SASU: Structure-Anchored Semantic Unification for Code Search

Qian Zhang, Hao Li, and Hao Jiang*
Wuhan University

Abstract

Code search aims to retrieve code snippets according to natural-language intent. Despite the progress of pretrained code models, there remains a gap between natural language and programming language: descriptions are usually sequential and lexical, while code is governed by syntax, data dependencies, and execution logic. Existing retrieval models often rely on token-level matching or use structure as auxiliary information, which limits their ability to capture executable behavior under lexical variation or incomplete code evidence. In this paper, we propose SASU, a structure-anchored semantic unification framework for code search. SASU extracts a normalized multi-relational graph for each function and composes syntax, data-flow, control-flow, call, return, and exception relations into a unified structural view. The structural representation is then used to guide node-level lexical evidence selection before code and text are aligned in the same retrieval space. Experiments on CodeSearchNet show that SASU achieves 0.835 MRR for code-to-text retrieval and 0.840 MRR for text-to-code retrieval across Python, Java, and JavaScript. Under 40–60% node deletion, SASU retains 86.6% of its full-graph MRR, indicating that structure provides useful retrieval signals beyond lexical pretraining.

1 Introduction

Code search is a practical task in neural code intelligence, where a system retrieves code snippets according to a natural-language query (Cambronero et al., 2019; Husain et al., 2019). Recent pretrained code models have improved this task by learning token-level representations shared between code and natural language from large-scale corpora (Feng et al., 2020; Wang et al., 2021, 2023). These advances provide strong lexical and semantic encoders for code search.

Beyond token-level code–text representation learning, structure-aware methods introduce program graphs to capture syntax and semantic dependencies in code. GraphCodeBERT incorporates data-flow information into pretrained code representations (Guo et al., 2021), and UniXcoder further unifies code, comments, and structural signals in one representation model (Guo et al., 2022). Graph neural networks have been used to propagate information over program graphs (Allamanis et al., 2018). Program-graph representations have also been studied for functional similarity modeling (Liu et al., 2023). Recent repository-level systems further use code context graphs or semantic graphs to retrieve relevant program context (Liu et al., 2024). RepoHyper expands and refines retrieval over semantic graphs for repository-level completion (Phan et al., 2025).

However, in many existing retrieval systems, structural information still plays a relatively limited role. Program graphs are often incorporated as auxiliary features around token representations, while the final retrieval behavior remains largely dominated by lexical similarity. As a result, structural evidence that reflects execution behavior or semantic dependencies may not be fully captured during code–text alignment. These limitations suggest that code search requires representations that can better preserve structural semantics beyond lexical overlap, especially when lexical evidence is weak or incomplete. Figure 1 illustrates this motivation.

To address these limitations, we propose SASU, a Structure-Anchored Semantic Unification framework for code search. Concretely, SASU 1) represents each function with UNIS-GRAPH, a normalized multi-relational graph that composes AST, DFG-style value relations, and CFG-style execution relations; 2) encodes the graph with relation-aware propagation and uses the structural summary to guide lexical evidence selection; and 3) trains the model with structural consistency and

*Corresponding author.

Findings of the Association for Computational Linguistics: IJCNLP-AACL 2026, pages 3724–3735
November 6-10, 2026 ©2026 Association for Computational Linguistics

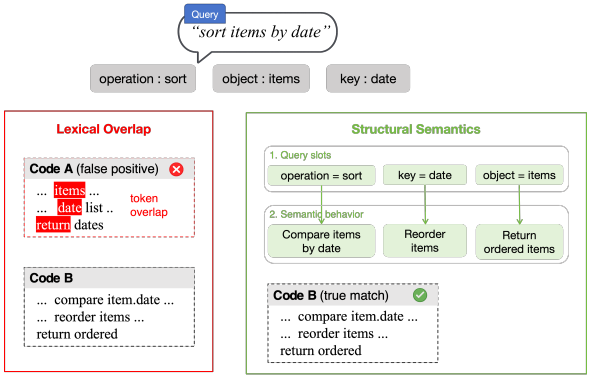


Figure 1: Motivation example for structure-anchored code search. Lexical overlap may retrieve a false positive, whereas structural semantics links query slots to comparison, ordering, and return behavior to identify the true semantic match.

code–text contrastive objectives. We evaluate SASU on CodeSearchNet across Python, Java, and JavaScript, including overall retrieval, language-level retrieval, incomplete-graph robustness, and component ablations.

In summary, our contributions are:

- We propose SASU, a structure-anchored code search framework that uses executable program structure to guide lexical evidence selection for code–text retrieval.
- We introduce UNIS-GRAPH, a normalized multi-relational program graph that composes syntax, data-flow, control-flow, call, return, and exception relations into a unified structural view.
- We design a two-stage training scheme that combines structural consistency pretraining with graph–text contrastive alignment, enabling the model to learn retrieval representations from both complete and perturbed graph views.

2 Related Work

2.1 Neural Code Search and Code Embedding

Code search has been studied from sparse lexical matching to neural code–text retrieval. Earlier systems match user intent against program text or signatures (Hill et al., 2011), while later neural models learn shared representations for code and natural-language queries (Cambronero et al., 2019). Multimodal retrieval further combines several code views to improve matching (Gu et al., 2021). Recent embedding-oriented work scales code-retrieval supervision through large synthetic datasets (Li et al., 2025a). Such methods are effective for semantic matching, but their retrieval behavior is still often shaped by identifiers, comments, and local token patterns. This motivates code representations that can use lexical evidence while being less dependent on surface overlap.

