SAD-Gen: Structure-Aware Unit Test Generation via Dual-Encoder Fusion and Latent Space Disentanglement

Jingqi Gao*, Haoran Yan, Kun Yang, Nan Liang, Wentao Qiu, Pengfei Ding, Jingan Chen, Qingqiang Wu†
School of Informatics, Xiamen University
Xiamen, China
30920241154565@stu.xmu.edu.cn, randy5240@stu.xmu.edu.cn, 36920241153270@stu.xmu.edu.cn,
liangnan@stu.xmu.edu.cn, wtqiu@stu.xmu.edu.cn, 30920241154564@stu.xmu.edu.cn,
chenjingan@stu.xmu.edu.cn, wuqq@stu.xmu.edu.cn

***Abstract*—Large Language Models (LLMs) have demonstrated impressive capabilities in automated unit test generation. However, they predominantly rely on textual co-occurrence patterns, often neglecting the explicit control flow and dependency structures inherent in source code. This limitation frequently results in test cases that suffer from logical hallucinations and insufficient boundary coverage. While Graph Neural Networks (GNNs) excel at modeling non-Euclidean code structures, effectively aligning structural features with the textual semantics of LLMs remains a significant challenge. To address these issues, we propose SAD-Gen, a Structure-Aware Disentangled Generation framework. First, we present a dual-modal encoding architecture that leverages a textual encoder to capture sequential semantics alongside Graph Attention Networks (GAT) for extracting structural features from Abstract Syntax Trees (ASTs). These representations are dynamically fused via an adaptive gated mechanism. Second, to maximize generation diversity, we propose a novel Latent Space Disentanglement Strategy that decouples the program representation into immutable Logic Latents and perturbable Data Latents. By injecting Gaussian noise into the data latent space during generation, our model actively explores diverse boundary conditions while strictly preserving the underlying program logic. Extensive experiments on the Methods2Test and MBPP datasets demonstrate that SAD-Gen consistently outperforms state-of-the-art baselines in terms of syntax pass rate, branch coverage, and mutation score, validating the effectiveness and robustness of our proposed approach.**

***Index Terms*—Unit Test Generation, Latent Space Disentanglement, Structure-Aware Modeling, Graph Neural Networks**

I. Introduction

Unit testing stands as a cornerstone in modern software development, serving as the first line of defense against software regression and functional defects. Writing high-quality unit tests manually, however, is a labor-intensive and error-prone process, consuming a substantial amount of development effort [1]. To mitigate this burden, Automated Unit Test Generation [2], [3]has garnered significant attention from both academia and industry. The primary goal is to automatically synthesize test cases that not only adhere to syntactic constraints but also achieve high code coverage to reveal potential vulnerabilities.

In recent years, LLMs trained on vast code corpora have revolutionized this field [4]–[6]. By modeling code as sequential text, LLMs can generate fluent and human-like test cases [7], [8]. However, a critical limitation persists: LLMs predominantly rely on statistical token co-occurrence probabilities, often neglecting the explicit control flow and rigid dependency structures inherent in source code. This textual bias frequently leads to logical hallucinations—such as invoking non-existent methods [9]–[11] or misinterpreting complex nested conditions. Although GNNs excel at capturing non-Euclidean structures [12]–[14], existing approaches often struggle to effectively align these structural features with the high-dimensional semantic spaces of LLMs, treating them as disjoint modalities rather than a unified representation [15].

Furthermore, even when generated tests are syntactically valid, they often suffer from low diversity [16]. LLMs tend to converge on average input patterns found in the training data, failing to explore edge cases or boundary conditions (like empty lists, extreme values) that are crucial for robust testing. This is largely because the generation process entangles the program logic (what to test) with the input data (how to test) in a chaotic latent space, making it difficult to precisely control the generation of diverse test inputs [17]–[19].

