefore report no ASTER/CodaMOSA numbers on TestGenEval; our code release will document the incompatibility and configuration attempts.

a single test case ($k = 1$) using temperature 0.2 and greedy decoding strategy, so that Pass@1 is directly comparable across models. (ii) *Text-only fine-tuning (FT).* We fine-tune the same backbone LLM as `GLMTest` on our (program, branch, test case) triples, but remove the GNN and represent the branch set purely as text (a serialized list of executed line ranges) concatenated with the program source and instruction prompt, capped at 8192 tokens. FT therefore has access to branch information only through this textual description, without any explicit graph structure or relational context, providing a strong non-structural baseline that isolates the contribution of CPG-based conditioning in `GLMTest`.

B Related work

LLMs for test case generation. Recently, LLMs have been applied to software testing to produce readable, executable test suites and improve coverage (Tufano et al., 2021). Existing approaches generally fall into two categories: fine-tuning and prompt engineering. Fine-tuning methods train on curated code–test pairs to specialize LLMs for test case generation (Tufano et al., 2021; Alagarsamy et al., 2024; Rao et al., 2024), whereas prompt-based methods keep the LLM frozen and construct structured prompts from extracted program features (e.g., signatures, control-flow summaries) to guide coverage-oriented generation (Schäfer et al., 2024; Siddiq et al., 2024; Chen et al., 2024; Dakhel et al., 2024). These techniques have shown promising gains in global coverage, but they do not explicitly represent or optimize for specific execution branches, which limits their effectiveness in scenarios where developers or security analysts need to exercise particular high-risk paths.

Combining CPGs with LLMs. Recent works (Lekssays et al., 2025; Chen et al., 2025; Liu et al., 2025b) have begun to combine CPGs with LLMs for downstream code understanding and analysis tasks, typically treating the graph as a knowledge source to enrich the prompt or as a generic encoder whose outputs are consumed at the sequence level. Lekssays et al. (Lekssays et al., 2025) leverage the CPG to generate a code slice from the codebase, keeping relevant code lines, and prompt the LLMs with the code slice for vulnerability detection. Chen et al. (Chen et al., 2025) proposed a framework that incorporates CPG-derived node features into the LLM's forward pass to enhance code understanding. However, existing works usually extract code snippets guided by the graph without explicitly encoding the underlying structural relationships into branch-specific representations.

C Use of AI Assistants

In this work, we leverage the help of AI assistants to facilitate the work as follows. For the literature search, we use the Google Scholar Labs agent to find relevant works. However, all citations are manually checked and selected by the authors. To implement the project, we use Copilot, equipped with Claude-Sonnet-4.5, as a coding assistant to edit the code. Nevertheless, all experimental designs, algorithmic choices, and executions are conducted manually by the authors. For writing, we used GPT-5.2 as an assistant purely with the language of the paper. The problem formulation, technical contributions, and empirical analysis were conducted by the authors.

D Supplemental Results

Project	# Train	# Test	# Stars
`astropy`	2,714	6	5.0k
`django`	19,364	690	86.3k
`xarray`	1,615	23	4.0k
`pytest`	2,617	93	13.4k
`scikit-learn`	5,918	319	64.4k
`sympy`	8,640	213	14.2k
Total	40,868	1,344	–

Table 2: Number of training and test data points per repository after preprocessing, along with approximate GitHub star counts (as of late 2025).

```
# INSTRUCTION: You are an AI agent that
generates executable Python test cases
targeting a specific execution branch
of a module.

Inputs:
- Module source: source code of the
target module (Could be truncated to
related line only).
- Execution branch information: the
lines of the target module executed.
- Module path: a valid, importable path
from the PYTHONPATH directory.
- Code Property Graph (CPG) embeddings
(Optional): semantic and structural
information about the code elements
related to the branch.

Tasks:
1. Generate a runnable Python test file
that executes the specified branch of
the module.
2. Include meaningful assertions that
confirm correct behavior and should pass
for the given branch.
3. Output only the final, runnable
Python test code, no explanations
or reasoning text.

Requirements:
- All imports must be valid and
correspond to existing modules; do not
invent or hallucinate any packages.
- Use standard testing practices (unit-
test, pytest, or assert statements).
- Keep the code clear, minimal,
and maintainable.
---------------------------------------

# INPUTS:

## Module Source: <Input>

## Execution Branches Information
(Line to Line executed): <Input>

## Module Path: <Input>

## Code Property Graph
(CPG) Node Embeddings: <Input>

## Here's how to import the target
module: <Input>
```

Figure 7: Prompt template used by GLMTest to instruct the LLM to generate branch-targeted Python test cases.