2.2 Structure-aware Code Representation

Program structure provides another important source of code semantics. Tree-based models encode AST information instead of treating code as a flat token sequence (Mou et al., 2016). Code2vec represents programs through sampled AST paths (Alon et al., 2019). Inst2vec learns statement representations from contextual relations in intermediate code (Ben-Nun et al., 2018). Later work explores unified AST representations for cross-language program classification (Wang et al., 2022). Recent graph-based retrieval models combine AST encoders with Transformer representations to capture both hierarchical and sequential code features (Ye et al., 2025). These studies show that structure can complement token-based representations, but they do not fully address how executable structure should guide lexical evidence selection during code–text alignment.

2.3 Graph-based Retrieval

Graph representations are also used in retrieval settings because they expose dependencies that may be weakly reflected by token sequences. Repository-level retrieval systems use commit histories and neural reranking to identify files relevant to repository questions or bug reports (Gandhi et al., 2025). CoRet integrates code semantics, repository structure, and call-graph dependencies for code-editing retrieval (Fehr et al., 2025). GraphCodeAgent constructs a requirement graph and a structural-semantic code graph to retrieve repository context for code generation (Li et al., 2025b). These studies suggest that structural evidence is useful for retrieval, but existing retrieval-oriented methods often use structure as additional context, auxiliary signal, or reranking evidence. They less directly study how executable structure can anchor the code representation itself and guide lexical evidence selection for code–text alignment.

3 SASU Framework

3.1 Overview

In this paper, we leverage both structural program graphs and pretrained lexical encoders to obtain code representations that are informative for natural-language retrieval and robust to incomplete code evidence. The overall process of SASU is shown in Figure 2. It consists of three stages as follows.

Unified Structure Extraction. We analyze each function-level code block and extract a normalized graphical view to better represent its syntax and executable behavior. For each code block c_i in the code corpus $\mathcal{D}$, this process can be formulated as

$$G_i \leftarrow \mathrm{Extract}(c_i),$$
$$\mathcal{G}.\mathrm{append}(\langle c_i, G_i \rangle),$$

where G_i denotes the extracted multi-relational graph of c_i. The graph contains normalized nodes and typed edges that describe syntax, data flow, control flow, calls, returns, and exception flow. Detailed extraction rules are given in Appendix A.

Structure-Anchored Code Encoding. Given the extracted graph, we encode its structural and lexical information into a unified code vector. The structural branch first summarizes the graph through relation-aware propagation and graph pooling. Then the structural summary is used to query node-level lexical features, so that token evidence is selected under the guidance of program structure:

$$\mathbf{s}_i \leftarrow \mathrm{GraphEncoder}(G_i),$$
$$\mathbf{z}_{c_i} \leftarrow \mathrm{Fuse}(\mathbf{s}_i, X_i),$$

where X_i denotes the node lexical features of G_i, $\mathbf{s}_i$ is the structural representation, and $\mathbf{z}_{c_i}$ is the final code representation.

Code–Text Alignment. For a natural-language description t_i, we obtain its representation with a pretrained text encoder and projection head:

$$\mathbf{z}_{t_i} \leftarrow \mathrm{TextEncoder}(t_i).$$

The model is optimized with two complementary contrastive objectives. The first objective preserves structural consistency between the full graph and its perturbed view. The second objective aligns the code representation and the text representation in the same retrieval space. During inference, retrieval is conducted by computing the cosine similarity between the L2-normalized vectors $\mathbf{z}_c$ and $\mathbf{z}_t$.

Unified type	Python	Java	JavaScript
FOR_LOOP	For	ForStatement	ForStatement
IF_STMT	If	IfStatement	IfStatement
FUNCTION_DEF	FunctionDef	MethodDeclaration	FunctionDeclaration
FUNCTION_CALL	Call	MethodInvocation	CallExpression
RETURN_STMT	Return	ReturnStatement	ReturnStatement

Table 1: Examples of type normalization in UNIS-GRAPH.

3.2 UniS-Graph Representation

In this section, we discuss how to extract informative graphical views for function-level code blocks. To capture both lexical semantics and executable logic, we construct UNIS-GRAPH by composing normalized syntax structure with data-flow, control-flow, call, return, and exception relations. For each function, the extracted graph is formulated as

$$G = (V, E, R, X),$$

where V is the node set, E is the edge set, R is the relation-type vocabulary, and X contains initial node features. After obtaining the graph, we use it as the structural view of the code block for retrieval representation learning.

AST normalization. An abstract syntax tree provides the basic syntactic organization of source code. However, raw AST node names are language-specific and cannot be directly compared across languages. We therefore normalize parser outputs into a shared node vocabulary. As shown in Table 1, language-specific constructs with the same program role are mapped to the same unified type. The normalized nodes are divided into three groups: syntactic nodes, symbol nodes, and semantic-role nodes. Syntactic nodes describe statements and expressions; symbol nodes keep identifiers, attributes, operators, and literals; semantic-role nodes mark function entry, function exit, returns, branch predicates, call sites, and exception handlers. This normalization preserves the useful lexical content of the source program while making the structural roles comparable across Python, Java, and JavaScript.

Multi-relational edges. The AST edges alone only describe parent-child containment. To represent the semantic and logical information of code blocks, we add data-flow and control-flow relations on top of the normalized syntax graph. The data-flow view describes how values are defined, propagated, used, and returned. The control-flow view describes the execution order of statements,

