CGBridge: Bridging Code Graphs and Large Language Models for Better Structure-Aware Code Understanding

Zeqi Chen[1], Zhaoyang Chu[2], Yi Gui[2], Feng Guo[1], Yao Wan[2], Chuan Shi[1]*
[1]Beijing University of Posts and Telecommunications
[2]Huazhong University of Science and Technology
{chenzeqi, shichuan}@bupt.edu.cn

Abstract

Large Language Models (LLMs) have demonstrated remarkable performance in code intelligence tasks such as code generation, summarization, and translation. However, their reliance on linearized token sequences makes them brittle to long-range program dependencies and superficial lexical shifts such as identifier renaming. Existing structure-aware approaches typically treat structure as serialized text prompts or auxiliary training objectives, which often inflate context length or rely on internalized structural priors, failing to provide explicit guidance during inference. To address these limitations, we propose **CGBRIDGE**, a novel plug-and-play method that enhances LLMs with Code Graph information through an external, trainable BRIDGE module. It aligns Code Property Graph structure with code semantics and compresses them into compact soft-prefixes, decoupling structural reasoning from textual generation without updating the backbone. Experiments across multiple code LLM backbones and scales show consistent gains over both text-only adaptation and graph-augmented baselines. Furthermore, CGBRIDGE remains robust under identifier renaming and enables over 4× faster inference than LoRA-tuned models, demonstrating both effectiveness and efficiency in structure-aware code understanding. The code is available at: https://github.com/BUPT-GAMMA/CGBridge.git.

1 Introduction

Large language models (LLMs) have advanced software engineering, powering applications like code generation, summarization, and translation (Wan et al., 2024). Recent code-focused LLMs like Code Llama (Rozière et al., 2024), Qwen-Coder (Hui et al., 2024), and DeepSeek-Coder (Guo et al., 2024) achieve strong performance by scaling up model and training data. Despite this success, they typically process code as linear text, overlooking the program's structural semantics captured by Abstract Syntax Trees (ASTs), Control-Flow Graphs (CFGs), and Data-Flow Graphs (DFGs), which can be naturally unified into a Code Property Graph (CPG) (Yamaguchi et al., 2014). This "code-as-text" paradigm forces models to rely on surface heuristics—such as naming patterns—rather than underlying logic, making them brittle to structural reasoning or simple obfuscation (Hooda et al., 2024). Fig. 1 illustrates this weakness: when identifiers are obfuscated (e.g., `price` → `efg`), the base model fails to capture the program logic, revealing its reliance on surface lexical patterns. Motivated by this brittleness, prior work has sought to incorporate program structure into language models.

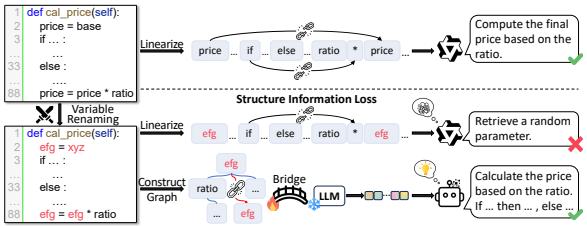

Figure 1: **A motivating example.** While text-only methods fail under identifier obfuscation due to lexical reliance, explicit structural guidance enables robust reasoning independent of naming patterns.

Existing attempts can be broadly categorized into three paradigms. (1) *Structure-aware neural code models* (Guo et al., 2021, 2022) inject AST/DFG signals into Transformer architectures and are typically fine-tuned with supervised, task-specific objectives or heads; however, they are designed for relatively small encoders and are less compatible with today's end-to-end instruction-following, decoder-only (often frozen) code LLM setting. (2) *Graph-as-text prompting* (Zhao et al., 2023; Lu et al., 2024) serializes graphs into long prompts fed to an LLM; while

*Corresponding author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 8945–8966
July 2-7, 2026

simple and model-agnostic, this often inflates context length and dilutes higher-order dependencies in lengthy sequences. (3) *Graph-augmented LLM training* (Zhang et al., 2025e; Li et al., 2023a) internalizes structural priors via joint training or backbone modification, but it is costly and commonly relies on internalized structural priors at inference. Without an explicit inference-time structural anchor, distribution shifts (e.g., identifier renaming) can cause structural loss and lexical guessing.

In this work, rather than treating structure as an auxiliary feature, a prompt artifact, or an implicit prior, we treat the CPG as an independent structural modality serving as an explicit structural anchor at inference time. We introduce CGBRIDGE, a plug-and-play framework that augments frozen LLMs with Code Graphs via an external, trainable BRIDGE module (Fig. 3). CGBRIDGE bridges non-Euclidean graphs and sequential text by learning a reusable structural expert (Code Graph Encoder) and a lightweight interface (Bridge module) that aligns CPG structural signals with code semantics and compresses them into compact soft-prefixes, thereby decoupling structural reasoning from textual generation without updating the backbone.

Our contributions are threefold: (1) we propose a new perspective for structure-aware code intelligence in the era of instruction-following LLMs—treating the CPG as an external structural modality and as an explicit inference-time structural anchor; (2) we instantiate this perspective with CGBRIDGE and a three-stage graph–LLM alignment strategy that learns a reusable Code Graph Encoder and a lightweight Bridge, aligns CPG structural signals with code semantics via multi-objective learning, and compresses them into soft-prefixes for a frozen backbone; and (3) we demonstrate through extensive experiments that CGBRIDGE yields consistent gains on code summarization and translation across diverse model families and scales, exhibits superior robustness under systematic identifier renaming, and achieves up to $4\times$ faster inference than LoRA-tuned models, demonstrating both effectiveness and efficiency.

2 Related Work

Code LLMs and Parameter-Efficient Adaptation. Code-focused LLMs (e.g., CodeLlama, Qwen-Coder, and DeepSeek-Coder) are typically obtained by full-parameter continual pretraining on large-scale code corpora. For downstream application, practitioners often employ PEFT methods (Xu et al., 2023) such as LoRA (Hu et al., 2022). However, text-only adaptation mainly optimizes token likelihood and can under-utilize long-range structural relations (e.g., control/def-use dependencies) that are central to program semantics.

Structure-Aware Neural Code Models. Prior to LLMs, many works enhanced code representation learning by injecting structural edges into neural architectures. For example, GraphCodeBERT and UniXcoder (Guo et al., 2021, 2022) inject AST/DFG signals into Transformer encoders to improve structural awareness. However, these methods typically target relatively small encoders and are fine-tuned with task-specific objectives or heads, making them less compatible with today's instruction-following, decoder-only code LLMs.

Injecting Graphs into LLMs. Recent graph–LLM methods can be grouped into three paradigms. (1) *Graph-as-text prompting* linearizes graphs into long prompts (e.g., GRACE (Lu et al., 2024)) or employs retrieval-augmented strategies (e.g., cAST (Zhang et al., 2025d), CODEXGRAPH (Liu et al., 2025), RepoGraph (Ouyang et al., 2025), CodeMEM (Wang et al., 2026)). While flexible and model-agnostic, methods in this line may suffer from context inflation and diluted structural signals. (2) *Graph-augmented training / backbone modification* incorporates structural priors via joint training or architectural integration (e.g., Graphix-T5 (Li et al., 2023a), GALLa (Zhang et al., 2025e)). Recent concurrent work also explored complementary graph-guided code LLM designs, including CPG-guided vulnerability detection and repository-level code graph integration (e.g., LLMxCPG (Lekssays et al., 2025), CGM (Tao et al., 2025)). While effective, these methods are often costly for large backbones and leave structural guidance largely implicit at inference time. (3) *Graph-LLM cross-modal alignment* aligns graph and text representations (e.g., GraphCLIP (Zhu et al., 2025)) or distills graphs into compact prefixes (e.g., GraphLLM (Chai et al., 2023), LLaGA (Chen et al., 2024b)). However, these general-purpose aligners transfer poorly to source code: they mainly target homogeneous or text-attributed graphs and graph reasoning/discriminative settings, rather than instruction-following code tasks grounded in heterogeneous program structure (AST/CFG/DFG). In contrast, our work treats the CPG as explicit inference-time structural

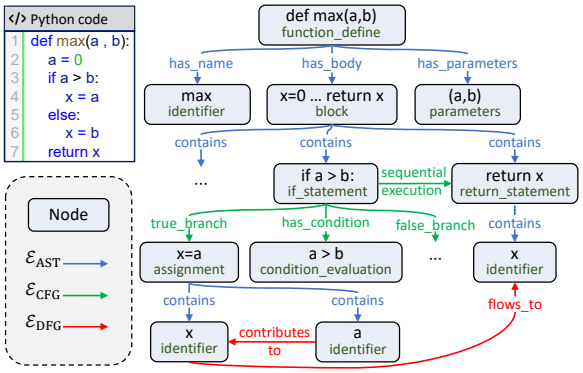

Figure 2: A Python function and its corresponding CPG.

guidance under a frozen backbone.

3 Preliminaries

3.1 Code Property Graph

To capture the heterogeneous semantics of source code, we employ the Code Property Graph (CPG), a unified representation widely used in program analysis (Liu et al., 2023). Fig. 2 illustrates a simplified CPG for a Python function, where three distinct structural dependencies are intertwined to provide a holistic view. Specifically, **the blue AST edges** encode the grammatical hierarchy and compositional structure (Sun et al., 2023), linking high-level constructs (`function_define`) to their constituents (`names`, `parameters`, `bodies`); **the green CFG edges** explicitly map the execution logic, defining control dependencies between statements such as the `true` and `false` branches of conditionals (Viet Phan et al., 2017); and **the red DFG edges** track the data flow (Guo et al., 2021), delineating how variable definitions (e.g., `a`) propagate to downstream assignments (e.g., `x`).

3.2 Problem Formulation

We focus on instruction-following code generation grounded in structural context. Formally, let $\mathcal{C}$ denote a source code snippet and $\mathcal{G}$ denote its corresponding CPG, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where nodes represent syntactic elements and edges capture structural relations. Given a natural language instruction $\mathcal{I}$ (e.g., "summarize the code" or "translate Python to Java"), the goal is to generate a target response $\mathcal{Y} = (y_1, \ldots, y_m)$. Unlike standard LLMs, we explicitly condition generation on the structural graph $\mathcal{G}$. The objective is to optimize the model parameters θ by maximizing the log-likelihood:

$$\max_{\theta} \sum_{t=1}^{m} \log P(y_t \mid y_{<t}, \mathcal{G}, \mathcal{C}, \mathcal{I}; \theta)\,, \tag{1}$$

where $y_{<t}$ is the generated prefix before step t.

4 CGBridge: The Proposed Method

We propose **CGBridge**, a novel plug-and-play approach for incorporating Code Graph information into LLMs via a trainable Bridge module. As illustrated in Fig. 3, CGBridge follows a three-stage training procedure: **Stage 1 (Self-Supervised Structural Pretraining)** constructs a reusable structural expert by training a Code Graph Encoder (CGE) on large-scale unlabeled CPGs via two self-supervised objectives; **Stage 2 (Cross-Modal Semantic Alignment)** trains an external Bridge module to align structural representations with code-text semantics via three complementary objectives, compressing each CPG into compact embeddings for the LLM token space; **Stage 3 (Instruction-Based Task Adaptation)** leverages the Bridge module to inject learned graph-derived soft prompts into the frozen LLM, fine-tuning the Bridge for downstream code intelligence tasks.