To address these challenges, we propose SAD-Gen, a novel framework that bridges the gap between semantic understanding and structural constraints. Unlike previous approaches that treat code structure and semantics as disjoint modalities, SAD-Gen unifies them to ensure both logical correctness and test coverage. The main contributions of this paper are summarized as follows:

- We propose a Dual-Modal Encoding Architecture that integrates a textual encoder (e.g., CodeT5+ [20]) for sequential semantics and GAT [21] for AST-based structural features. An adaptive gated mechanism is employed to dynamically fuse these representations, effectively mitigating logical hallucinations caused by textual bias.
- We introduce a Latent Space Disentanglement Strategy to enhance test diversity. By projecting code representations into orthogonal Logic Latents (immutable) and Data Latents (perturbable), we enable the model to actively ”imagine” diverse boundary conditions through noise injection without compromising logical correctness.
- We conduct extensive experiments on the Methods2Test

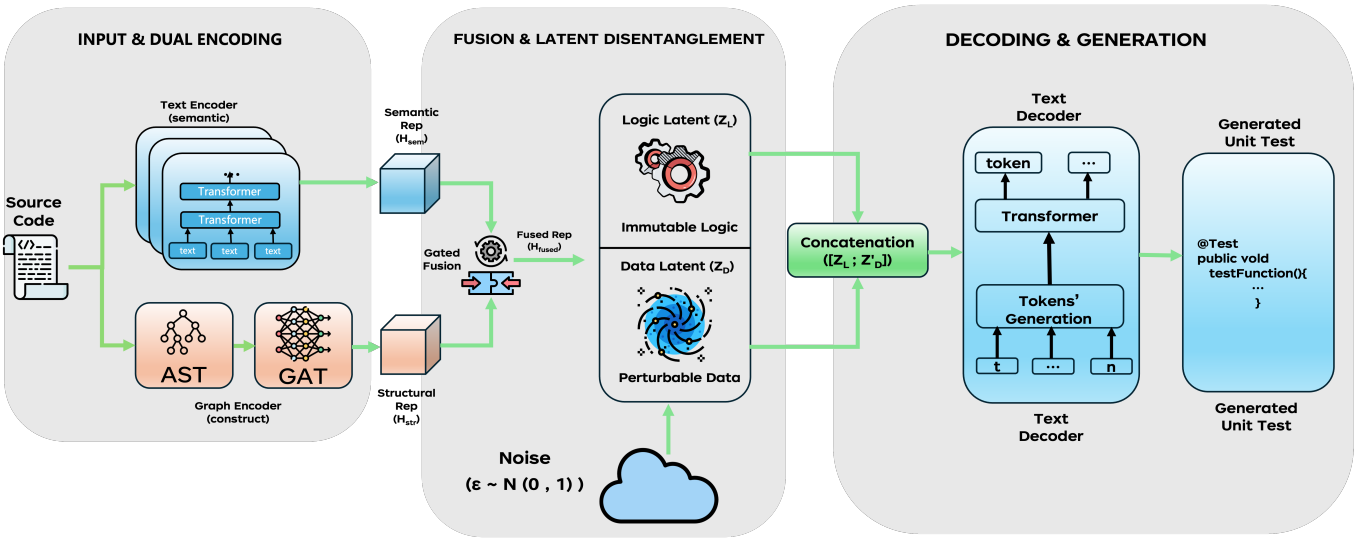

Fig. 1. Overall architecture of the SAD-Gen framework.

[22] and MBPP [23] datasets. The results demonstrate that SAD-Gen significantly outperforms state-of-the-art baselines, achieving substantial improvements in branch coverage and mutation score.

II. Method

A. Overview

The overall architecture of our proposed SAD-Gen framework is illustrated in Fig.1.The framework consists of three distinct stages: (1) Dual-Modal Encoding, which processes source code through parallel semantic and structural streams; (2) Latent Disentanglement, which projects the fused representation into orthogonal logic and data subspaces via a variational mechanism; and (3) Structure-Aware Decoding, which reconstructs high-coverage unit tests from the disentangled latent representations.

B. Input & Dual-Modal Encoding

As shown in the "Input & Dual Encoding" module of Fig.1, we represent the input source code C from two complementary perspectives: sequence and graph.