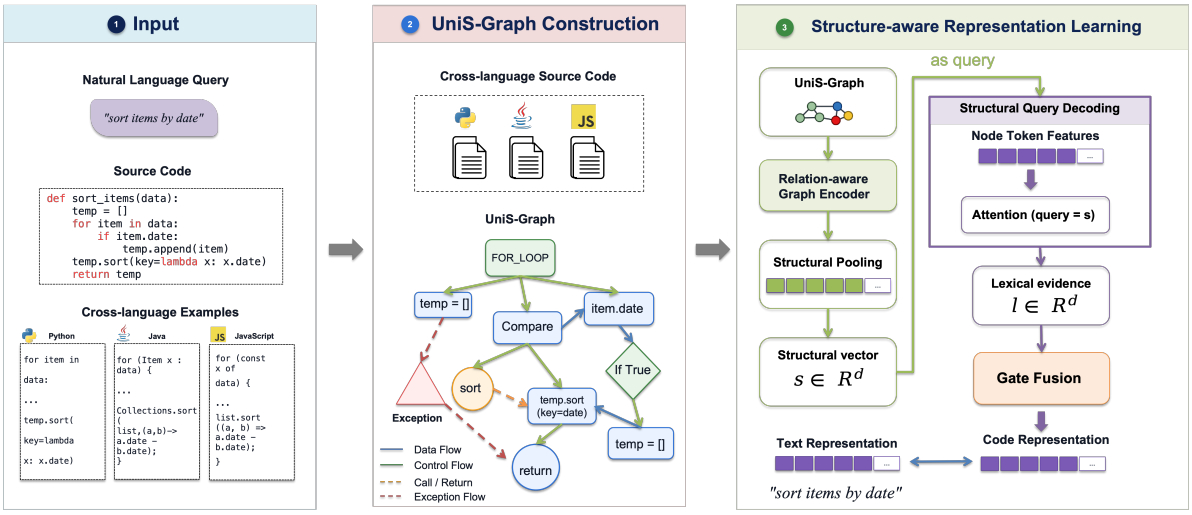


Figure 2: Overall framework of SASU. Given a natural-language query and source code, SASU constructs a normalized multi-relational UNIS-GRAPH, learns structure-aware graph representations, uses the graph representation as the query for structural query decoding, and aligns the final code representation with the text representation.

View	Relation	Meaning
Syntax	SYNTAX_PARENT_OF	AST parent-child containment
DFG	DATA_FLOW_TO	Definition/use and value propagation
	RETURNS_TO	Return value to function exit
CFG	CONTROL_NEXT	Sequential execution order
	CONTROL_TRUE	True branch or loop body
	CONTROL_FALSE	False branch
	CONTROL_LOOP_BACK	Loop back edge
	CALLS	Call site to callee symbol
	EXC_FLOW_TO	Exception-related transition

Table 2: Relation groups in UNIS-GRAPH. The graph composes normalized syntax with DFG-style value relations and CFG-style execution relations.

branches, loops, calls, and exception paths. These two views are composed with the normalized AST to form a multi-relational graph. Table 2 summarizes the relation groups used in UNIS-GRAPH.

Different relation groups correspond to different code behaviors. Data-flow relations are useful for value transformation, while control-flow relations are useful for conditional and iterative logic. We keep these relations separated so that the graph encoder can learn relation-specific propagation instead of treating the graph as a homogeneous structure.

Node features. Each node representation is initialized with lexical and type information. The initial representation of node v_i is computed as

$$\mathbf{h}_i^{(0)} = \mathbf{W}_{in}[\mathbf{e}_{tok}(i) \| \mathbf{e}_{type}(i)].$$

Here $\mathbf{e}_{tok}(i)$ is encoded by a pretrained CodeT5+ encoder (Wang et al., 2023), and $\mathbf{e}_{type}(i)$ is a trainable embedding of the normalized node type. For identifier and literal nodes, the input text is the corresponding surface string. For structural nodes without explicit surface tokens, the normalized type name is used as the textual input. In this way, the graph view contains both the concrete lexical evidence and the abstract structural role of each node. Appendix A provides the concrete feature construction procedure.

3.3 Structure-Aware Retrieval Encoding

In this section, we discuss how to encode the extracted graph for code–text retrieval. The main problems of the retrieval encoder lie in three aspects: 1) how to preserve the semantic and logical information contained in different relation types; 2) how to summarize a variable-size graph into a fixed-size code representation; and 3) how to combine structural information with lexical evidence. To address these problems, we design a structure-aware encoder that consists of relation-aware graph propagation, attention-based readout, and Structural Query Decoding.

Relation-aware graph encoding. Given the graph G, a message passing process is first conducted to fuse semantic and structural information into each node representation. For a node i and its neighbor j connected by relation r, the relation-specific attention score is computed as

$$e_{ij}^r = \mathrm{LeakyReLU}\left(\mathbf{a}_r^\top [\mathbf{W}_r \mathbf{h}_i^{(l)} \| \mathbf{W}_r \mathbf{h}_j^{(l)}]\right).$$

The attention weight is normalized over the relation-specific neighborhood:

$$\alpha_{ij}^{r} = \frac{\exp(e_{ij}^{r})}{\sum_{k \in \mathcal{N}_r(i)} \exp(e_{ik}^{r})}.$$

The message from relation r is then obtained by

$$\mathbf{m}_i^r = \sum_{j \in \mathcal{N}_r(i)} \alpha_{ij}^{r} \mathbf{W}_r \mathbf{h}_j^{(l)}.$$

Messages from different relations are composed with the self representation:

$$\mathbf{h}_i^{(l+1)} = \mathrm{FFN}\left(\mathbf{h}_i^{(l)} + \sum_{r \in R} \mathbf{m}_i^r\right).$$

Residual connection, LayerNorm, and dropout are applied after each propagation layer. Through this process, each node absorbs information from syntactic neighbors and executable dependencies while preserving the type of each relation.

Structural pooling. A global attention-based readout is applied to obtain the structural representation of the whole graph:

$$s_i = \mathbf{q}^{\top} \tanh(\mathbf{W}_p \mathbf{h}_i^{(L)}),$$
$$\beta_i = \frac{\exp(s_i)}{\sum_{j \in V} \exp(s_j)},$$
$$\mathbf{g} = \sum_{i \in V} \beta_i \mathbf{h}_i^{(L)}.$$

The pooled representation is projected into the retrieval space:

$$\mathbf{s} = \mathrm{MLP}_s(\mathbf{g}).$$

Here $\mathbf{s}$ denotes the structural representation of the code block. Compared with mean pooling, the attention readout can assign larger weights to nodes that carry more useful retrieval evidence, such as return nodes, branch predicates, call sites, and identifiers involved in data-flow paths.