4.1 Self-Supervised Structural Pretraining

4.1.1 Code Graph Encoder

We model a source code snippet $\mathcal{C}$ as a heterogeneous CPG $\mathcal{G}$ and employ a CGE to encode it into structural embeddings $H_{\mathcal{G}}$. The CGE is encoder-agnostic and can be instantiated by different graph encoders. In this work, we instantiate it as an L_g-layer graph neural network (GNN). At layer l, the representation of node i is updated by aggregating edge-conditioned messages from its neighbors:

$$h_i^{(l)} = \phi\big(h_i^{(l-1)}, \bigoplus_{j \in \mathcal{N}(i)} \psi\left(h_j^{(l-1)}, e_{ji}\right)\big)\,, \tag{2}$$

where $\mathcal{N}(i)$ is the neighbor set of node i, $\psi(\cdot)$ combines the neighbor-node representation $h_j^{(l-1)}$ with the edge feature e_{ji}, $\bigoplus$ is a permutation-invariant aggregator, and $\phi(\cdot)$ is the node-state update function. Conditioning on e_{ji} is crucial for heterogeneous CPGs, as it enables CGE to distinguish different relation types (e.g., syntactic vs. data-flow). After L_g layers, $H_{\mathcal{G}} = \{h_i^{(L_g)}\}_{i \in \mathcal{V}} \in \mathbb{R}^{|\mathcal{V}| \times d_{\text{model}}}$ will serve as the structural input for the Bridge.

4.1.2 Training Procedure

To ensure the CGE can produce meaningful graph representations, we first pre-train it on a large corpus of unlabeled code graphs, guided by the following two self-supervised objectives (Fig. 3(a)).

Graph-Level Contrastive Learning. To learn robust global representations, we adopt contrastive

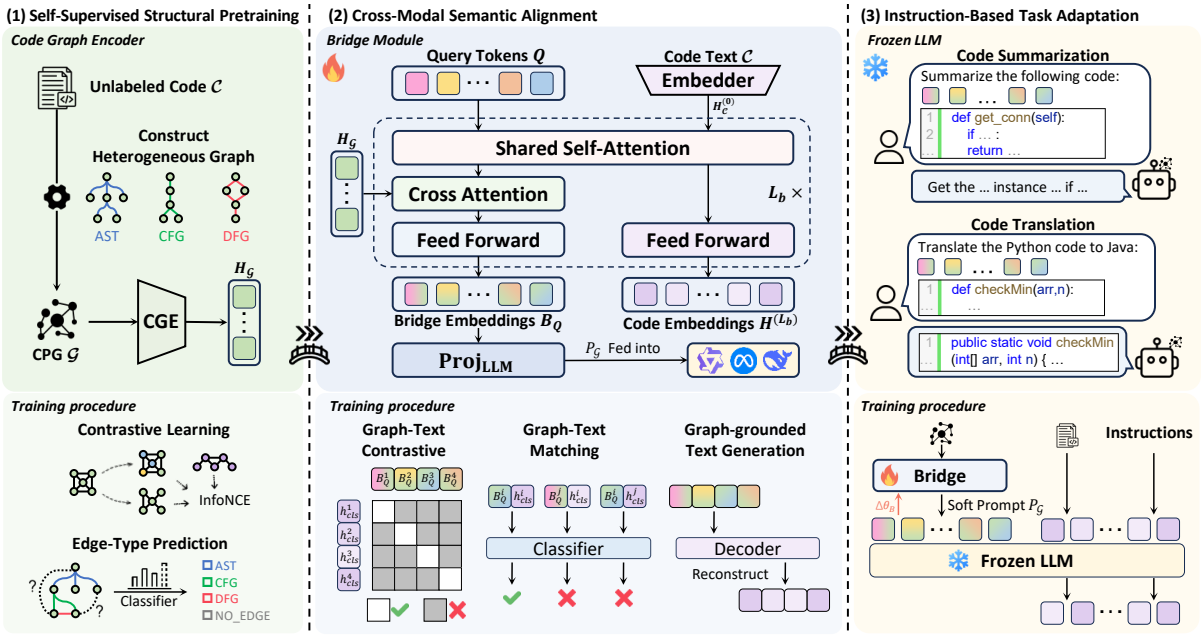

Figure 3: **Overview of CGBRIDGE.** (1) **Code Graph Encoder**: Pre-trains on heterogeneous graphs to extract structural features. (2) **Bridge Module**: Aligns graph representations with textual semantics to synthesize soft-prompts. (3) **Frozen LLM**: Utilizes the injected structural guidance for efficient, instruction-based task adaptation.

learning (You et al., 2020). For each graph, we generate two augmented views (e.g., via feature masking and edge dropping) and train the CGE to maximize their embedding similarity while minimizing similarity against batch negatives, using an InfoNCE loss (van den Oord et al., 2018) ($\mathcal{L}_{\text{cl}}$).

Edge-Type Prediction. To capture fine-grained relational semantics, we predict edge types for sampled node pairs. We incorporate negative sampling (Grover and Leskovec, 2016) by including unconnected pairs, which the model must classify as a distinct "NO_EDGE" type. This classification loss is denoted as $\mathcal{L}_{\text{edge}}$.

The loss for Stage 1 is a weighted sum of the two objectives: $\mathcal{L}_{\text{Stage1}} = \lambda_{\text{cl}} \cdot \mathcal{L}_{\text{cl}} + \lambda_{\text{edge}} \cdot \mathcal{L}_{\text{edge}}$.

4.2 Cross-Modal Semantic Alignment

4.2.1 Bridge Module

The Bridge module addresses the modality gap between the graph embeddings and the LLM's sequential input space. Architecturally, it is an L_b-layer Transformer (Vaswani et al., 2017) with N_q learnable query tokens, $Q \in \mathbb{R}^{N_q \times d_{\text{model}}}$, following multimodal learning paradigms (Li et al., 2023b). Specifically, the module takes the structural embeddings $H_{\mathcal{G}}$ from CGE and the code-text embeddings $H_{\mathcal{C}}$ (from $\mathcal{C}$ via the module's text embedder) as inputs. Within layer l, Q interacts with these cross-modal representations through distinct attention mechanisms. The process unfolds in three steps:

1) Multi-Modal Fusion via Shared Self-Attention. $Q^{(l-1)}$ and $H_{\mathcal{C}}^{(l-1)}$ from the previous layer are concatenated and processed by a shared self-attention layer, allowing them to exchange information. For $l = 1$, the inputs are the initial learnable queries Q and code text embeddings $H_{\mathcal{C}}$:

$$[Q_{\text{sa}}^{(l)}, H_{\mathcal{C},\text{sa}}^{(l)}] = \text{Self-Attn}([Q^{(l-1)}, H_{\mathcal{C}}^{(l-1)}])\,. \quad (3)$$

2) Graph Information Extraction via Cross-Attention. $Q_{\text{sa}}^{(l)}$ from the self-attention step then acts as the query to extract relevant structural information from $H_{\mathcal{G}}$ via cross-attention:

$$Q_{\text{cross}}^{(l)} = \text{Cross-Attn}(Q = Q_{\text{sa}}^{(l)}, K = H_{\mathcal{G}}, V = H_{\mathcal{G}})\,. \quad (4)$$

3) Feed-Forward Network. The output representations from the attention layers are refined by separate Feed-Forward Networks (FFNs):

$$Q^{(l)} = \text{FFN}_Q(Q_{\text{cross}}^{(l)}), \quad H_{\mathcal{C}}^{(l)} = \text{FFN}_C(H_{\mathcal{C},\text{sa}}^{(l)})\,. \quad (5)$$

The Bridge outputs $B_Q = Q^{(L_b)} \in \mathbb{R}^{N_q \times d_{\text{model}}}$, which is projected into a soft prompt $P_{\mathcal{G}} = \text{Proj}_{\text{LLM}}(B_Q)$ to guide task-specific generation.

4.2.2 Training Procedure

This stage aligns code graph-text pairs $(H_{\mathcal{G}}, \mathcal{C})$ within the Bridge module, independent of the LLM

backbone (Fig. 3(b)). Following established multimodal learning paradigms (Li et al., 2022), we employ three complementary objectives:

Graph-Text Contrastive Learning (GTC). This objective aligns global representations of graphs and texts. Given a batch of M paired samples, the Bridge produces $B_Q^i \in \mathbb{R}^{N_q \times d_{\text{model}}}$ for each sample i, and the text encoder provides h_{cls}^i (the `[CLS]` embedding from $H_{\mathcal{C}}^{(L_b)}$). We use symmetric InfoNCE (Zhang et al., 2023) to pull matched pairs $(B_Q^i, h_{\text{cls}}^i)$ together and push apart negatives:

$$\mathcal{L}_{\text{GTC}} = \text{SymInfoNCE}\left(h_{\text{cls}}, B_Q, \tau\right). \quad (6)$$

Similarity is computed as the max cosine similarity over N_q queries, scaled by a learnable τ; the detailed implementation is provided in Appendix I.

Graph–Text Matching (GTM). This objective trains a binary classifier ϕ_{gtm} to predict whether a graph-text pair $(B_Q^i, h_{\text{cls}}^i)$ matches, with predicted probability $p^i = \phi_{\text{gtm}}(B_Q^i, h_{\text{cls}}^i)$. We minimize the standard cross-entropy over M samples with hard negative mining (Robinson et al., 2021):

$$\mathcal{L}_{\text{GTM}} = -\frac{1}{M}\sum_{i=1}^{M}\Big(y^i \log(p^i) + (1-y^i)\log(1-p^i)\Big). \quad (7)$$

Graph-grounded Text Generation (GTG). This objective encourages B_Q to encapsulate a comprehensive summary of the graph by reconstructing the code text $\mathcal{C}$. During autoregressive decoding, each token attends to B_Q, grounding the generation in the structural representation. The loss is the average negative log-likelihood over M samples, where the model predicts each token $t_{\mathcal{C},j}^i$ of the i-th sequence $t_{\mathcal{C}}^i$ conditioned on its preceding tokens $t_{\mathcal{C},<j}^i$ and the corresponding B_Q^i:

$$\mathcal{L}_{\text{GTG}} = -\frac{1}{M}\sum_{i=1}^{M}\sum_{j=1}^{|\mathcal{C}^i|}\log P\left(t_{\mathcal{C},j}^i \mid t_{\mathcal{C},<j}^i, B_Q^i\right). \quad (8)$$

The loss for Stage 2 is the sum of the three objectives: $\mathcal{L}_{\text{Stage2}} = \mathcal{L}_{\text{GTC}} + \mathcal{L}_{\text{GTM}} + \mathcal{L}_{\text{GTG}}$.

4.3 Instruction-Based Task Adaptation

4.3.1 Frozen LLM

CGBRIDGE uses a domain-specialized code LLM as the final inference engine (Fig. 3(c)). We keep the LLM fully frozen and perform task adaptation solely through the Bridge module, thereby leveraging the foundation model's knowledge while avoiding the computational cost of LLM fine-tuning.

4.3.2 Training Procedure

Following cross-modal alignment, this final stage adapts the Bridge module to downstream tasks under a frozen LLM setup (Fig. 3(c)). Specifically, the task instruction $\mathcal{I}$ and source code $\mathcal{C}$ are first tokenized and mapped into continuous embeddings through the LLM's input embedding layer, denoted as $E(\mathcal{I})$ and $E(\mathcal{C})$. The soft prompt $P_{\mathcal{G}}$ from Bridge Module is then concatenated with these text embeddings in the continuous space, forming the composite LLM input: $C_{\text{LLM}} = [P_{\mathcal{G}}; E(\mathcal{I}); E(\mathcal{C})]$. The frozen LLM is then conditioned on C_{LLM} to autoregressively generate the target answer sequence T_A. Training in Stage 3 is supervised and updates only the Bridge parameters θ_B by minimizing the negative log-likelihood of the target tokens:

$$\mathcal{L}_{\text{Stage3}}(\theta_B) = -\sum_{i=1}^{|T_A|}\log P_{\text{LLM}}(t_i \mid t_{<i}, C_{\text{LLM}}). \quad (9)$$

5 Experiments

5.1 Experimental Setup

Datasets and Tasks. We evaluate CGBRIDGE on two representative code intelligence tasks that require deep structural understanding. (1) **Code Summarization** requires capturing key control-flow paths, data dependencies, and functional roles to produce accurate natural language descriptions; we use the *CodeSearchNet (Python subset)* (Husain et al., 2020). (2) **Code Translation** re-expresses the same semantics across different languages, and thus relies on preserving abstract syntax and long-range dependencies; we use the *XLCoST (Python to Java)* (Zhu et al., 2022). For each snippet, we construct a CPG (AST+CFG+DFG) as the structural input. Detailed dataset refinement, graph construction, and statistics are provided in Appendix H. The training data used in each stage of CGBRIDGE and for the baselines are detailed in Appendix J.