1) Semantic Encoding Stream: To capture the sequential semantics (e.g., identifiers, comments, and natural language cues), we employ a stack of Transformer [24] encoder layers (labeled "Text Encoder" in Fig.1). Given the tokenized code sequence $X = \{x_1, x_2, \ldots, x_n\}$, the encoder computes the contextualized representation $\mathbf{H}_{sem} \in \mathbb{R}^{n \times d}$ via multi-head self-attention:

$$\mathbf{H}_{sem} = \text{TransformerEncoder}(X) \tag{1}$$

where each layer applies the scaled dot-product attention mechanism followed by position-wise feed-forward networks.

2) Structural Encoding Stream: To model the rigorous syntactic constraints, we simultaneously parse the code into an AST and construct a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. We utilize a GAT (labeled "Graph Encoder" in Fig.1) to aggregate neighborhood information. The structural embedding for each node v_i at layer l is updated as:

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W}^{(l)} \mathbf{h}_j^{(l)}\right) \tag{2}$$

where α_{ij} is the attention coefficient derived from the node features. A global pooling operation is then applied to obtain the fixed-size structural representation $\mathbf{H}_{str} \in \mathbb{R}^d$.

C. Gated Fusion Mechanism

Directly concatenating heterogeneous features may lead to semantic misalignment. To address this, we introduce an Adaptive Gated Fusion module (visualized as the fusion gear in Fig.1). We compute a learnable gate vector $\mathbf{g}$ to dynamically weigh the contribution of each modality:

$$\mathbf{g} = \sigma\left(\mathbf{W}_f[\mathbf{H}_{sem} \| \mathbf{H}_{str}] + \mathbf{b}_f\right) \tag{3}$$

The fused representation $\mathbf{H}_{fused}$ is obtained by:

$$\mathbf{H}_{fused} = \mathbf{g} \odot \mathbf{H}_{sem} + (1 - \mathbf{g}) \odot \mathbf{H}_{str} \tag{4}$$

where $\odot$ denotes element-wise multiplication and $\|$ denotes concatenation.

D. Latent Space Disentanglement

This is the core innovation of SAD-Gen. As depicted in the "Fusion & Latent Disentanglement" panel, we map the fused representation into a variational latent space. To separate immutable program logic from variable data patterns, we explicitly disentangle the latent space into two orthogonal subspaces: Logic Latent ($\mathcal{Z}_L$) and Data Latent ($\mathcal{Z}_D$).

1) Probabilistic Projection: We model the posterior distributions $q_\phi(\mathbf{z}|C)$ using two separate inference networks. We assume multivariate Gaussian priors:

$$\mathbf{z}_L \sim \mathcal{N}(\boldsymbol{\mu}_L, \boldsymbol{\sigma}_L^2 \mathbf{I}) \quad \text{(Immutable Logic)}$$
$$\mathbf{z}_D \sim \mathcal{N}(\boldsymbol{\mu}_D, \boldsymbol{\sigma}_D^2 \mathbf{I}) \quad \text{(Perturbable Data)}$$

The parameters $(\boldsymbol{\mu}, \boldsymbol{\sigma})$ are learned from $\mathbf{H}_{fused}$. Using the reparameterization trick, we sample latent vectors [25]:

$$\mathbf{z}\_k = \boldsymbol{\mu}\_k + \boldsymbol{\sigma}\_k \odot \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \quad k \in \{L, D\} \quad (5)$$

2) Stochastic Data Perturbation: To encourage the generation of diverse boundary conditions, we introduce a noise injection mechanism. As shown in the "Noise" module of Fig.1, we inject Gaussian noise δ specifically into the data latent space during training:

$$\tilde{\mathbf{z}}_D = \mathbf{z}_D + \lambda \cdot \delta, \quad \delta \sim \mathcal{N}(0, \mathbf{I}) \quad (6)$$

This forces the decoder to reconstruct valid test logic ($\mathbf{z}_L$) even when the input data features ($\tilde{\mathbf{z}}_D$) are perturbed, thereby enhancing robustness.

E. Decoding & Generation

The final stage is the reconstruction of the unit test.