Structural Query Decoding. After obtaining the structural representation, we use it to query lexical information from the node feature sequence. Let $\mathbf{T}$ denote the encoded node token features and $\mathbf{H}^{(L)}$ denote the final node states. We first construct structure-aware lexical features:

$$\mathbf{U} = \mathrm{Proj}_{tok}(\mathbf{T}) + \mathrm{Proj}_{str}(\mathbf{H}^{(L)}).$$

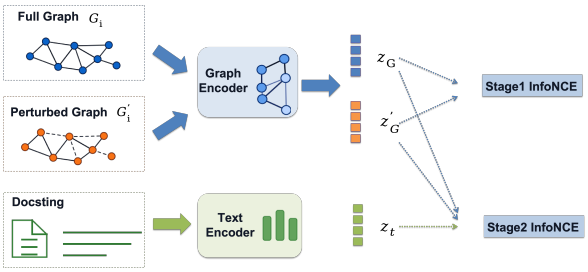


Figure 3: Two-stage training objective of SASU.

Then the structural vector $\mathbf{s}$ serves as the query, while $\mathbf{U}$ is used as both keys and values:

$$\mathbf{l} = \mathrm{Attn}(Q = \mathbf{s}, K = \mathbf{U}, V = \mathbf{U}).$$

The obtained vector $\mathbf{l}$ is the lexical evidence selected under the structural context. Finally, a gate is used to combine the structural representation and the decoded lexical representation:

$$[w_s, w_l] = \mathrm{softmax}(\mathrm{MLP}_g([\mathbf{s} \| \mathbf{l}])),$$
$$\mathbf{z}_c = w_s \mathbf{s} + w_l \mathbf{l}.$$

The representation $\mathbf{z}_c$ is used as the final code embedding. This design allows the model to use lexical cues when they are informative, while still grounding the lexical selection in the executable structure of the graph. The implementation details of this decoding step are summarized in Appendix A.

3.4 Training Objective

There are two types of alignment to be ensured during training: 1) structure-preserving alignment between a full graph and its perturbed view, and 2) code–text alignment between the code representation and the natural-language description. We optimize these objectives in two stages, as illustrated in Figure 3.

Structural consistency pretraining. For structure-preserving purpose, each graph G_i is paired with a corrupted view G_i' obtained by graph perturbation. The positive pairs are defined as $\mathcal{I}_{struct}^{+} = \{\langle G_i, G_i' \rangle\}$, and the negative pairs are other graph pairs within the minibatch. The structural consistency loss is formulated as

$$\mathcal{L}_{struct} = -\log \frac{\exp(\mathrm{sim}(\mathbf{z}_{G_i}, \mathbf{z}_{G_i'})/\tau)}{\sum_j \exp(\mathrm{sim}(\mathbf{z}_{G_i}, \mathbf{z}_{G_j'})/\tau)}.$$

Here τ is the temperature. This stage does not use natural-language descriptions and only encourages the graph encoder to produce stable representations under structural corruption.

Code–text contrastive alignment. In the second stage, the code embedding and the description embedding are aligned in a shared retrieval space. For a minibatch of paired code graphs and descriptions, the bidirectional contrastive loss is

$$\mathcal{L}_{C2T} = -\frac{1}{N}\sum_i \log \frac{\exp(\mathrm{sim}(\mathbf{z}_{c_i}, \mathbf{z}_{t_i})/\tau)}{\sum_j \exp(\mathrm{sim}(\mathbf{z}_{c_i}, \mathbf{z}_{t_j})/\tau)}.$$

$$\mathcal{L}_{T2C} = -\frac{1}{N}\sum_i \log \frac{\exp(\mathrm{sim}(\mathbf{z}_{t_i}, \mathbf{z}_{c_i})/\tau)}{\sum_j \exp(\mathrm{sim}(\mathbf{z}_{t_i}, \mathbf{z}_{c_j})/\tau)}.$$

The alignment loss is

$$\mathcal{L}_{align} = \frac{1}{2}(\mathcal{L}_{C2T} + \mathcal{L}_{T2C}).$$

This bidirectional form is used for the full–text, drop–text, and full–drop alignment terms in Stage 2. We additionally align the full-graph structural projection with the paired description:

$$\mathcal{L}_{struct\text{-}text} = \mathcal{L}_{align}(\mathbf{s}_i, \mathbf{z}_{t_i}),$$

The final objective is

$$\mathcal{L} = \lambda_1 \mathcal{L}_{full\text{-}text} + \lambda_2 \mathcal{L}_{drop\text{-}text} + \lambda_3 \mathcal{L}_{full\text{-}drop} + \lambda_4 \mathcal{L}_{struct\text{-}text},$$

where $\lambda_1, \lambda_2, \lambda_3, \lambda_4$ are loss weights. At inference time, only the learned code and text encoders are used, and retrieval ranks candidates by cosine similarity between $\mathbf{z}_c$ and $\mathbf{z}_t$. The training schedule is listed in Appendix A.

4 Experiments

In this section, we conduct experiments to answer the following research questions:

- **RQ1**: Does SASU improve code search over strong pretrained lexical encoders?
- **RQ2**: Does the benefit hold across programming languages?
- **RQ3**: Is SASU robust when code graphs are incomplete?
- **RQ4**: Which structural and lexical components contribute most to retrieval quality?
- **RQ5**: Does the advantage persist after direct identifier overlap is removed?
- **RQ6**: What code-side cost does structure-aware retrieval introduce?

Language	Train	Dev	Test
Python	342,533	18,747	18,534
Java	302,721	9,833	18,533
JavaScript	89,947	5,918	4,924
Total	735,201	34,498	41,991

Table 3: CodeSearchNet splits used in our experiments.