Backbones and Baselines. We evaluate CGBRIDGE across four code LLM backbones spanning multiple families and scales: Qwen2.5-Coder (1.5B, 7B) (Hui et al., 2024), CodeLlama-7B (Rozière et al., 2024), and DeepSeek-Coder (1.3B, 6.7B) (Guo et al., 2024). For each backbone, we compare against three general baseline categories: Zero-shot (vanilla LLM), +LoRA (Hu et al., 2022) (text-only PEFT), and +GraphText (Wang et al., 2023) (graph-as-text prompting). In addition, we report a focused comparison with representative

Table 1: Code Summarization (left) and Code Translation (right) results across model groups. All scores are in %, except LLM-J (0–4). Bold denotes the best per group.

Method	Code Summarization				Code Translation			
	ROUGE-L	SBCS	B-Score	LLM-J	CB	SM	DM	EA
Qwen2.5-Coder-1.5B	16.40	56.68	81.96	2.70	58.89	71.43	40.74	70.63
+ LoRA	**23.40**	56.97	84.86	2.85	63.76	76.64	63.44	84.68
+ GraphText	13.69	51.61	81.36	2.66	55.68	64.27	54.28	57.26
+ CGBRIDGE	23.04	**59.12**	**85.56**	**3.18**	**69.81**	**78.80**	**68.55**	**89.01**
Qwen2.5-Coder-7B	19.91	59.13	83.57	2.78	63.39	74.67	48.07	89.46
+ LoRA	**23.60**	60.53	86.27	2.98	71.16	**77.72**	66.42	97.21
+ GraphText	15.94	53.18	82.77	2.96	63.52	67.85	67.52	70.75
+ CGBRIDGE	20.08	**61.62**	**86.61**	**3.23**	**73.91**	77.20	**74.18**	**98.26**
CodeLlama-7B	18.71	55.85	82.91	2.83	27.64	54.65	46.05	71.76
+ LoRA	**25.31**	60.44	86.23	2.89	69.10	74.14	70.88	93.17
+ GraphText	14.30	49.50	81.60	2.87	57.81	62.84	59.02	60.32
+ CGBRIDGE	23.38	**60.65**	**88.35**	**2.99**	**71.32**	**79.21**	**74.49**	**95.07**

Graph-LLM integrated methods—GALLa (Zhang et al., 2025e), GraphLLM (Chai et al., 2023), and GraphCLIP (Zhu et al., 2025)—on the Qwen2.5-Coder-1.5B. We limit these specialized baselines to a single backbone due to their distinct training paradigms and relatively high computational cost.

Evaluation Metrics. To prioritize **structural** and **semantic** fidelity while adhering to established metrics in neural software engineering, we focus on reference-free and functional metrics, and report lexical similarity as complementary. For summarization, we use LLM-as-a-Judge (LLM-J) (Wu et al., 2024; Wang et al., 2025) to score overall quality (0–4) based on {Coherence, Consistency, Fluency, Relevance}, supplemented by ROUGE-L (Lin, 2004), Sentence-BERT cosine similarity (SBCS) (Reimers and Gurevych, 2019) and BERTScore (B-Score) (Zhang et al., 2020). For translation, we measure functional correctness via Execution Accuracy (EA), alongside CodeBLEU (CB) (Ren et al., 2020), Syntax Match (SM), and Dataflow Match (DM) for structural alignment.

Implementation Details. We provide implementation details (including the CGE selection), hyperparameters with sensitivity analysis, and prompt templates in Appendices J, K, and L respectively.

5.2 Overall Performance

Overall performance across backbones. Table 1 shows that CGBRIDGE yields consistent gains across model families and scales, especially on our primary semantic and functional metrics. **On summarization**, CGBRIDGE improves LLM-J across backbones (e.g., 2.83 (Zero-shot) / 2.89 (LoRA)→2.99 (CGBRIDGE) on CodeLlama-7B and 2.78 / 2.98 → 3.23 on Qwen2.5-Coder-7B), indicating that structural alignment yields more logically faithful summaries beyond surface overlap. In contrast, LoRA remains competitive on lexical-overlap metrics (ROUGE-L), its gains on semantic metrics (B-Score and LLM-J) are marginal, suggesting that text-only adaptation mainly improves surface form rather than semantic fidelity and overall quality. **On translation**, CGBRIDGE improves both functional correctness and structural alignment (e.g., on Qwen2.5-Coder-1.5B, EA: 70.63 / 84.68 → 89.01 (CGBRIDGE); DM: 40.74 / 63.44 → 68.55), indicating better capture of executable semantics and variable dependencies than text-only baselines. We also observe consistent improvements on DeepSeek-Coder (1.3B/6.7B); full results are reported in Appendix A. To verify statistical robustness, 5-seed evaluations on the Qwen2.5-Coder-7B backbone demonstrate that CGBRIDGE consistently outperforms LoRA with lower variance (detailed in Appendix G).

Table 2: Comparison with Graph-LLM Methods.

Method	CB	SM	DM	EA
Qwen2.5-Coder-1.5B	58.89	71.43	40.74	70.63
+ GraphLLM	65.11	74.68	53.81	79.89
+ GraphCLIP	68.24	76.57	50.25	74.76
+ GALLa	69.47	78.19	62.73	86.39
+ **CGBRIDGE**	**69.81**	**78.80**	**68.55**	**89.01**

Comparison with Graph–LLM Methods. Table 2 compares CGBRIDGE with representative Graph–LLM approaches on a controlled setting (Qwen2.5-Coder-1.5B, code translation). CGBRIDGE achieves the best CB (69.81) and EA (89.01), outperforming GraphLLM and GraphCLIP by +9.12 and +14.25 EA points, respectively, and also surpasses GALLa in EA (89.01 vs. 86.39) under a frozen backbone. These results show that explicit inference-time structural grounding via the

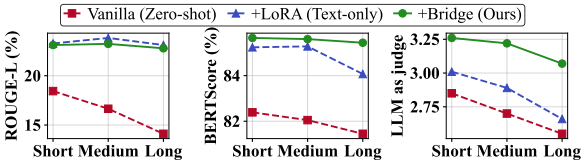

Figure 4: Performance across different code length bins.

Table 3: Robustness to variable renaming.

Method	Original	Obfuscated	Change
Qwen2.5-Coder-7B	2.78	2.71	-2.5%
+ LoRA	2.98	2.53	-15.1%
+ GraphText	2.96	2.13	-28.4%
+ **CGBridge**	3.23	3.22	**-0.3%**

Bridge module is an effective and competitive design for graph-enhanced code understanding.

Performance vs. Code Complexity. To further examine whether the observed gains are associated with improved long-range structural understanding, we partition the test set into three relative code-length bins via equal-frequency binning based on token counts computed with the Qwen2.5 tokenizer. As code becomes longer, all methods degrade, but CGBridge exhibits the smallest drop, especially on semantic metrics (BERTScore and LLM-J), whereas text-only baselines deteriorate more noticeably on Long code. This suggests that explicit CPG-based structural anchoring helps frozen LLMs preserve program-level semantics and long-range dependencies in more complex code.

5.3 Additional Evaluations

Code execution understanding on CRUXEval-O. We further evaluate CGBridge on *CRUXEval-O*, the output-prediction task of CRUXEval (Gu et al., 2024), which explicitly tests execution tracing and structural reasoning. We report Pass@1 both with and without Stage 3 task adaptation. CGBridge improves performance in both settings, while Stage 3 adaptation yields additional gains, further supporting the benefit of explicit graph grounding for structural code understanding. Detailed results are provided in Appendix B.

Cross-language generalization. To verify that CGBridge generalizes beyond Python source inputs, we further evaluate it on the *XLCoST (Java to Python)* translation task. CGBridge consistently yields substantial and statistically significant improvements, confirming robust cross-language generalization beyond Python-input settings. Detailed results are provided in Appendix C.

Table 4: Efficiency Analysis.

Method	Trainable Params (M)	Training Time (s)[a]	Infer. Latency (ms)[b]
Qwen2.5-Coder-7B	-	-	∼299
+ LoRA	43.12	110.5	∼1463
+ GraphText	-	-	∼1197
+ **CGBridge**[c]	180.80	272.1	**∼371**

[a] Measured per epoch. [b] Measured per sample.
[c] Inference latency excludes offline graph construction (∼281 ms/sample) and embedding generation (∼0.42 ms/sample).

Compatibility with instruct and thinking variants. We further evaluate CGBridge on an instruct/thinking pair from the Qwen3-4B (Yang et al., 2025) series as an additional compatibility check. CGBridge yields statistically significant gains on both variants, indicating that explicit structural grounding remains beneficial under both standard instruction-following and reasoning-augmented decoding styles. Detailed results are provided in Appendix D.

5.4 Robustness to Identifiers Renaming

CGBridge exhibits superior robustness to identifier renaming by reducing reliance on surface-level semantics. Following (Hooda et al., 2024), we verify this by systematically replacing function and variable names with random identifiers (e.g., `compute`→`xYz`) while preserving external library calls. Since renaming makes lexical n-gram metrics (ROUGE) less informative, we report the semantic LLM-J score. On Qwen2.5-Coder-7B (Table 3), text-only baselines degrade substantially (e.g., LoRA drops by 15.1%), whereas CGBridge remains stable with only a -0.3% change; results on 1.5B show the same trend (Appendix E). These results suggest that explicit structural graphs help capture program logic and data flow, resisting superficial identifier variations.

5.5 Efficiency Analysis

Table 4 compares the spatial and temporal efficiency of CGBridge with baselines on 7B, while Appendix F details results for 1.5B.

CGBridge achieves strong inference efficiency, delivering a ∼4× speedup over LoRA on both 7B (∼371ms vs. ∼1463ms) and 1.5B (∼215ms vs. ∼969ms). CGBridge gains efficiency by injecting compact prefixes upfront, avoiding the context inflation of GraphText or the per-layer autoregressive decoding overhead of LoRA. Regarding the pre-processing cost, the graph construction and embedding generation (∼282 ms/sample) can be

Table 5: Ablation experiments of graph components.

Graph Components	CB	SM	DM	EA
AST	69.97	76.55	65.95	93.12
AST + CFG	70.64	74.82	69.21	94.53
AST + DFG	71.76	74.07	73.62	97.39
CFG + DFG	65.10	68.21	72.00	91.57
CPG (AST + CFG + DFG)	**73.91**	**77.20**	**74.18**	**98.26**

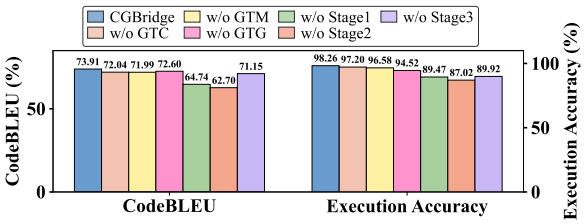

Figure 5: Ablation experiments of training components.

executed fully offline and parallelized, making it negligible for real-time inference.

CGBRIDGE exhibits excellent scalability in parameter efficiency. The Bridge module maintains a fixed size (180.80M) independent of the LLM scale, so its relative overhead shrinks as the backbone grows (from ∼11.5% (1.5B) to ∼2.5% (7B), and a projected ∼0.26% at 70B), making CGBRIDGE a parameter-efficient solution for adapting future large-scale foundation models.