1) Latent Concatenation: We first concatenate the immutable logic latent and the perturbed data latent to form the final context vector (labeled "Concatenation" in Fig.1):

$$\mathbf{z}_{final} = [\mathbf{z}_L \| \tilde{\mathbf{z}}_D] \quad (7)$$

2) Text Decoding: The decoder (Transformer-based model) receives $\mathbf{z}_{final}$ and automatically generates the target unit test $Y = \{y_1, \ldots, y_m\}$. The probability of generating the next token y_t is:

$$P(y_t|y_{<t}, \mathbf{z}_{final}) = \text{softmax}(\mathbf{W}_{out}\mathbf{h}_t^{dec}) \quad (8)$$

where $\mathbf{h}_t^{dec}$ is the hidden state of the decoder at step t.

F. Training Objective

We train the entire framework end-to-end by minimizing a composite loss function $\mathcal{L}_{total}$, ensuring both generation quality and effective disentanglement.

1) Reconstruction Loss:

$$\mathcal{L}_{rec} = -\sum_{t=1}^{m} \log P(y_t|y_{<t}, \mathbf{z}_{final}) \quad (9)$$

2) KL Divergence Regularization: To align the posterior distributions with the standard normal prior:

$$\mathcal{L}_{KL} = \sum_{k \in \{L,D\}} D_{KL}(q_\phi(\mathbf{z}_k|C) \| \mathcal{N}(\mathbf{0}, \mathbf{I})) \quad (10)$$

3) Orthogonality Constraint: To prevent information leakage between $\mathcal{Z}_L$ and $\mathcal{Z}_D$, we enforce an orthogonality constraint on their mean vectors:

$$\mathcal{L}_{orth} = \left\| \frac{\boldsymbol{\mu}_L \cdot \boldsymbol{\mu}_D}{\|\boldsymbol{\mu}_L\|_2 \|\boldsymbol{\mu}_D\|_2} \right\|^2 \quad (11)$$

Total Loss:

$$\mathcal{L}_{total} = \mathcal{L}_{rec} + \beta \mathcal{L}_{KL} + \gamma \mathcal{L}_{orth}$$

III. EXPERIMENTS

A. Research Questions

To comprehensively evaluate the effectiveness of our proposed framework, we design our experiments to answer the following four research questions:

- **RQ1:** How does SAD-Gen perform in comparison to the representative benchmarks in all the evaluation metrics?
- **RQ2:** What is the contribution of each key component (i.e., Structural Encoding, Gated Fusion, and Latent Disentanglement) to the overall performance?
- **RQ3:** Can the proposed variational mechanism effectively disentangle the immutable program logic from the perturbable data patterns in the latent space?
- **RQ4:** How does the intensity of noise injection in the data latent space impact the diversity and boundary coverage of the generated test cases?

B. Datasets and Preprocessing

To evaluate the versatility and robustness of SAD-Gen across different programming paradigms and languages, we conducted experiments on two distinct datasets: Methods2Test and MBPP.

- **Methods2Test:** This is a large-scale dataset consisting of real-world Java methods and their corresponding JUnit test cases mined from open-source repositories. It represents complex, enterprise-level logic.
- **MBPP:** To assess the cross-language generalizability of our framework, we employ the Mostly Basic Python Problems (MBPP) dataset. Unlike Methods2Test, MBPP focuses on algorithmic problems and foundational programming logic.

To ensure data quality and task alignment, we applied rigorous preprocessing across both datasets. For Methods2Test, we filtered out noise by removing methods with fewer than 3 lines or syntax errors, and eliminated duplicate pairs to prevent data leakage. For MBPP, we adapted the original code synthesis benchmarks for unit test generation by treating the ground-truth solutions as input source (C) and the accompanying assertions as targets (T). Regarding structure extraction, we employed Tree-sitter to parse both Java and Python code into ASTs, pruning redundant nodes (e.g., comments) to simplify graph complexity. This unified parsing approach demonstrates that our structural encoding module is language-agnostic and capable of adapting to diverse syntactic structures, from Java's block-based syntax to Python's indentation rules.The final statistics of the dataset are summarized in Table I.