4.1 Setup

In this paper, we evaluate SASU on CodeSearchNet (Husain et al., 2019), a widely used code search benchmark. We use Python, Java, and JavaScript because they contain sufficient training examples and cover different syntax styles. Each example contains a function and its natural-language documentation. After preprocessing and graph construction, the statistics of the processed splits are shown in Table 3. The filtering rules and graph construction details are provided in Appendix B.

We compare SASU with five retrieval baselines: **CodeBERT** (Feng et al., 2020), **GraphCodeBERT** (Guo et al., 2021), **UniXcoder** (Guo et al., 2022), **Jina Embeddings v2 Code** (Günther et al., 2023), and **CodeT5+** (Wang et al., 2023). For Jina Embeddings v2 Code, we use the `jinaai/jina-embeddings-v2-base-code` checkpoint. For CodeT5+, we use the `Salesforce/codet5p-110m-embedding` checkpoint, which has about 110M parameters. Since SASU also uses CodeT5+-based node token features, improvements over CodeT5+ indicate that the gain does not only come from the lexical backbone. For all baselines, we encode the function body and its paired documentation into the same retrieval space and evaluate them under the same ranking protocol. SASU uses the graph representation as the code side, while the text side is encoded from the documentation.

We report Mean Reciprocal Rank (MRR) and Recall@K for $K \in \{1, 5, 10\}$. Each query is ranked against all candidates in the corresponding evaluation split. Before code–text alignment, we initialize the graph encoder with structural consistency pretraining. During code–text training, full graphs, perturbed graphs, and paired descriptions are jointly optimized with the contrastive losses described in Section 3. Implementation details, hyperparameters, and evaluation protocol are listed in Appendix B.

4.2 Overall Results (RQ1)

The main results are summarized in Table 4. SASU consistently outperforms the lexical and pretrained retrieval baselines in both retrieval directions. Compared with CodeT5+, SASU improves C→T MRR from 0.714 to 0.835 and T→C MRR from 0.727 to 0.840. The Recall@1 improvement is also clear, showing that the proposed structure-anchored representation improves the top-ranked retrieval results. The improvement over GraphCodeBERT and Jina Embeddings v2 Code further suggests that the gain is not limited to replacing one pretrained encoder with another. Instead, the normalized graph and the structure-guided fusion provide additional retrieval signals on top of pretrained lexical embeddings. The Stage-1 diagnostics in Appendix A further show that structural consistency pretraining improves graph-pair matching before code–text alignment, which supports its use as the initialization step for the retrieval model.

4.3 Language-Level Results (RQ2)

To answer RQ2, we further evaluate C→T retrieval for each programming language. The results are presented in Table 5. SASU improves over CodeT5+ on Python, Java, and JavaScript. The improvement shows that the graph representation is not only effective on one language but also useful across different syntax styles. Among the three languages, Python obtains the highest absolute score, while JavaScript remains the hardest setting. The performance gap across languages is consistent with the different graph construction difficulty. JavaScript contains more flexible syntax and more diverse coding patterns, while Python and Java functions are more regular in the filtered benchmark.

4.4 Robustness to Incomplete Graphs (RQ3)

To answer RQ3, we evaluate whether SASU can still retrieve relevant descriptions when the code graph is incomplete. We delete 40–60% of graph nodes at test time and use the damaged graph for retrieval. The construction of incomplete graphs and damaged-code snippets is described in Appendix C. The results are shown in Table 6. From the results, we can see that SASU still keeps strong retrieval performance under graph perturbation. Its C→T MRR decreases from 0.835 to 0.723, retaining 86.6% of its full-graph performance. The damaged-graph result of SASU is still higher than the complete-code CodeT5+ baseline. This indicates that the model does not rely on a single exact graph instance. The structural consistency pretraining and perturbed-graph alignment help the representation remain useful when part of the graph evidence is missing.

4.5 Ablation Study (RQ4)

To answer RQ4, we study the impact of different components of SASU. We first evaluate the structural and lexical pathways separately by forcing the fusion module to use only one pathway at a time. The results are presented in Table 7. Lexical evidence is a strong single branch, which is expected on CodeSearchNet because function descriptions often overlap with identifiers and comments. However, the full model still outperforms lexical-only retrieval, indicating that the structural branch provides complementary information. We use the terms "structural pathway" and "lexical-decoding pathway" because both pathways are built on the same graph input and node features; the ablation controls the fusion route rather than removing every possible lexical or structural signal from the model.

We then evaluate the impact of relation groups by removing different types of graph edges. The results are shown in Table 8. Data-flow relations contribute the most, followed by control-flow relations. This validates that code search benefits from behavior-level program relations, rather than only AST topology. The relation-group protocol is summarized in Appendix D.

Finally, we inspect the fusion gate behavior. Table 9 reports the average gate weights on clean and incomplete graph inputs. On clean test examples, the average lexical fusion weight is larger than the structural weight, which is consistent with the strong lexical-decoding pathway result. Under node deletion, the structural weight increases from 0.39 to 0.58. This indicates that the fusion module does not treat structure and lexical information as fixed features; it changes their relative influence according to the quality of the input graph.

4.6 Lexical-Overlap Control (RQ5)

CodeSearchNet descriptions can repeat function names and code identifiers, creating a lexical shortcut. We therefore remove terms derived from the function name and code identifiers from each test query, while retaining the remaining behavioral description. Table 10 shows that all methods degrade,

Method	Code → Text				Text → Code			
	MRR	R@1	R@5	R@10	MRR	R@1	R@5	R@10
UniXcoder	0.466	0.372	0.575	0.636	0.504	0.400	0.629	0.695
CodeBERT	0.684	0.594	0.793	0.831	0.695	0.607	0.806	0.841
GraphCodeBERT	0.702	0.611	0.816	0.854	0.715	0.625	0.829	0.864
Jina-v2-code	0.678	0.583	0.796	0.841	0.695	0.607	0.803	0.843
CodeT5+	0.714	0.620	0.830	0.869	0.727	0.634	0.843	0.879
SASU	**0.835**	**0.762**	**0.917**	**0.939**	**0.840**	**0.766**	**0.920**	**0.941**

Table 4: Overall retrieval performance on CodeSearchNet.