5.6 Ablation Study

We conduct two ablation studies on code translation with Qwen2.5-Coder-7B.

Impact of Graph Components. Table 5 shows that the full CPG achieves the best EA (98.26%). Notably, removing the AST backbone (leaving only CFG+DFG) drops EA to 91.57% (-6.69%), even below AST-only. This suggests that **AST provides the dense syntactic backbone that keeps structural context well-formed**; without it, the remaining CFG/DFG edges become too sparse and less well-connected, weakening message passing and diminishing the benefit of semantic edges. Thus, the superior performance stems from anchoring semantic flows onto this dense syntactic structure. We verified the statistical significance of these graph-component contributions. Detailed paired significance test results (with p-values < 0.05 for all structural additions) are provided in Appendix G.

Impact of Training Components. Figure 5 validates the necessity of our multi-stage pipeline. **Stage 2 (cross-modal semantic alignment) is the most critical**: removing it reduces EA from 98.26% to 87.02%, confirming that explicit bridging is necessary to project graph-derived structure into the LLM token space as an effective soft-prefix; otherwise, the model cannot reliably leverage structural signals. Furthermore, removing specific objectives (e.g., w/o GTG) also degrades performance, indicating that both contrastive alignment and generative grounding are essential for translating structural cues into executable code.

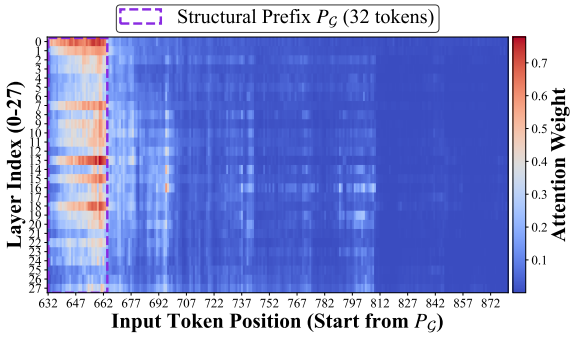

Figure 6: **Static attention during encoding.** Sustained attention from late-position tokens to the prefix confirms it acts as a **globally accessible anchor**.

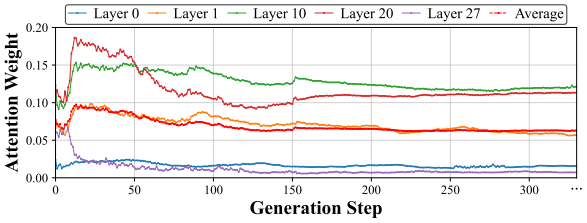

Figure 7: **Dynamic attention during decoding.** Consistent attention weights across generation steps indicate that the model **continuously refers** to structural priors.

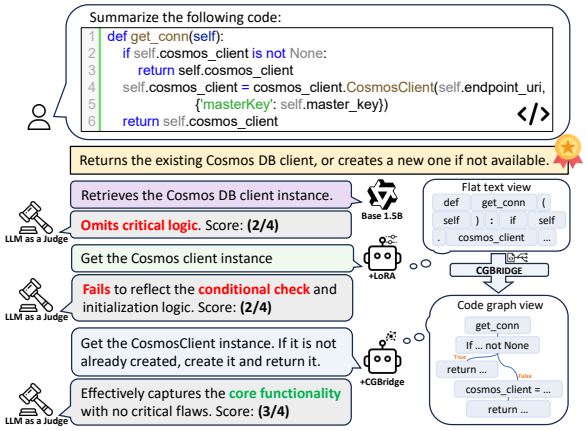

Figure 8: **Case study on capturing control flow logic.** While Base and LoRA models miss the critical lazy-initialization check (`if... is not None`), CGBRIDGE accurately captures this conditional logic.

5.7 Qualitative Analysis

To understand how CGBRIDGE utilizes the injected structural prefix during the encoding and generation stages, we analyze attention patterns following standard protocols (Vig, 2019), alongside a case study.

Structure is fused early and acts as a global anchor. Fig. 6 (Prompt Stage) visualizes attention from input text tokens to the structural prefix during encoding. Attention concentrates in lower-to-mid layers, suggesting that the model integrates structural constraints early in the processing pipeline. Crucially, non-trivial attention persists for late-position tokens (e.g., index > 740), indicating that the structural prefix remains a global context throughout long-context processing. We further provide a micro-level analysis of how distinct Bridge query tokens capture specific graph semantics (e.g., data vs. control flow) in Appendix M.1.

Structural guidance persists throughout decoding. Fig. 7 shows that the model maintains consistent attention to the prefix across decoding steps. This suggests the injected structure is not discarded but acts as a continuous reference. Layer-wise, mid layers attend steadily, while later layers show occasional peaks, indicating dynamic revisits to structural cues during token prediction.

CGBRIDGE captures fine-grained control flow. Fig. 8 illustrates a Python function involving lazy initialization, where capturing the conditional branch and return path is crucial for accuracy summary. While Base and +LoRA models produce fluent but incomplete summaries (missing key conditional logic), CGBRIDGE yields a faithful description consistent with the program's control flow. This confirms that explicit structural guidance enhances logic coverage, explaining the quantitative gains in summary fidelity. Additional success and failure cases are in Appendix M.2.

6 Conclusion

We proposed CGBRIDGE, a plug-and-play framework that bridges heterogeneous program structures and LLMs. Unlike approaches that rely on implicit structural priors or lengthy graph-as-text prompting, we treat the Code Property Graph as an independent modality and align it with the LLM's representation space via a lightweight, trainable Bridge module. Our three-stage alignment strategy enables frozen LLMs to leverage explicit structural anchors during inference. Experiments across multiple model families and scales confirm that CGBRIDGE boosts performance on code summarization and translation. Moreover, explicit structural anchoring mitigates the "code-as-text" brittleness and strengthens robustness to identifier renaming. By decoupling structural reasoning from text generation, CGBRIDGE offers a modular, efficient, and model-agnostic way to equip foundation models with deeper structure-aware code understanding.

Limitations

Although CGBRIDGE achieves strong performance, several limitations remain. First, our framework relies on static analysis tools such as Tree-sitter for CPG construction. It is therefore best suited to code understanding settings with syntactically parseable inputs, and is less applicable to severely malformed snippets in early-stage code completion, repair, or debugging scenarios. Second, the gains on short code snippets are relatively limited due to sparse structural signals (Appendix M.2), suggesting that CGBRIDGE is most beneficial for logic-heavy code with rich structural dependencies. Finally, although we adopt established automated metrics such as LLM-as-a-judge for scalable evaluation, human studies with developers would further strengthen the assessment of real-world utility. We leave the exploration of adaptive structural injection and more comprehensive human evaluations to future work.

References

Ziwei Chai, Tianjie Zhang, Liang Wu, Kaiqiao Han, Xiaohai Hu, Xuanwen Huang, and Yang Yang. 2023. Graphllm: Boosting graph reasoning ability of large language model. *Preprint*, arXiv:2310.05845.

Junkai Chen, Xing Hu, Zhenhao Li, Cuiyun Gao, Xin Xia, and David Lo. 2024a. Code search is all you need? improving code suggestions with code search. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Runjin Chen, Tong Zhao, Ajay Kumar Jaiswal, Neil Shah, and Zhangyang Wang. 2024b. Llaga: Large language and graph assistant. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: pre-training of

deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Matthias Fey, Jinu Sunil, Akihiro Nitta, Rishi Puri, Manan Shah, Blaz Stojanovic, Ramona Bendias, Alexandria Barghi, Vid Kocijan, Zecheng Zhang, Xinwei He, Jan Eric Lenssen, and Jure Leskovec. 2025. Pyg 2.0: Scalable learning on real world graphs. *ArXiv*, abs/2507.16991.

Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 855–864, New York, NY, USA. Association for Computing Machinery.

Alex Gu, Baptiste Rozière, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, Proceedings of Machine Learning Research, pages 16568–16621. PMLR / OpenReview.net.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Saizhuo Wang, Kun Zhang, Yuanzhuo Wang, Wen Gao, Lionel Ni, and Jian Guo. 2025. A survey on llm-as-a-judge. *Preprint*, arXiv:2411.15594.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 7212–7225. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Ashish Hooda, Mihai Christodorescu, Miltiadis Allamanis, Aaron Wilson, Kassem Fawaz, and Somesh Jha. 2024. Do large code models understand programming concepts? counterfactual analysis for code predicates. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: Evaluating the state of semantic code search. *Preprint*, arXiv:1909.09436.

Sarthak Jain and Byron C. Wallace. 2019. Attention is not Explanation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3543–3556, Minneapolis, Minnesota. Association for Computational Linguistics.

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. *Preprint*, arXiv:2507.16585.

Jinyang Li, Binyuan Hui, Reynold Cheng, Bowen Qin, Chenhao Ma, Nan Huo, Fei Huang, Wenyu Du, Luo Si, and Yongbin Li. 2023a. Graphix-t5: mixing pre-trained transformers with graph-aware layers for text-to-sql parsing. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'23/IAAI'23/EAAI'23.

Junnan Li, Dongxu Li, Silvio Savarese, and Steven C. H. Hoi. 2023b. BLIP-2: bootstrapping language-image pre-training with frozen image encoders and large language models. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202, pages 19730–19742. PMLR.

Junnan Li, Dongxu Li, Caiming Xiong, and Steven C. H. Hoi. 2022. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation. In *International Conference on Machine Learning*.

Qimai Li, Zhichao Han, and Xiao-Ming Wu. 2018. Deeper insights into graph convolutional networks for semi-supervised learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 3538–3545.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Jiahao Liu, Jun Zeng, Xiang Wang, and Zhenkai Liang. 2023. Learning graph-based code representations for source-level functional similarity detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 345–357.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Qizhe Shieh, and Wenmeng Zhou. 2025. 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, NAACL 2025 - Volume 1: Long Papers, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, pages 142–160. Association for Computational Linguistics.

Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. *J. Syst. Softw.*, 212(C).

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025. Repograph: Enhancing AI software engineering with repository-level code graph. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 3980–3990. Association for Computational Linguistics.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *Preprint*, arXiv:2009.10297.

Joshua David Robinson, Ching-Yao Chuang, Suvrit Sra, and Stefanie Jegelka. 2021. Contrastive learning with hard negative samples. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjing Wang, and Yu Sun. 2021. Masked label prediction: Unified message passing model for semi-supervised classification. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 1548–1554. ijcai.org.

Weisong Sun, Chunrong Fang, Yun Miao, Yudu You, Mengzhe Yuan, Yuchen Chen, Quanjun Zhang, An Guo, Xiang Chen, Yang Liu, and Zhenyu Chen. 2023. Abstract syntax tree for programming language understanding and representation: How far are we? *Preprint*, arXiv:2312.00413.

Hongyuan Tao, Ying Zhang, Zhenhao Tang, Hongen Peng, Xukun Zhu, Bingchang Liu, Yingguang Yang, Ziyin Zhang, Zhaogui Xu, Haipeng Zhang, Linchao Zhu, Rui Wang, Hang Yu, Jianguo Li, and Peng Di. 2025. Code graph model (cgm): A graph-integrated large language model for repository-level software engineering tasks. *Preprint*, arXiv:2505.16901.

Weixi Tong and Tianyi Zhang. 2024. Codejudge: Evaluating code generation with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 20032–20051.

Aäron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *CoRR*, abs/1807.03748.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Neural Information Processing Systems*.

Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. 2017. Convolutional neural networks over control flow graphs for software defect prediction. In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 45–52.

Jesse Vig. 2019. A multiscale visualization of attention in the transformer model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 37–42, Florence, Italy. Association for Computational Linguistics.

Yao Wan, Zhangqian Bi, Yang He, Jianguo Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Hai Jin, and Philip Yu. 2024. Deep learning for code intelligence: Survey, benchmark and toolkit. *ACM Comput. Surv.*, 56(12).

Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. 2023. Can language models solve graph problems in natural language? In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Peiding Wang, Li Zhang, Fang Liu, Chongyang Tao, and Yinghao Zhu. 2026. Codemem: Ast-guided adaptive memory for repository-level iterative code generation. *Preprint*, arXiv:2601.02868.

Ruiqi Wang, Jiyu Guo, Cuiyun Gao, Guodong Fan, Chun Yong Chong, and Xin Xia. 2025. Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering. *Proc. ACM Softw. Eng.*, 2(ISSTA):1955–1977.

Xin-Cheng Wen, Cuiyun Gao, Jiaxin Ye, Yichen Li, Zhihong Tian, Yan Jia, and Xuan Wang. 2024. Meta-path based attentional graph learning model for vulnerability detection. *IEEE Trans. Software Eng.*, 50(3):360–375.

Yang Wu, Yao Wan, Zhaoyang Chu, Wenting Zhao, Ye Liu, Hongyu Zhang, Xuanhua Shi, Hai Jin, and Philip S. Yu. 2024. Can large language models serve as evaluators for code summarization? *IEEE Transactions on Software Engineering*, 51:3205–3217.

Lingling Xu, Haoran Xie, Si-Zhao Joe Qin, Xiaohui Tao, and Fu Lee Wang. 2023. Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment. *Preprint*, arXiv:2312.12148.

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.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. 2020. Graph contrastive learning with augmentations. In *Advances in Neural Information Processing Systems*, volume 33, pages 5812–5823. Curran Associates, Inc.

Hao Zhang, Mengsi Lyu, Zhuo Chen, Xingrun Xing, Yulong Ao, and Yonghua Lin. 2025a. Pdtrim: Targeted pruning for prefill-decode disaggregation in inference. *arXiv preprint arXiv:2509.04467*.

Hao Zhang, Mengsi Lyu, Chenrui He, Yulong Ao, and Yonghua Lin. 2025b. Trimtokenator: Towards adaptive visual token pruning for large multimodal models. *arXiv preprint arXiv:2509.00320*.

Hao Zhang, Mengsi Lyu, Bo Huang, Yulong Ao, and Yonghua Lin. 2025c. Trimtokenator-lc: Towards adaptive visual token pruning for large multimodal models with long contexts. *arXiv preprint arXiv:2512.22748*.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. Bertscore: Evaluating text generation with BERT. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*.

Yilin Zhang, Xinran Zhao, Zora Zhiruo Wang, Chenyang Yang, Jiayi Wei, and Tongshuang Wu. 2025d. cast: Enhancing code retrieval-augmented generation with structural chunking via abstract syntax tree. *Preprint*, arXiv:2506.15655.

Yuhui Zhang, Yuichiro Wada, Hiroki Waida, Kaito Goto, Yusaku Hino, and Takafumi Kanamori. 2023. Deep clustering with a constraint for topological invariance based on symmetric infonce. *Neural Computation*, 35(7):1288–1339.

Ziyin Zhang, Hang Yu, Sage Lee, Peng Di, Jianguo Li, and Rui Wang. 2025e. Galla: Graph aligned large language models for improved source code understanding. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 13784–13802.

Jianan Zhao, Le Zhuo, Yikang Shen, Meng Qu, Kai Liu, Michael Bronstein, Zhaocheng Zhu, and Jian Tang. 2023. Graphtext: Graph reasoning in text space. *Preprint*, arXiv:2310.01089.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *Preprint*, arXiv:2206.08474.

Yun Zhu, Haizhou Shi, Xiaotang Wang, Yongchao Liu, Yaoke Wang, Boci Peng, Chuntao Hong, and Siliang Tang. 2025. Graphclip: Enhancing transferability in graph foundation models for text-attributed graphs. In *Proceedings of the ACM on Web Conference 2025*, WWW '25, page 2183–2197, New York, NY, USA. Association for Computing Machinery.

A Additional Backbone Results: DeepSeek-Coder

To further assess the generality of CGBRIDGE beyond the backbones reported in the main paper (Table 1), we additionally evaluate it on the DeepSeek-Coder family under the same datasets/splits, prompt templates, training protocol, and evaluation metrics as in Section 5.2. Table 6 summarizes the results on code summarization and translation. Overall, CGBRIDGE yields consistent improvements over the base model and text-only

Table 6: Results on DeepSeek-Coder under the same setup as Table 1.

Method	Code Summarization				Code Translation			
	ROUGE-L	SBCS	B-Score	LLM-J	CB	SM	DM	EA
Deepseek-coder-1.3B	6.12	45.30	79.08	1.63	48.82	67.67	55.58	51.37
+ LoRA	5.88	49.74	80.05	1.87	55.33	**69.85**	54.75	73.96
+ GraphText	14.76	48.76	81.86	1.94	55.02	60.39	54.22	69.31
+ CGBRIDGE	**16.30**	**50.44**	**84.02**	**2.61**	**55.48**	69.67	**55.44**	**76.18**
Deepseek-coder-6.7B	17.33	56.79	82.29	2.93	61.34	73.21	56.20	90.93
+ LoRA	21.11	58.21	84.30	2.97	64.40	73.12	70.67	93.88
+ GraphText	16.90	49.62	82.64	2.72	58.42	62.61	60.01	53.97
+ CGBRIDGE	**24.79**	**59.10**	**85.75**	**3.22**	**68.09**	**76.20**	**72.28**	**95.92**

Table 7: CRUXEval-O results (Pass@1, %) under settings with and without Stage 3 adaptation.

Model	Vanilla	+ CGBRIDGE (w/o Stage 3)*	+ CGBRIDGE (w/ Stage 3)*
CodeLlama-7B	36.4	39.3	**52.0**
DeepSeek-Coder-6.7B	41.3	43.8	**54.5**
Qwen2.5-Coder-7B	59.0	60.9	**64.9**
Qwen3-4B	36.0	38.4	**49.0**

* All reported improvements over the vanilla backbone are statistically significant under the corresponding paired tests ($p < 0.05$).

adaptation baselines, suggesting that the proposed structural anchoring transfers across different code LLM families.

B CRUXEval Results

We further evaluate CGBRIDGE on *CRUXEval-O*, the output-prediction task of CRUXEval, where the model predicts the program output given the program and its input. We report results both with and without Stage 3 adaptation. Specifically, the "w/o Stage 3" setting serves as a zero-shot evaluation without task-specific adaptation on CRUXEval-O, while the "w/ Stage 3" setting follows the same Stage 3 adaptation protocol as in the main downstream tasks. As shown in Table 7, CGBRIDGE consistently improves Pass@1 across all evaluated backbones, and Stage 3 adaptation yields additional gains.

C Cross-Language Generalization

To demonstrate that CGBRIDGE's structural anchoring is not limited to Python source inputs, we additionally evaluated our framework on a Java-to-Python translation task using the XLCoST benchmark. As shown in Table 8, applying CGBRIDGE to the Qwen2.5-Coder-7B backbone yields substantial improvements over the vanilla model. Specifically, it achieves a +11.73% increase in CodeBLEU and a +10.37% increase in Execution Accuracy. Both improvements are highly statistically significant. These results confirm that explicit structural graph grounding generalizes robustly across different source programming languages.

Table 8: Cross-language generalization results on the XLCoST (Java to Python) translation task using the Qwen2.5-Coder-7B backbone.

Method	CB	EA
Qwen2.5-Coder-7B	63.39	54.73
+ CGBRIDGE*	**75.12**	**65.10**

* Improvements in both metrics are highly statistically significant (CB: $p < 0.001$ via Wilcoxon signed-rank test; EA: $p < 0.001$ via McNemar's test).

Table 9: Results on Qwen3-4B instruct/thinking variants on XLCoST (Python to Java).

Model Variant	CB	EA
Qwen3-4B-Instruct	63.71	90.13
+ CGBRIDGE*	**74.27**	**95.58**
Qwen3-4B-Thinking	72.10	94.35
+ CGBRIDGE*	**74.31**	**95.89**

* All reported improvements are statistically significant under the corresponding paired tests ($p < 0.05$).

D Results on Instruct and Thinking Variants

To further examine the compatibility of CGBRIDGE with both standard instruction-following and reasoning-augmented decoding styles, we evaluate it on an instruct/thinking pair from the Qwen3-4B series on *XLCoST (Python to Java)*. As shown in Table 9, CGBRIDGE consistently improves both CodeBLEU and Execution Accuracy on the two variants. Notably, Qwen3-4B-Instruct + CGBRIDGE surpasses standalone Qwen3-4B-Thinking, suggesting that explicit structural modeling can provide benefits complementary to the backbone's reasoning capabilities.

E Extended Robustness Analysis

Table 10 reports the variable-renaming robustness results on the 1.5B backbone. The trend is consis-

Table 10: Robustness to variable renaming.

Method	Original	Obfuscated	Change
Qwen2.5-Coder-1.5B	2.70	2.59	-4.1%
+ LoRA	2.85	2.36	-17.2%
+ GraphText	2.66	1.90	-28.6%
+ **CGBRIDGE**	3.18	3.13	**-1.6%**
Qwen2.5-Coder-7B	2.78	2.71	-2.5%
+ LoRA	2.98	2.53	-15.1%
+ GraphText	2.96	2.13	-28.4%
+ **CGBRIDGE**	3.23	3.22	**-0.3%**

Table 11: Efficiency Analysis.

Method	Trainable Params (M)	Training Time (s)[a]	Inference Latency (ms)[b]
Qwen2.5-Coder-1.5B	-	-	∼202
+ LoRA	18.46	56.2	∼969
+ GraphText	-	-	∼598
+ **CGBRIDGE**[c]	180.80	67.5	∼**215**
Qwen2.5-Coder-7B	-	-	∼299
+ LoRA	43.12	110.5	∼1463
+ GraphText	-	-	∼1197
+ **CGBRIDGE**[c]	180.80	272.1	∼**371**

[a] Measured per epoch. [b] Measured per sample.
[c] Inference latency excludes offline graph construction (∼281 ms/sample) and embedding generation (∼0.42 ms/sample).

tent with the 7B setting in the main paper (Table 3): text-only adaptations suffer notable degradation (e.g., LoRA: -17.2%), while CGBRIDGE remains stable (-1.6%), suggesting stronger invariance to identifier-level perturbations. These results suggest that by leveraging explicit structural graphs, our framework captures true program logic and data flow, effectively resisting superficial identifier variations—a key weakness of text-only approaches.

F Extended Efficiency Results

In Section 5.5, we highlighted the efficiency and scalability of CGBRIDGE on the 7B model. Here, we further report efficiency on Qwen2.5-Coder-1.5B in Table 11. CGBRIDGE adds only negligible latency over the vanilla model (∼202→∼215ms), while being ∼4.5× faster than LoRA (∼969ms). In terms of parameter overhead, the fixed-size Bridge (180.80M) corresponds to ∼11.5% of a 1.5B backbone, confirming that the relative overhead decreases as the backbone scales. CGBRIDGE gains efficiency by injecting compact prefixes upfront, avoiding the context inflation of GraphText or the per-layer autoregressive decoding overhead of LoRA. This efficiency-oriented design is complementary to orthogonal efforts that improve inference efficiency through pruning (Zhang et al., 2025a) and token reduction in large language and multimodal models (Zhang et al., 2025b,c).

Table 12: Five-seed robustness results on XLCoST (Python to Java) with Qwen2.5-Coder-7B.

Model	CB (Mean±Std)	EA (Mean±Std)
Qwen2.5-Coder-7B	63.39	89.46
+ LoRA	71.17±0.37	97.21±0.34
+ **CGBRIDGE**	**73.91±0.16**	**98.26±0.21**

Table 13: Paired significance tests for graph-component ablations.