C. Baselines

To provide a comprehensive and rigorous evaluation, we compare SAD-Gen against four representative state-of-the-art baselines, covering a diverse spectrum of model scales (from 1.5B to 8B) and architectures (Dense, MoE, and Encoder-Decoder):

- **Qwen2.5-Coder-1.5B:** The current SOTA among small-sized open models, known for its exceptional coding performance despite its compact size.

TABLE I
STATISTICS OF EVALUATION DATASETS

Dataset Name	Language Source	Type Domain	Split Statistics		
			Train	*Valid*	*Test*
Methods2Test	Java	Real-world	78,450	8,920	9,800
MBPP	Python	Algorithmic	-	-	964
Total	-	-	**78,450**	**8920**	**10,764**

[a]Methods2Test is for training; MBPP is for zero-shot evaluation.

- **DeepSeek-Coder-V2-Lite:** A Mixture-of-Experts (MoE) model that demonstrates superior code reasoning capabilities, serving as a strong baseline for logic generation.
- **Llama 3-8B-Instruct:** A powerful general-purpose foundation model. Comparing with it highlights the advantages of our structure-aware specialized approach over larger generalist models.
- **CodeT5+:** The backbone of our framework. We include it to rigorously quantify the specific improvements contributed by our structural encoding and latent disentanglement modules.

D. Evaluation Metrics

We comprehensively evaluate the generated test cases using four standard metrics. We first measure Compilability to ensure syntactic correctness. To assess testing thoroughness, we report Line Coverage and Branch Coverage, prioritizing the latter as the primary indicator of the model's ability to navigate complex control flows. Furthermore, we calculate Mutation Score using the Pitest framework to quantify the tests' semantic robustness and their practical capability in detecting software faults.

E. Implementation Details

We implement SAD-Gen using PyTorch and employing CodeT5+ as the semantic backbone and a 3-layer GAT (hidden dimension of 768) for structural encoding. The model is trained on a single NVIDIA RTX A6000 GPU for 20 epochs using the AdamW optimizer, configured with an initial learning rate of $2e^{-5}$, a batch size of 32, and a linear warmup over the first 10% of training steps. Regarding hyperparameters, we set the dimensions of both logic and data latent subspaces ($\mathcal{Z}_L, \mathcal{Z}_D$) to 384, the noise injection coefficient λ to 0.4, and utilize beam search with a width of 5 for final test generation.

F. Main Results

1) RQ1: Effectiveness Comparison: Table II presents the comparative results on the Methods2Test dataset. We observe that SAD-Gen establishes a new SOTA performance in terms of structural coverage and fault detection capability. Specifically:

- **Coverage Advantage:** SAD-Gen achieves a Branch Coverage of 69.5%, significantly outperforming the massive general-purpose model Qwen2.5-Coder (63.5%) by +6.0%. This result highlights the limitation of purely data-driven LLMs: while they generate syntactically perfect code (achieving a higher Pass Rate of 84.5%), they often miss complex edge cases. Our GAT-based structural encoding effectively guides the generation to traverse deeper branches.
- **Mutation Score:** In terms of bug detection, SAD-Gen achieves a mutation score of 63.8%, surpassing the strongest baseline (Qwen2.5) by +6.3%. This indicates that the test cases generated by our framework are not only diverse but also semantically meaningful, capable of verifying critical logic paths.

2) RQ2: Ablation Study: To validate the contribution of each component, we compare SAD-Gen with three variants: (1) *w/o Structure* (removing the GAT encoder), (2) *w/o Disentanglement* (removing the dual-latent mechanism), and (3) *w/o Gated Fusion* (using simple concatenation). As shown in Table III:

- **Impact of Structural Encoding:** Removing the GAT module causes the sharpest drop in Branch Coverage ($\downarrow$ 5.2%). This confirms that the AST-based graph representation is essential for understanding the control flow of the focal methods.
- **Impact of Disentanglement:** The *w/o Disentanglement* variant suffers a significant decline in Mutation Score. Without explicitly separating logic from data, the model tends to generate repetitive test inputs, failing to explore diverse boundary conditions.
- **Impact of Gated Fusion:** Replacing the gated mechanism with simple concatenation leads to a minor performance drop, suggesting that the adaptive gate effectively filters out noise when fusing semantic and structural features.