Lang.	Method	MRR	R@1	R@5	R@10
Python	UniXcoder	0.463	0.367	0.573	0.635
	CodeBERT	0.675	0.589	0.781	0.815
	GraphCodeBERT	0.694	0.607	0.804	0.839
	Jina-v2-code	0.716	0.612	0.844	0.892
	CodeT5+	0.753	0.658	0.872	0.910
	SASU	**0.883**	**0.812**	**0.953**	**0.969**
Java	UniXcoder	0.504	0.405	0.622	0.685
	CodeBERT	0.678	0.591	0.785	0.822
	GraphCodeBERT	0.693	0.604	0.803	0.840
	Jina-v2-code	0.708	0.625	0.811	0.845
	CodeT5+	0.701	0.611	0.812	0.850
	SASU	**0.801**	**0.723**	**0.891**	**0.920**
JavaScript	UniXcoder	0.463	0.360	0.584	0.652
	CodeBERT	0.623	0.523	0.751	0.795
	GraphCodeBERT	0.646	0.544	0.780	0.825
	Jina-v2-code	0.685	0.582	0.816	0.856
	CodeT5+	0.654	0.549	0.789	0.835
	SASU	**0.751**	**0.662**	**0.853**	**0.892**

Table 5: Code-to-text retrieval by programming language.

Method	C→T			T→C		
	MRR	R@1	R@10	MRR	R@1	R@10
UniXcoder	0.418	0.335	0.569	0.429	0.343	0.590
CodeT5+	0.602	0.513	0.757	0.577	0.488	0.738
SASU	**0.723**	**0.641**	**0.861**	**0.746**	**0.663**	**0.882**

Table 6: Retrieval performance when 40–60% of graph nodes are deleted at test time.

Variant	C→T MRR	T→C MRR	R@1	R@10
Structural pathway only	0.542	0.518	0.461	0.688
Lexical-decoding pathway only	0.791	0.805	0.712	0.915
Full SASU	**0.835**	**0.840**	**0.762**	**0.939**

Table 7: Ablation of fusion pathways. R@1 and R@10 are for C→T.

Relation configuration	C→T MRR	Δ
Full model	0.835	–
No exception-flow edges	0.832	-0.003
No return edges	0.828	-0.007
No call edges	0.816	-0.019
No control-flow edges	0.801	-0.034
No data-flow edges	0.763	-0.072
Syntax edges only	0.734	-0.101

Table 8: Ablation of UniS-Graph relation types.

Input setting	Structural weight	Lexical weight
Clean graph	0.39	0.61
Incomplete graph	0.58	0.42

Table 9: Average fusion gate weights on clean and incomplete graph inputs.

Query	Method	C→T			T→C		
		MRR	R@1	R@10	MRR	R@1	R@10
Original	CodeT5+	0.714	0.620	0.869	0.727	0.634	0.879
	SASU	**0.835**	**0.762**	**0.939**	**0.840**	**0.766**	**0.941**
Sanitized	UniXcoder	0.286	0.205	0.462	0.302	0.219	0.486
	CodeT5+	0.431	0.342	0.621	0.447	0.358	0.639
	SASU	**0.506**	**0.414**	**0.694**	**0.512**	**0.421**	**0.701**

Table 10: Retrieval after removing query terms derived from code identifiers.

confirming that lexical overlap is useful on this benchmark. Nevertheless, SASU remains ahead of CodeT5+ by 0.075 C→T MRR and 0.065 T→C MRR, indicating that its gain is not solely explained by direct identifier matching.

4.7 Efficiency (RQ6)

We profile code-side embedding on the CodeSearchNet test split using one NVIDIA RTX 4090 and report graph construction separately because corpus graphs and node features can be cached. As shown in Table 11, token-only encoders are faster, while cached SASU processes 116.3 functions/s. Including graph construction yields 10.6 ms/function; thus, SASU trades additional corpus-side computation for its structure-sensitive retrieval gains.

5 Conclusion

Despite the progress of pretrained code models, code search still faces challenges caused by the gap between natural-language intent and executable program behavior. This paper introduces SASU, a structure-anchored retrieval framework

Method	Build	Forward	Total	fn/s
UniXcoder	–	1.86	1.86	539.0
CodeT5+	–	2.18	2.18	458.5
SASU, cached	precomp.	8.60	8.60	116.3
SASU, online build	2.00	8.60	10.60	94.3

Table 11: Code-side latency in ms/function and throughput. "Cached" uses precomputed graphs and node features.

for code search. SASU extracts a normalized multi-relational graph from each function, summarizes syntax, data-flow, control-flow, call, return, and exception information, and uses the structural representation to guide lexical evidence selection before code–text alignment. By integrating executable structure into the retrieval representation, SASU reduces the dependence on surface lexical overlap while preserving the useful token evidence provided by pretrained encoders. Experiments on CodeSearchNet show that SASU improves both code-to-text and text-to-code retrieval over strong embedding baselines across Python, Java, and JavaScript. The controlled lexical-overlap, robustness, and ablation results further indicate that structural information provides complementary signals to lexical matching, including when direct identifier overlap is removed or graph inputs are incomplete. Overall, the findings suggest that program structure can serve as an effective anchor for building more reliable code search representations.

6 Limitations

This work has several limitations. First, CodeSearchNet contains strong lexical evidence in function names, identifiers, and documentation. Although identifier sanitization weakens direct lexical overlap, it does not remove every corpus-specific shortcut; comment removal and harder cross-implementation pairs are still needed to isolate structural reasoning more fully.