Comparison	CB (p-value)	EA (p-value)
AST + CFG vs. AST	0.003	0.019
AST + DFG vs. AST	2.0×10^{-13}	7.3×10^{-12}
CPG vs. AST + CFG	1.0×10^{-39}	1.3×10^{-8}
CPG vs. AST + DFG	1.2×10^{-19}	0.021
CPG vs. CFG + DFG	7.0×10^{-129}	2.5×10^{-15}

G Statistical Robustness and Significance Analysis

To assess whether the improvements of CGBRIDGE are robust to random initialization, we repeated the main translation experiments on Qwen2.5-Coder-7B over five random seeds and report the mean performance and standard deviation in Table 12. As shown, CGBRIDGE consistently outperforms LoRA while exhibiting lower variance across runs. We also conduct paired significance tests for the graph-component ablations in Table 5. The results in Table 13 show that the observed gains remain statistically significant even when the absolute margins are relatively small.

H Details of the CPG dataset Construction

We construct our dataset from two primary benchmarks: **CodeSearchNet** (Husain et al., 2020) (Python subset) for code summarization and **XL-CoST** (Zhu et al., 2022) (Python-to-Java) for code translation. To establish a foundation for our experiments, we refined and constructed a large-scale code graph dataset comprising approximately **270K samples**. This process involved two main steps: data refinement and graph construction.

H.1 Dataset Refinement.

To improve source data quality, we performed task-specific refinement. For **CodeSearchNet**, we removed inline comments and utilized GPT-4-0613 to refine the ground-truth summaries, reducing

Table 14: List of supported node types in the constructed CPGs.

$\mathcal{V}$	$\mathcal{T}^v$
Nodes	module, function_definition, identifier, parameters, default_parameter, none, comment, block, try_statement, if_statement, comparison_operator, expression_statement, assignment, call, argument_list, for_statement, integer, binary_operator, subscript, pattern_list, expression_list, while_statement, parenthesized_expression, string, string_start, string_content, string_end, elif_clause, else_clause, augmented_assignment, break_statement, continue_statement, return_statement, except_clause, finally_clause

Table 15: Taxonomy of fine-grained textual attributes for CPG edges.

$\mathcal{T}^e$	**Fine-grained Textual Attributes**
AST	has_name, has_parameters, has_body, has_condition, has_then_body, has_else_body, has_elif_branch, has_target, has_value, contains
CFG	sequential_execution, true_branch, false_branch, alternate_condition_branch, condition_evaluation, for_loop_body, for_loop, iteration_range, while_loop_body, while_loop_condition, try_block, exception_handler, finally_block, block_exit, loop_exit, loop_back, break_jump, condition_false_jump, function_call
DFG	contributes_to, flows_to

noise and aligning them better with the code logic. For **XLCoST**, we processed and formatted the code snippets to ensure syntactic correctness and executability, filtering out incomplete or unparseable fragments.

H.2 Graph Construction.

Following refinement, we construct a CPG for each snippet. Formally, the CPG is defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{T}^v, \mathcal{T}^e)$. Here, $\mathcal{V}$ denotes the set of nodes with a type mapping function $\phi : \mathcal{V} \to \mathcal{T}^v$, associating each node $v_i \in \mathcal{V}$ with a type $t_i^v \in \mathcal{T}^v$ (e.g., `identifier`). Similarly, $\mathcal{E}$ denotes the set of edges with a type mapping function $\psi : \mathcal{E} \to \mathcal{T}^e$, assigning each edge $e_{ij} = (v_i, v_j) \in \mathcal{E}$ a relation type $t_{ij}^e \in \mathcal{T}^e$ (e.g., `flows_to`).

Adopting established graph construction protocols (Liu et al., 2023; Guo et al., 2021; Chen et al., 2024a), we begin by parsing the source code using Tree-sitter[1]. Each graph node corresponds to a named AST element. As detailed in Table 14, these nodes are labeled with predefined types (*e.g.*, `assignment`, `if_statement`). For textual representation, high-level statement nodes retain only the first line of their source text (*e.g.*, the condition in an `if` statement) to reduce redundancy, while atomic nodes (*e.g.*, `expressions`, `identifiers`) retain their full span.

[1] https://tree-sitter.github.io/tree-sitter

Table 16: Structural statistics for the refined code graph datasets.

Statistic	**CodeSearchNet**	**XLCoST**
Total Samples	261,372	10,344
Avg. Nodes	125.41	145.77
Avg. AST Edges	124.41	144.77
Avg. CFG Edges	16.21	27.95
Avg. DFG Edges	22.85	32.23

Edges in the CPG encode structural and semantic relations. We categorize them into three types, with their fine-grained textual attributes comprehensively detailed in Table 15. AST edges capture syntactic hierarchy (*e.g.*, `has_name`, `contains`). CFG and DFG edges are both derived from the static analysis of the AST: CFG edges encode control-flow semantics (*e.g.*, `sequential_execution`, `true_branch`) by analyzing statement sequences and control structures, while DFG edges model data dependencies (*e.g.*, `flows_to`, `contributes_to`) based on variable definition and usage patterns. Each edge is also assigned a textual attribute indicating its semantic role.

Finally, all node and edge textual attributes are encoded into dense feature vectors using a pretrained code encoder (*e.g.*, UniXCoder) or LLM encoder, and the resulting graph is stored and processed using the PyG (Fey et al., 2025) for downstream GNN computation. Table 16 summarizes the scale and structural density of our curated datasets. The high number of structural edges relative to nodes highlights the richness of the logic captured by our CPGs.

I Detailed InfoNCE Loss Formulation

This section provides the detailed mathematical formulation of the symmetric InfoNCE loss used in Eq. 6 for Graph–Text Contrastive Learning (GTC). Given a batch of M graph–text pairs, the loss is

expanded as:

$$\mathcal{L}_{\text{GTC}} = -\frac{1}{2M}\sum_{i=1}^{M}\Bigg(\log\frac{\exp(s(B_Q^i, h_{\text{cls}}^i)/\tau)}{\sum_{j=1}^{M}\exp(s(B_Q^i, h_{\text{cls}}^j)/\tau)} + \log\frac{\exp(s(h_{\text{cls}}^i, B_Q^i)/\tau)}{\sum_{j=1}^{M}\exp(s(h_{\text{cls}}^i, B_Q^j)/\tau)}\Bigg), \tag{10}$$

where $B_Q^i \in \mathbb{R}^{N_q \times d_{\text{model}}}$ is the Bridge output (a set of N_q graph-query embeddings) for the i-th graph, and $h_{\text{cls}}^i \in \mathbb{R}^{d_{\text{model}}}$ is the corresponding `[CLS]` embedding from $H_{\mathcal{C}}^{(L_b)}$.

Similarity function. The similarity $s(\cdot,\cdot)$ is defined via the maximum cosine similarity over all query tokens:

$$s(B_Q^i, h_{\text{cls}}^j) = \max_{k\in\{1,\ldots,N_q\}} \cos\Big(B_{Q,k}^i,\ h_{\text{cls}}^j\Big), \tag{11}$$

where $B_{Q,k}^i \in \mathbb{R}^{d_{\text{model}}}$ denotes the k-th query embedding in B_Q^i and $\cos(\cdot,\cdot)$ is the cosine similarity. We use the same definition for $s(h_{\text{cls}}^i, B_Q^j)$. The temperature τ is learnable and scales similarity scores in Eq. 10.

J Implementation Details of Experiments.

We implement the CGE using a 2-layer Graph Transformer, with the Bridge Module utilizing $N_q = 32$ query tokens. The Bridge Module is initialized with the weights of a pre-trained BERT-base-uncased model (Devlin et al., 2019), adopting its original architecture, including the number of layers L_b. The initial node and edge features are encoded using UniXcoder-base (Guo et al., 2022). For Stage 1 self-supervised structural pretraining and Stage 2 cross-modal semantic alignment, we use the Python training split of *CodeSearchNet*. Specifically, Stage 1 uses the code-only portion, while Stage 2 uses the corresponding code-text pairs. For Stage 3 instruction-based task adaptation, we use the task-specific training splits, namely *CodeSearchNet* for code summarization and *XL-CoST* for code translation. For fair comparison, the LoRA baselines are trained on the same downstream training splits as Stage 3. We conduct experiments on a variety of open-source Code LLMs, including *Qwen2.5-Coder-Instruct* (1.5B, 7B) (Hui et al., 2024), *CodeLlama7b-Instruct-hf* (Rozière et al., 2024), and *Deepseek-coder-Instruct* (1.3B, 6.7B) (Guo et al., 2024). All experiments are performed on a server equipped with 8 NVIDIA H800 80GB GPUs.

J.1 The Choice of Graph Encoder.

CGBridge is designed to be **encoder-agnostic**: the CGE serves as a reusable structural expert that produces structural embeddings $H_{\mathcal{G}}$ for the Bridge, and thus can be instantiated with different graph encoders. In this work, we adopt a Graph Transformer (GT) (Shi et al., 2021) as the default CGE and compare it with representative message-passing GNNs (e.g., GCN and GAT).

Rationale. We choose GT as a strong default for three reasons. **(1) Generality and reusability.** We deliberately avoid highly specialized code-specific GNNs (e.g. MAGNET (Wen et al., 2024), TAILOR (Liu et al., 2023)) so that improvements can be attributed to the **structure-as-modality** paradigm and the Bridge-based alignment, and the CGE can be upgraded without changing the Bridge–LLM interface. **(2) Long-range dependencies.** Program structure contains long-distance relations (e.g., def–use and control dependencies) that shallow message passing may struggle to capture without deep stacking and potential over-smoothing; GT-style global attention provides a direct mechanism to model such interactions over structural tokens. **(3) Heterogeneity.** CPGs include multiple relation types (AST/CFG/DFG). GT integrates relation-aware signals to differentiate structural relations and flexibly weight their contributions.

Empirical Comparison. We conducted an ablation study comparing GT against standard baselines (GCN and GAT) on the Qwen2.5-Coder-1.5B backbone. As shown in Table 17, **all graph encoders yield performance gains** over the text-only baseline, demonstrating the general effectiveness of our CGBRIDGE framework in utilizing structural signals. Among them, the Graph Transformer achieves the most favorable results, supporting it as a stable default for aligning heterogeneous program graphs with LLM semantics. We emphasize that CGBRIDGE does not strictly depend on GT; the choice of encoder is orthogonal, and more specialized program-graph encoders can be seamlessly plugged into our framework.

K Hyperparameter Analysis and Settings

In this section, we first analyze the sensitivity of key hyperparameters to validate our design choices, and then provide the detailed configuration tables for reproducibility.

Table 17: Ablation study on the GNN backbone for the Code Graph Encoder (CGE), evaluated on the Qwen2.5-Coder-1.5B model.

GNN Type	Code Summarization				Code Translation			
	ROUGE-L	SBCS	B-Score	LLM-J	CB	SM	DM	EA
GCN	22.86	57.49	85.03	3.16	66.13	74.39	57.11	84.23
GAT	21.77	54.82	84.50	2.98	**72.09**	76.75	**70.48**	85.37
GT	**23.04**	**59.12**	**85.56**	**3.18**	69.81	**78.80**	68.55	**89.01**

K.1 Hyper-parameter Sensitivity Analysis

To validate our design and assess the robustness of CGBRIDGE, we study two key hyperparameters: the number of GNN layers L_g in the CGE and the number of Bridge queries N_q. Experiments are conducted on the code translation task (Qwen2.5-Coder-1.5B), using Execution Accuracy (EA) as the primary metric, and CodeBLEU (CB) as a complementary metric.

Analysis on the GNN Layers (L_g). L_g is crucial for capturing multi-hop dependencies. As shown in Figure 9 (left), performance peaks at $L_g = 2$. A single-layer GNN lacks capacity for capturing structural patterns, while deeper GNNs ($L_g > 2$) suffer from over-smoothing (Li et al., 2018) and redundancy in the heterogeneous code graph, as CFG and DFG edges already encode many shortcuts.