3) RQ3: Disentanglement Analysis: To answer RQ3, we quantitatively investigate whether the proposed variational mechanism effectively disentangles the immutable program logic from the perturbable data patterns. Since the Methods2Test dataset does not provide fine-grained algorithmic labels, we constructed a labeled subset for analysis. We randomly selected 1,000 focal methods from the test set and employed GPT-4 to perform zero-shot topic modeling, identifying 5 distinct algorithmic categories: *String Manipulation*, *Numerical Calculation*, *Collection Handling*, *File I/O*, and *Security/Encryption*. These categories serve as the ground truth for our evaluation. We employ a Linear Probing technique to assess the semantic content of the learned representations. specifically, we freeze the pre-trained encoders and train simple linear classifiers on the Logic Latent vectors ($\mathcal{Z}_L$) and Data Latent vectors ($\mathcal{Z}_D$) respectively, aiming to predict the algorithmic category of the input code. Figure2 presents the classification accuracy on the held-out test set. The results reveal a sharp contrast between the two latent spaces: The quantitative results reveal a distinct separation between the two latent spaces. The classifier trained on the Logic Space ($\mathcal{Z}_L$) achieves a high accuracy of 94.2%, confirming that the logic encoder successfully captures the core functional

TABLE II
OVERALL PERFORMANCE COMPARISON WITH REPRESENTATIVE STATE-OF-THE-ART BASELINES ON THE METHODS2TEST DATASET

Model	Effectiveness Metrics (%)			Correctness Metrics (%)	
	Line Cov.	*Branch Cov.*	*Mut. Score*	*Comp.*	*Pass Rate*
Llama 3-8B	62.5	54.2	48.5	92.1	72.4
DeepSeek-Coder-V2	69.4	61.8	55.2	95.5	80.8
Qwen2.5-Coder	71.8	63.5	57.5	**97.2**	**84.5**
Magicoder-S-DS	70.1	62.0	56.1	96.0	81.2
CodeT5+ (Base)	65.2	56.8	51.4	94.8	78.2
SAD-Gen (Ours)	**78.4**	**69.5**	**63.8**	96.5	82.6

TABLE III
ABLATION STUDY OF KEY COMPONENTS

Variant	Branch Cov.	Mut. Score	Δ Avg.
SAD-Gen (Full)	**58.6**	**52.8**	-
w/o Structure	53.4 (↓5.2)	48.1	-4.9%
w/o Disentanglement	55.1 (↓3.5)	47.5	-4.4%
w/o Gated Fusion	57.2 (↓1.4)	51.0	-1.6%

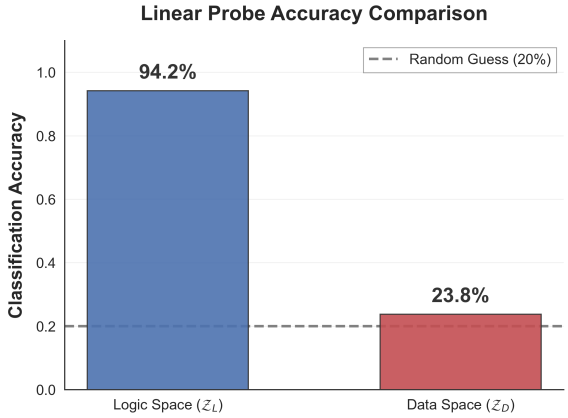

Fig. 2. **Linear Probe Accuracy on Latent Spaces.** The classifier trained on the Logic Space ($\mathcal{Z}_L$) achieves 94.2% accuracy, indicating it captures rich semantic information. In contrast, the Data Space ($\mathcal{Z}_D$) accuracy (23.8%) is close to the random guess baseline (dashed line), demonstrating successful disentanglement.