Second, UniS-Graph relies on static parsing and lightweight program analysis. Dynamic dispatch, reflection, metaprogramming, incomplete files, and parser errors may introduce noisy or missing relations, especially for dynamic languages and repository-dependent code.

Finally, we focus on function-level retrieval for Python, Java, and JavaScript. Repository-level search and broader language coverage would require interprocedural graph construction, scalable graph batching, and expanded normalization rules. Repository deployment would also require maintaining the precomputed graph and node-feature cache as code changes.

References

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *International Conference on Learning Representations*.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29.

Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. In *Advances in Neural Information Processing Systems*, volume 31.

Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 964–974. ACM.

Fabio James Fehr, Prabhu Teja S, Luca Franceschi, and Giovanni Zappella. 2025. CoRet: Improved retriever for code editing. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 775–789, Vienna, Austria. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Siddharth Gandhi, Luyu Gao, and Jamie Callan. 2025. Repository-level code search with neural retrieval methods. *arXiv preprint arXiv:2502.07067*.

Jian Gu, Zimin Chen, and Martin Monperrus. 2021. Multimodal representation for neural code search. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 483–494. IEEE.

Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdessalem, Tanguy Abel, Mohammad Kalim Akram, Susana Guzman, Georgios Mastrapas, Saba Sturua, Bo Wang, Maximilian Werk, Nan Wang, and Han Xiao. 2023. Jina Embeddings 2: 8192-token general-purpose text embeddings for long documents. *arXiv preprint arXiv:2310.19923*.

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)*, pages 7212–7225, Dublin, Ireland. 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 Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2011. Improving source code search with natural language phrasal representations of method signatures. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 524–527. IEEE.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Chaofan Li, Jianlyu Chen, Yingxia Shao, Defu Lian, and Zheng Liu. 2025a. Towards a generalist code embedding model based on massive data synthesis. In *Advances in Neural Information Processing Systems*, volume 38, pages 164871–164900. Curran Associates, Inc. Datasets and Benchmarks Track.

Jia Li, Xianjie Shi, Kechi Zhang, Ge Li, Zhi Jin, Lei Li, Huangzhao Zhang, Jia Li, Fang Liu, Yuwei Zhang, Zhengwei Tao, Yihong Dong, Yuqi Zhu, and Chongyang Tao. 2025b. GraphCodeAgent: Dual graph-guided LLM agent for retrieval-augmented repo-level code generation. *arXiv preprint arXiv:2504.10046*.

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*, pages 345–357. IEEE.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024. GraphCoder: Enhancing repository-level code completion via coarse-to-fine retrieval based on code context graph. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 570–581. ACM.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, pages 1287–1293.

Huy N. Phan, Hoang N. Phan, Tien N. Nguyen, and Nghi D. Q. Bui. 2025. RepoHyper: Search-expand-refine on semantic graphs for repository-level code completion. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pages 14–25. IEEE.

Kesu Wang, Meng Yan, He Zhang, and Haibo Hu. 2022. Unified abstract syntax tree representation learning for cross-language program classification. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 390–400. ACM.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088, Singapore. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Yufan Ye, Pu Pang, Ting Zhang, and Hua Huang. 2025. GNN-Coder: Boosting semantic code retrieval with combined GNNs and transformer. *arXiv preprint arXiv:2502.15202*.

A Framework Details

Graph extraction pipeline. For each function-level code block c, SASU first parses the source program, normalizes parser-specific node labels, and then augments the syntax tree with behavior-oriented relations. The extraction can be written as

$$T \leftarrow \mathrm{Parse}(c),$$
$$V \leftarrow \mathrm{NormalizeNodes}(T),$$
$$E \leftarrow E_{\mathrm{AST}} \cup E_{\mathrm{DFG}} \cup E_{\mathrm{CFG}}.$$

Here E_{AST} contains normalized parent-child syntax edges. E_{DFG} contains value-oriented edges, including definition/use propagation and return-value links. E_{CFG} contains execution-oriented edges, including sequential transitions, branch transitions, loop-back transitions, call links, and exception-flow transitions. The final graph is

$$G = (V, E, R, X),$$

where R is the relation vocabulary and X stores the node-level textual and type features.

Node text construction. Each graph node is assigned a textual input before feature encoding. If a node corresponds to an identifier, literal, operator, attribute, or source token, we use its surface text. If a node is a structural node without an explicit surface token, we use its normalized type name. Formally,

$$x_i = \begin{cases} \mathrm{Surface}(v_i), & \text{token node,} \\ \mathrm{TypeName}(v_i), & \text{otherwise.} \end{cases}$$

The token feature is encoded as $\mathbf{e}_{tok}(i) = \mathrm{CodeT5+}(x_i)$. The type feature $\mathbf{e}_{type}(i)$ is a learned embedding indexed by the normalized node type. These two features are concatenated and projected into the graph encoder input space.

Relation-aware propagation. During graph encoding, edges are grouped by their relation labels. For each relation r, the encoder applies a relation-specific linear projection and attention parameter. If a graph does not contain edges of relation r, that relation contributes a zero message for the current batch. After relation-specific messages are summed, the model applies a feed-forward block, residual connection, LayerNorm, and dropout. This keeps the encoder compatible with graphs that contain different subsets of syntax, data-flow, and control-flow relations.

Structural Query Decoding. The SQD module receives the structural projection $\mathbf{s}$, token-level features $\mathbf{T}$, final node states $\mathbf{H}^{(L)}$, and the graph batch index. Node features are first converted into dense per-graph sequences with a padding mask. The lexical memory is

$$\mathbf{U} = \mathrm{Proj}_{tok}(\mathbf{T}) + \mathrm{Proj}_{str}(\mathbf{H}^{(L)}).$$

The structural projection is then expanded as a single-query sequence:

$$\mathbf{Q} = \mathbf{s}[\,\cdot\,].$$

Multi-head attention is applied with $\mathbf{Q}$ as query and $\mathbf{U}$ as both keys and values. The padding mask prevents invalid padded nodes from contributing to attention. The attention output is normalized and projected into the lexical retrieval space before being combined with the structural projection through a two-way softmax gate.