Analysis on the Bridge Queries (N_q). N_q governs the information capacity of the Bridge module in extracting structural knowledge. As illustrated in Figure 9 (right), with the Qwen2.5-Coder-1.5B backbone, EA peaks at $N_q = 32$, while CB remains relatively stable across different values. This indicates that $N_q = 32$ achieves a favourable trade-off, providing sufficient representation power while mitigating the noise introduced by excessive queries. A small N_q may limit information flow, whereas a large N_q risks signal dilution.

K.2 Detailed Experimental Configurations

Based on the analysis above and standard practices, our main experiment hyperparameter configurations are detailed in the tables below. We structure them by our three-stage training: Table 18 (Stage 1: Self-Supervised Structural Pretraining), Table 19 (Stage 2: Cross-Modal Alignment), and Table 20 (Stage 3: Instruction-Based Task Adaptation). All experiments, including ablation studies, adhere to these settings unless stated otherwise.

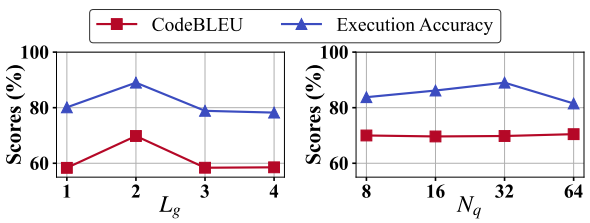

Figure 9: Analysis of the hyper-parameter N_q and L.

Table 18: Hyperparameters for Stage 1: Self-Supervised Structural Pretraining.

Parameter	Value
GNN Type	Graph Transformer
Input Dimension	768
Hidden Dimension	1024
Output Dimension	768
GNN Layers L_g	2
Attention Heads	4
Dropout Rate	0.1
Node Feature Drop Rate	0.05
Edge Drop Rate	0.05
Batch Size	128
Learning Rate	1e-5
Weight Decay	0.01
Contrastive Temperature τ_{cl}	0.3
Contrastive Loss Weight λ_{cl}	0.6
Edge Prediction Loss Weight λ_{edge}	0.4
Negative Sampling Ratio	0.5
Early Stopping Patience	20

L Prompt Templates

This section provides the exact instruction templates used during the instruction-based task adaptation and for LLM-as-a-judge evaluation.

L.1 Code summarization

The goal is to generate a high-quality PEP 257–compliant docstring for a given code snippet.

System Prompt: You are an expert Python assistant. Generate clear, concise, and accurate docstrings strictly following PEP 257 triple quote formatting.

User Prompt: Generate a Python docstring for the code below.
Format: PEP 257 triple quotes.

Table 19: Hyperparameters for Stage 2: Cross-Modal Alignment.

Parameter	Value
Initial Weight	BERT-base-uncased
Bridge layers L_b	12
Query Tokens N_q	32
Graph Embedding Dim	768
Model Dim d_{model}	768
Cross-Attention Frequency	2
Batch Size	16
Contrastive Temperature τ	0.07
Learning Rate	5e-7
Weight Decay	0.01
Warmup Ratio	0.01
Max Epochs	200
Scheduler Type	Plateau
Early Stopping Patience	20

Table 20: Hyperparameters for Stage 3: Instruction-Based Task Adaptation.

Parameter	Value
Bridge Module	Initialized from Stage 2
Precision	bfloat16
Batch Size	8
Learning Rate	1e-6
Weight Decay	0.01
Max Epochs	200
Warmup Ratio	0.01
Early Stopping Patience	20
Scheduler Type	Plateau
Scheduler Patience	2
Scheduler Factor	0.5
Min Learning Rate	1e-10
Temperature	0
Repetition Penalty	1.1

Required Structure:
- One-line summary of the function. (Optional) More detailed explanations are provided if the logic is complex.
- Parameters: param_name (param_type): Description of parameter. (Use 'None' if no parameters)
- Returns: return_type: Description of returned value. (Use 'None' if no explicit return)
Code:

L.2 Code Translation

The goal is to translate source functions into functionally equivalent, runnable target code.

System Prompt:
Role: Python to Java Translator.
Output: Single, runnable Java class (e.g., `Solution` or `Main`).
Structure:
1. Core logic in a `public static` method.
2. `public static void main(String[] args)`: Calls static method (use user's sample inputs if given, else defaults) and prints output via `System.out.println()`.
Ensure:
- Functional Python-Java equivalence.
- Correct Java types and standard library mapping.
- Necessary `import`s.
- Compilable and runnable code.

User Prompt:
Translate the following Python code to Java.

L.3 LLM as a Judge

For reference-free evaluation of code summarization, we use GPT-4-0613 as the LLM evaluator consistently across all compared methods (Wu et al., 2024; Wang et al., 2025; Gu et al., 2025; Tong and Zhang, 2024). The following prompts guide the LLM to score the generated summaries based on predefined criteria, with a heavy emphasis on functional consistency.

System Prompt:
You are an expert Principal Software Engineer acting as a meticulous Code Reviewer. Your sole task is to provide a critical and objective evaluation of a candidate code summary based on the provided source code.
Your evaluation must follow these steps: - Carefully read the source code to fully understand its functionality, inputs, outputs, and key logic.
- Critically analyze the candidate summary against the code.
- Provide a structured evaluation based on the four dimensions below.
Evaluation Dimensions:
- Coherence (0-4): How logically organized and well-structured is the summary? Does it form a coherent description of the code? (0=Incoherent, 4=Perfectly coherent).
- Consistency (0-4): Does the summary accurately reflect the code's functionality and logic? Are there any factual errors or hallucinations? This is the most critical dimension. (0=Contradicts the code, 4=Perfectly consistent).
- Fluency (0-4): Is the summary written in clear, natural, and grammatically correct language?

(0=Unreadable, 4=Perfectly fluent).
- Relevance (0-4): Does the summary capture the essential information without including redundant or trivial details? (0=Irrelevant, 4=Perfectly relevant).
Overall Score:
After rating the four dimensions, provide a holistic Overall Score (0-4). This score is NOT a simple average. You must weigh Consistency most heavily, as an inconsistent summary is fundamentally flawed, regardless of its fluency.

User Prompt:
Please evaluate the following code summary.
Source Code:
Candidate Summary:
{candidate}

M Extended Qualitative Analysis

M.1 Interpreting Bridge Query Tokens (Micro-level Alignment)

To obtain a fine-grained view of what Bridge queries encode, we inspect the cross-attention from query tokens to the structural embeddings inside the Bridge on the case-study instance in Fig. 8. For each query q_k, we extract its cross-attention weights over graph nodes from the (last) Bridge layer and average them over attention heads. We then aggregate these weights by CPG node type (e.g., `if_statement`, `return_statement`) to compute a per-type attention share:

$$p_{k,t} = \frac{\sum_{i:\,\mathrm{type}(v_i)=t} A_{k,i}}{\sum_i A_{k,i}}, \qquad (12)$$

where $A_{k,i}$ denotes the (head-averaged) attention weight from q_k to node v_i. Table 21 reports the top-attended node types for representative queries. While strict one-to-one correspondences are uncommon due to distributed representations, we consistently observe **dominant** query-to-type patterns that are semantically plausible (e.g., control-flow cues vs. return-path cues). We emphasize that attention provides a diagnostic signal rather than a definitive causal explanation (Jain and Wallace, 2019).

M.2 Additional Cases

To provide a holistic view of CGBRIDGE's capabilities and boundaries, we present detailed case studies covering both success and failure scenarios across Code Summarization and Code Translation tasks. These examples are selected to empirically validate our core hypothesis: structural information (AST, CFG, DFG) acts as an anchor for complex algorithmic logic, but may offer diminishing returns in short, linear, or dependency-heavy contexts. For clarity, comments within the displayed code snippets (e.g., highlighting errors or logic flow) were added manually for illustrative purposes and were not part of the raw model outputs.

M.2.1 Analysis of Code Translation

In Code Translation, CGBRIDGE better preserves deep algorithmic structures. The success case (or Figure 8 in Section 5.7 or Table 22) shows how CFG/DFG constraints enable the model to reconstruct complex recursive logic (e.g., linked list subtraction) where baselines suffer from task hallucination. However, we also identify two critical failure modes (Tables 23 and 24): (1) Metric Misalignment: CGBRIDGE may receive lower n-gram scores (e.g., CodeBLEU) due to valid code refactoring, despite semantic correctness. (2) Graph Incompleteness: The reliance on static analysis implies that if external dependencies (e.g., library imports) are not resolved in the CPG, the model may fail to generate necessary 'import' statements. Furthermore, our analysis highlights that execution success on isolated test cases does not always equate to semantic correctness, as demonstrated by the baseline passing through mathematical coincidence despite incorrect logic.

M.2.2 Analysis of Code Summarization

In Code Summarization, we observe that CGBRIDGE excels in scenarios involving complex control flows. As shown in the success case (Table 25), the model leverages CFG connections to accurately describe exception handling and fallback logic, which baseline models (and occasionally ground-truth references) often miss. Conversely, failure cases (Table 26) reveal a limitation in handling extremely short code snippets. When structural signals are sparse (e.g., linear "one-liners"), graph priors provide little advantage over strong variable naming conventions, occasionally leading to over-generic summaries compared to baselines that attend closely to surface text.

Table 21: Micro-level interpretability of Bridge queries via cross-attention mass to CPG node types on the case study in Fig. 8. Percentages are computed by aggregating cross-attention weights from a query token to all graph nodes of a given type (averaged over heads/layers) and normalizing across types.

Query Token	Top-attended Nodes	Interpretation
Query #7	if_statement (41%)	Focuses on control flow (`if self.cosmos_client is not None:`)
Query #25	assignment, call (38%)	Focuses on data flow (`...= CosmosClient(...)`)
Query #31	return_statement (45%)	Focuses on the function's return path (`return self.cosmos_client`)

Table 22: **Good Case (Translation).** Comparison on Linked List Subtraction. CGBRIDGE captures the correct recursive logic, while the Baseline hallucinates a completely different task (List Reordering).

Input (Python)

```python
# ... Node class omitted ...

# Helper for recursive logic
def subtractLinkedListHelper(l1, l2):

    global borrow
    # Base case
    if l1 is None and l2 is None and
        not borrow:
        return None

    # Recursive step matches
       structure
    previous =
       subtractLinkedListHelper(
        l1.next if l1 else None,
        l2.next if l2 else None
    )

    d1 = l1.data
    d2 = l2.data

    # Borrow logic
    if borrow:
        d1 -= 1
        borrow = False
    if d1 < d2:
        borrow = True
        d1 += 10

    sub = d1 - d2
    current = newNode(sub)
    current.next = previous
    return current

def subtractLinkedList(l1, l2):
    # ... (Pad zeros logic omitted)
       ...
    return subtractLinkedListHelper(
       lNode, sNode)

# ... main omitted ...
```

Reference (Java)

```java
// ... Node class omitted ...

Node subtractLinkedListHelper(Node
       l1, Node l2) {
    if (l1 == null && l2 == null &&
       borrow == false)
        return null;

    // Reference recursion
    Node previous =
       subtractLinkedListHelper(
        (l1 != null) ? l1.next :
       null,
        (l2 != null) ? l2.next :
       null
    );

    int d1 = l1.data;
    int d2 = l2.data;

    // Logic exactly matches Python
    if (borrow) {
        d1--;
        borrow = false;
    }
    if (d1 < d2) {
        borrow = true;
        d1 = d1 + 10;
    }

    Node current = new Node(d1 - d2)
       ;
    current.next = previous;
    return current;

// ... main omitted ...
}
```

CGBRIDGE (Ours)

```java
// ... Node class omitted ...

static Node subtractLinkedListHelper
       (Node l1, Node l2) {
    if (l1 == null && l2 == null &&
       !borrow)
        return null;

    // CGBridge correctly generates
    // the recursive calls
    Node previous =
       subtractLinkedListHelper(
        (l1!=null)? l1.next:null,
        (l2!=null)? l2.next:null
    );

    int d1 = l1.data;
    int d2 = l2.data;

    // Correctly handles borrow
       state
    if (borrow) {
        d1 -= 1;
        borrow = false;
    }
    if (d1 < d2) {
        borrow = true;
        d1 += 10;
    }

    Node current = newNode(d1 - d2);
    current.next = previous;
    return current;
// ... main omitted ...
}
```

CB: 55.55 **Exec: ✓**

Baseline

```java
// ... Node class omitted ...

public class Main {
    static Node left = null;

    // HALLUCINATION:
    // Generates "Reorder List"
       logic
    // instead of Subtraction!
    public static void rearrange(
       Node head) {
        left = head;
        reorderListUtil(left);
    }

    public static void
       reorderListUtil(Node right) {
        if (right == null) return;
        reorderListUtil(right.next);

        // This logic folds the list
        // (L0->Ln->L1...)
        // Completely unrelated to
       subtraction
        if (left != right && left.
       next != right) {
            Node temp = left.next;
            left.next = right;
            right.next = temp;
            left = temp;
        }
        // ...
    }
// ... main omitted ...
}
```

CB: 22.29 **Exec: ✗**

Analysis: The task requires implementing *Linked List Subtraction*. CGBRIDGE (Column 3) successfully aligns with the Python reference, correctly implementing the `subtractLinkedListHelper` with recursive borrow logic. In stark contrast, the Baseline (Column 4) fails to identify the task semantics, hallucinating a `reorderListUtil` function (likely retrieving a solution for "Reorder List") which is functionally irrelevant.