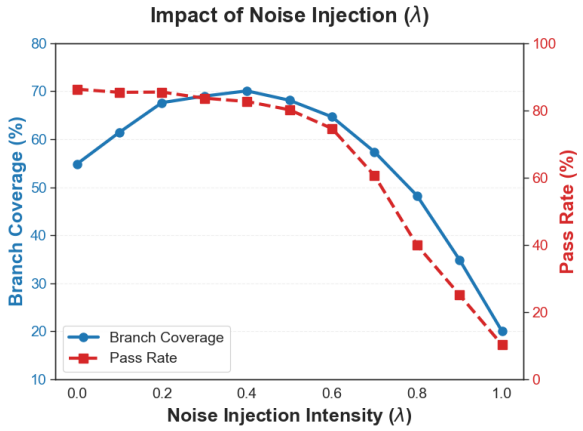

Fig. 3. **Impact of Noise Injection Intensity (λ) on Testing Performance.** The blue solid line (left axis) represents Branch Coverage, which peaks around $\lambda = 0.4$, indicating the optimal balance for exploration. The red dashed line (right axis) represents the Pass Rate, which remains stable initially but exhibits a sharp decline when λ exceeds approximately 0.6. This trade-off justifies our choice of $\lambda = 0.4$.

semantics and structural characteristics of the code, making different algorithms linearly separable. In sharp contrast, the classifier trained on the Data Space ($\mathcal{Z}_D$) yields an accuracy of only 23.8%, which is marginally above the random guess baseline (20.0% for 5 categories). This substantial performance gap serves as strong evidence that the logic information has been effectively stripped away from $\mathcal{Z}_D$, validating the effectiveness of our disentanglement mechanism. The substantial performance gap ($\Delta \approx 70\%$) quantitatively demonstrates the effectiveness of our disentanglement strategy. The Data Latent Space $\mathcal{Z}_D$ is proven to be agnostic to the program's functional logic, encoding only style-related or random patterns. This successful disentanglement is the theoretical premise that allows SAD-Gen to inject diversity-enhancing noise into $\mathcal{Z}_D$ without corrupting the correctness of the generated test cases.

4) RQ4: Sensitivity Analysis of Noise Injection: We analyze the impact of the noise injection intensity λ on the diversity and validity of generated tests by varying λ from 0.0 to 1.0 and recording the Branch Coverage and Pass Rate. As illustrated in Figure3, we observe a distinct "sweet spot" where Branch Coverage initially improves with the increase of λ, peaking around $\lambda = 0.4$. This trend suggests that moderate perturbations in the data latent space effectively encourage the model to explore diverse input values and boundary conditions. However, a trade-off emerges as λ continues to increase; specifically, when λ exceeds approximately 0.6, there is a sharp decline in the Pass Rate. Excessive noise appears to disrupt the validity of the data representation, causing the generated tests to become syntactically incorrect or logically incoherent. Based on these empirical results, we set $\lambda = 0.4$ as the optimal configuration to balance coverage maximization with test validity.

IV. RELATED WORK

Traditional automated test generation has evolved from Search-Based Software Testing tools like EvoSuite—which excel in coverage but often lack readability—to modern Deep Learning approaches. While LLMs such as Codex and StarCoder have revolutionized the field by generating fluent, human-like assertions, they frequently suffer from logical hallucinations due to a reliance on textual patterns that neglect the underlying non-Euclidean code structure. Although

integrating structural information via GNNs and models like GraphCodeBERT [26] has emerged as a critical research direction, effectively aligning these structural features with textual semantics remains a significant challenge; existing frameworks often rely on suboptimal static concatenation, a limitation our proposed SAD-Gen overcomes by employing an adaptive gated fusion mechanism to dynamically balance sequential and structural signals.