Training schedule. Training follows two stages. Stage 1 optimizes structural consistency between the full graph and the perturbed graph. Stage 2 initializes from the Stage-1 graph encoder and optimizes code–text alignment.

Metric	Untrained	Stage 1	Δ
Loss	2.4872	0.0291	−2.4581
MRR	0.8359	0.9413	+0.1054
MRR (reverse)	0.6772	0.9427	+0.2655
R@1	0.7828	0.9132	+0.1304
R@1 (reverse)	0.6267	0.9178	+0.2911
R@10	0.9292	0.9838	+0.0546
R@10 (reverse)	0.7723	0.9838	+0.2115

Table 12: Stage-1 structural consistency diagnostics on the held-out graph-pair evaluation set. Stage 1 denotes the model after five epochs of structural consistency pretraining.

Stage-1 diagnostics. Table 12 reports the held-out structural matching results before and after Stage-1 pretraining. The evaluation ranks the paired perturbed graph for each full graph, and also reports the reverse direction.

During Stage 2, the loss weights are scheduled as follows:

Epochs	$\lambda_{full\text{-}text}$	$\lambda_{drop\text{-}text}$	$\lambda_{full\text{-}drop}$	$\lambda_{struct\text{-}text}$
1–3	0.50	0.30	0.20	0.40
4–8	0.70	0.20	0.10	0.40
9–16	0.85	0.10	0.05	0.40

The last term is used only during training and is not included in evaluation loss reporting.

B Experimental Details

Dataset preprocessing. We use the Python, Java, and JavaScript subsets of CodeSearchNet. We keep examples whose documentation has 3–256 CodeT5+ tokens, is mostly English, and contains no HTML-style special tokens or URLs. Code longer than 512 CodeT5+ tokens is removed, and leading Python docstrings are stripped from the code side. After parsing, examples without a valid function-level graph are discarded. Node and edge vocabularies are built from the training and validation splits, and each kept example is exported with a full graph, a perturbed subgraph, and its paired documentation.

Ranking protocol. For code-to-text retrieval, the code representation is used as the query and all documentation representations in the evaluation split are used as candidates. For text-to-code retrieval, the direction is reversed. MRR is computed as

$$\mathrm{MRR} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{\mathrm{rank}_i},$$

where rank_i is the rank of the paired item for the i-th query. Recall@K measures whether the paired

item appears in the top K results.

Implementation. SASU is implemented with PyTorch and PyTorch Geometric. The graph encoder contains two relation-aware attention layers with hidden size 352, four attention heads, and dropout 0.1. The projection dimension is 128. Node token features are produced by a pretrained CodeT5+ encoder. Stage 2 uses AdamW with learning rate 2.5×10^{-4}, batch size 512, cosine scheduling, and 16 epochs. The contrastive temperature is 0.07.

C Incomplete Inputs

Graph perturbation. Let a full graph be $G = (V, E, R)$, where V is the node set, E is the edge set, and R is the relation-type set. We construct an incomplete graph $G' = (V', E', R)$ by selecting a connected subset of nodes and removing all incident edges outside the subset:

$$V' \subset V, \qquad \rho_{\min}|V| \le |V'| \le \rho_{\max}|V|,$$
$$E' = \{(u, r, v) \in E \mid u \in V', v \in V'\}.$$

In our experiments, $\rho_{\min} = 0.4$ and $\rho_{\max} = 0.6$. The retained set V' is sampled by breadth-first expansion on the undirected version of G. The seed node is sampled from behavior-bearing nodes when possible, including branch statements, loops, call expressions, blocks, expression statements, and return statements. If no such node exists, the sampler falls back to syntax nodes. Function entry, function exit, and the root node are always inserted into V' when they are available. We denote this anchor set as

$$V_{\text{anchor}} = \{v_{\text{entry}}, v_{\text{exit}}, v_{\text{root}}\},$$
$$V' \leftarrow V' \cup V_{\text{anchor}}.$$

If the sampled component is still smaller than the target size, remaining nodes are added in random order until the size constraint is satisfied. This procedure damages the graph without turning it into an arbitrary set of isolated nodes.

Damaged-code reconstruction. For text-side robustness checks, we also reconstruct a partial code snippet that is consistent with the retained graph. Each graph node v may carry a source span $\sigma(v) = (a_v, b_v)$. The retained spans are collected from V':

$$S' = \{\sigma(v) \mid v \in V',\ \sigma(v) \text{ is defined}\}.$$

Overlapping or adjacent spans are merged after sorting by the start offset:

$$\text{Merge}(S') = \{(\alpha_1, \beta_1), \ldots, (\alpha_m, \beta_m)\},$$
$$\beta_j < \alpha_{j+1}.$$

Given the original source string c, the damaged-code view is reconstructed as

$$c' = \text{concat}\big(c[\alpha_1 : \beta_1], \ldots, c[\alpha_m : \beta_m]\big).$$

Thus the text view and the graph view remove evidence from the same retained-node decision. This prevents the damaged-code setting from introducing a different corruption policy from the incomplete-graph setting.

D Ablation Details

Fusion-pathway ablation. The structural-pathway variant forces the fusion gate to use the structural projection. The lexical-decoding-pathway variant forces the gate to use the lexical projection decoded by structural query attention. Both variants keep the same input graph and node features, so the ablation measures the contribution of the fusion route rather than removing all structural or lexical information from the system.

Relation-group ablation. For relation ablation, we evaluate the trained model after masking selected relation groups from the input graph. The syntax-only setting keeps only AST-style parent relations. The no-data-flow, no-control-flow, no-call, no-return, and no-exception settings remove the corresponding relation groups while leaving the remaining graph unchanged.