Why CGBRIDGE Succeeds: It leverages explicit structural anchors—**AST** preserves the recursive skeleton, **CFG** enforces the correct execution order (length alignment → recursive calls), and **DFG** maintains critical data dependencies for the global `borrow` state. These graph priors constrain the generation to the correct subtraction algorithm.

Metric Consistency: The significant gap in CodeBLEU (55.55 vs. 22.29) quantitatively confirms the Baseline's severe hallucination. While the Baseline collapses due to task divergence, CGBRIDGE achieves a robust score. Notably, CGBRIDGE's score is capped at 55.55 due to **valid code refactoring** rather than errors: it uses a static `borrow` field and a `newNode` factory, differing from the reference's instance fields and constructors. These stylistic choices reduce n-gram matching but preserve perfect functional equivalence.

Table 23: **Failure Case #1 (Translation).** An example of metric misalignment. CodeBLEU penalizes valid code refactoring. CGBRIDGE generates a concise one-liner, while the Reference uses a verbose two-step assignment. The metric penalizes CGBRIDGE for missing the intermediate variable tokens, despite the logic being identical.

Input (Python)	Reference (Java)	CGBRIDGE (Ours)	Baseline

`def CountWays(n):`
`  ans = (n - 1) // 2`
`  return ans`
 	
`N = 8`
`print(CountWays(N))`
	`class GFG {`
	`  static int CountWays(int n) {`
	`  // Verbose style`
	`  int ans = (n - 1) / 2;`
	`  return ans;`
	`  }`
	`  // ... main omitted`
	`}`
		`public class Main  {`
		`  static int CountWays(int n) {`
		`  // Concise Refactoring`
		`  return (n - 1) / 2;`
		`  }`
		`  // ... main omitted`
		`}`
		
		CB: 67.13
		Execution Accuracy: ✓
			`public class Main {`
			`  public static int countWays(int n) {`
			`  // Matches Verbosity`
			`  int ans = (n - 1) / 2;`
			`  return ans;`
			`  }`
			`  // ... main omitted`
			`}`
			
			CodeBLEU: 71.67
			Execution Accuracy: ✓

Analysis: CGBRIDGE (Column 3) produces a functionally correct but *concise* translation, collapsing the reference's two-step body (`int ans = ...; return ans;`) into a single expression. CodeBLEU penalizes this refactoring because it reduces n-gram overlap (missing tokens `int`, `ans`, `=`) and simplifies the AST. In contrast, the Baseline (Column 4) scores higher simply because it mimics the verbose style of the Reference. This demonstrates how surface-level metrics can discourage efficient coding styles.

Table 24: **Failure Case #2 (Translation).** An example of graph incompleteness (missing import) in CGBRIDGE, and a **False Positive** execution in the Baseline caused by task hallucination. The specific input array is critical to explaining the baseline's lucky pass.

Input (Python)	Reference (Java)	CGBRIDGE (Ours)	Baseline

`def maxPossibleSum(arr, N):`
`  arr.sort() # Ascending`
`  total_sum = 0`
`  j = N - 3`
`  while j >= 0:`
`  total_sum += arr[j]`
`  j -= 3`
`  print(total_sum)`
 	
`if __name__ == "__main__":`
`  arr = [7, 4, 5, 2, 3, 1, 5, 9]`
`  N = 8`
`  maxPossibleSum(arr, N)`
	`import java.io.*;`
	`import java.util.*;`
	
	`class GFG {`
	`  public static void`
	`  maxPossibleSum(int[] arr, int N) {`
	`  Arrays.sort(arr);`
	`  int sum = 0;`
	`  int j = N - 3;`
	`  while (j >= 0) {`
	`  sum += arr[j];`
	`  j -= 3;`
	`  }`
	`  System.out.println(sum);`
	`  }`
	
	`  public static void main(String[] args) {`
	`  int[] arr = {7, 4, 5, 2, 3, 1, 5, 9};`
	`  int N = arr.length;`
	`  maxPossibleSum(arr, N);`
	`  }`
	`}`
		`// MISSING IMPORT`
		`class GFG {`
		`  static void maxPossibleSum(int arr[], int N) {`
		`  // Compile Error:`
		`  // cannot find symbol '`
		`  Arrays'`
		`  Arrays.sort(arr);`
		`  int total_sum = 0;`
		`  for (int j = N - 3; j >= 0; j -= 3) {`
		`  total_sum += arr[j];`
		`  }`
		`  System.out.println(total_sum);`
		`  }`
		
		`  public static void main(String[] args) {`
		`  int arr[] = new int[]{7, 4, 5, 2, 3, 1, 5, 9};`
		`  int N = 8;`
		`  maxPossibleSum(arr, N);`
		`  }`
		`}`
		
		CodeBLEU: 46.75
		Execution Accuracy: ✗
			`import java.util.*;`
			
			`public class Main {`
			`  public static void main(String[] args) {`
			`  // Same input data`
			`  Integer arr[] = new Integer[]{7, 4, 5, 2, 3, 1, 5, 9};`
			`  int N = 8;`
			
			`  maxPossibleSum(arr, N);`
			`  }`
			
			`  public static void maxPossibleSum(Integer arr[], int N){`
			`  // HALLUCINATION: Descending!`
			`  Arrays.sort(arr, Collections.reverseOrder());`
			`  int total_sum = 0;`
			`  for (int j=N-3; j>=0; j-=3) {`
			`  total_sum += arr[j];`
			`  }`
			`  System.out.println(total_sum);`
			`  }`
			`}`
			
			CodeBLEU: 40.40
			Execution Accuracy: ✓*

Analysis: CGBRIDGE (Column 3) fails execution due to a missing `import`. The Baseline (Column 4) is marked with ✓* to indicate a false positive. It hallucinates a `reverseOrder` sort (violating the Python semantics). However, it passes this specific test case because the input array `[7, 4, 5, 2, 3, 1, 5, 9]` creates a mathematical coincidence: the indices summed (2 and 5) correspond to values (3 and 5 in ascending sort, 5 and 3 in descending sort) that sum to the same result (8). This demonstrates that execution success on isolated or limited test cases does not strictly guarantee semantic correctness, highlighting the importance of structural alignment.

Table 25: **Good Case (Summarization).** CGBRIDGE leverages CFG information to capture the exception handling logic, whereas the Baseline misses the fallback path.

Input Code (Python):

```python
def _dag_file_to_dict(self):
    try:
        dag_file = open(os.path.join(self.workdir, ".treeDag.json"))
        dag_json = json.load(dag_file)
    except (FileNotFoundError, json.decoder.JSONDecodeError):
        # Capturing this exception logic is the key challenge
        logger.warning(colored_print(
            "WARNING: dotfile named .treeDag.json not found or corrupted",
            "red_bold"))
        dag_json = {}
    return dag_json
```

Reference	CGBRIDGE (Ours)	Baseline
Function that opens the dotfile named .treeDag.json in the current working directory.	Convert the dag file to a dictionary. **If the file is not found or corrupted, return an empty dict.** **ROUGE-L:** 12.50 **BERTScore:** 84.99 **LLM-as-a-Judge: 3.75**	Converts the contents of a .treeDag.json file to a dictionary. **ROUGE-L: 25.00** **BERTScore: 90.59** **LLM-as-a-Judge:** 3.0

Analysis: This case highlights a **reference bias** where standard metrics penalize semantic completeness. The ground-truth Reference describes only the successful execution path ("opens the dotfile"), omitting the critical exception handling logic visible in the code's `try-except` block. The Baseline mimics this omission, achieving high ROUGE/BERTScore (25.00/90.59) simply by copying the specific string literal ".treeDag.json". In contrast, CGBRIDGE abstracts the filename to "dag file" (lowering lexical overlap scores to 12.50) but uniquely captures the control-flow semantics: **"If the file is not found... return an empty dict."** The **LLM-as-a-Judge** correctly identifies this superiority (3.75 vs 3.0), recognizing that CGBRIDGE provides a more faithful description of the function's actual behavior, even if it deviates from the reference's phrasing.

Table 26: **Failure Cases for Code Summarization.** In extremely short code snippets with sparse structural signals, CGBRIDGE tends to generate generic summaries compared to the Baseline, as the graph priors offer limited gain over strong naming conventions.

Input Code	Reference	CGBRIDGE (Ours)	Baseline
Case #1: One-liner (Loss of Specificity)			
`def filter_dict_by_key(d, keys):` `  return {k: v for k, v in d.items() if k in keys}`	Filter the dict *d* to remove keys not in *keys*.	Filter a dictionary by keys. **ROUGE-L:** 26.67 **BERTScore: 87.09** **LLM-as-a-Judge:** 3.0	Filter a dictionary by **keeping only the keys** that are in a given list. **ROUGE-L: 33.33** **BERTScore:** 86.86 **LLM-as-a-Judge: 3.5**
Case #2: Linear Control Flow (Dominance of Naming Priors)			

`def create_secret_link(self, title, description=None, expires_at=None):`
`  self.link = SecretLink.create(`
`  title,`
`  self.receiver,`
`  extra_data=dict(recid=self.recid),`
`  description=description,`
`  expires_at=expires_at,`
`  )`
`  return self.link`
	Create a secret link from request.	Creates a secret link.
		ROUGE-L: 80.00
		BERTScore: 88.88
		LLM-as-a-Judge: 3.5
			Creates a secret link **for the current user**.
			ROUGE-L: 57.14
			BERTScore: 90.71
			LLM: 3.75

Analysis: These cases illustrate the trade-off between **faithfulness** and **informativeness** in sparse contexts.

Case #1: CGBRIDGE generates a concise summary ("Filter... by keys"), achieving a high BERTScore (87.09) as it captures the general intent. However, it fails to specify the filtering direction ("keeping only" vs "removing"), whereas the Baseline (ROUGE 33.33) explicitly captures this whitelist logic, albeit with more verbose phrasing.

Case #2: CGBRIDGE relies heavily on the function name, producing a faithful but generic summary ("Creates a secret link"). This results in a very high ROUGE (80.00) due to exact token overlap with the reference. In contrast, the Baseline infers implicit context ("for the current user") from the `self.receiver` argument. Although this lowers its ROUGE score (57.14), the **LLM-Judge** prefers this informative detail (3.75 vs 3.5), highlighting that graph priors (which drive CGBRIDGE) offer limited gain in simple linear code compared to semantic inference.