V. Conclusion and Limitations

In this paper, we proposed SAD-Gen, a novel framework that enhances automated unit test generation by explicitly disentangling code logic from data patterns via a structure-aware dual-latent variational mechanism. Our extensive experiments demonstrate that SAD-Gen effectively balances test diversity and validity, achieving SOTA performance in Branch Coverage and Pass Rate compared to existing LLM-based approaches. While this study validates the effectiveness of SAD-Gen on standard benchmarks, industrial-grade software often involves intricate proprietary logic and specific coding standards. Consequently, the generalizability of our framework in such heterogeneous environments remains to be fully explored through real-world deployment. Future work will focus on addressing these industrial-scale constraints and extending SAD-Gen to handle complex external dependencies in broader software engineering workflows.

References

[1] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*. Wiley Online Library, 2004, vol. 2.

[2] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[3] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.

[4] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of bert models for code completion," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 108–119.

[5] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[6] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[7] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1703–1726, 2024.

[8] S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of automated unit test generation for python," *Empirical Software Engineering*, vol. 28, no. 2, p. 36, 2023.

[9] V. Agarwal, Y. Pei, S. Alamir, and X. Liu, "Codemirage: Hallucinations in code generated by large language models," *arXiv preprint arXiv:2408.08333*, 2024.

[10] A. Eghbali and M. Pradel, "De-hallucinator: Mitigating llm hallucinations in code generation tasks via iterative grounding," *arXiv preprint arXiv:2401.01701*, 2024.

[11] N. Jiang, Q. Li, L. Tan, and T. Zhang, "Collu-bench: A benchmark for predicting language model hallucinations in code," *arXiv preprint arXiv:2410.09997*, 2024.

[12] M. Tiezzi, G. Marra, S. Melacci, M. Maggini *et al.*, "Deep lagrangian propagation in graph neural networks," in *ICML Workshop on Graph Representation Learning and Beyond (GRL+)*, 2020.

[13] Z. Pan, L. Xu, and N. Chen, "Combining graph neural network and convolutional lstm network for multistep soil moisture spatiotemporal prediction," *Journal of Hydrology*, vol. 651, p. 132572, 2025.

[14] L. Yang, Z. Miao, T. Li, S. Tan, B. Wang, D. Li, Y. Liu, H. Wei, J. Li, J. Li *et al.*, "Lstm-gcn based multidimensional parameter relationship analysis and prediction framework for system level experimental bench," *Annals of Nuclear Energy*, vol. 210, p. 110890, 2025.

[15] K. Bayoudh, R. Knani, F. Hamdaoui, and A. Mtibaa, "A survey on deep multimodal learning for computer vision: advances, trends, applications, and datasets," *The Visual Computer*, vol. 38, no. 8, pp. 2939–2970, 2022.

[16] S. T. Cynthia and B. Roy, "An empirical study on the impact of gender diversity on code quality in ai systems," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 2025, pp. 1540–1549.

[17] H. Zhang, Y.-F. Zhang, W. Liu, A. Weller, B. Schölkopf, and E. P. Xing, "Towards principled disentanglement for domain generalization," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 8024–8034.

[18] I. Higgins, D. Amos, D. Pfau, S. Racaniere, L. Matthey, D. Rezende, and A. Lerchner, "Towards a definition of disentangled representations," *arXiv preprint arXiv:1812.02230*, 2018.

[19] Z. Liu, M. Li, C. Han, S. Tang, and T. Guo, "Stdnet: Rethinking disentanglement learning with information theory," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 8, pp. 10 407–10 421, 2023.

[20] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, "Codet5+: Open code large language models for code understanding and generation," in *Proceedings of the 2023 conference on empirical methods in natural language processing*, 2023, pp. 1069–1088.

[21] C. Ding, S. Sun, and J. Zhao, "Mst-gat: A multimodal spatial–temporal graph attention network for time series anomaly detection," *Information Fusion*, vol. 89, pp. 527–536, 2023.

[22] M. Tufano, S. K. Deng, N. Sundaresan, and A. Svyatkovskiy, "Methods2test: A dataset of focal methods mapped to test cases," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 299–303.

[23] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *CoRR*, vol. abs/2108.07732, 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[24] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[25] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2022. [Online]. Available: https://arxiv.org/abs/1312.6114

[26] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," 2021. [Online]. Available: https://arxiv.org/abs/2009.08366