Research Article

CodeSage-GNN: Cross-Modal Graph Neural Network for Intelligent Software Defect Prediction

Armaghan Mubeen Butt[1], Muhammad Zayaan Waqar[2*], Muhammad Haseeb Zia

[1, 2] Department of Computer Science, The University of Alabama at Birmingham
[2] Department of Criminology and Forensic Sciences, Lahore Garrison University, Lahore, Pakistan
[*]Corresponding Author: Muhammad Haseeb Zia Email: haseebzia896@gmail.com

Received: June 14, 2025 **Accepted:** November 20, 2025 **Published:** November 23, 2025

Abstract: This study proposes CodeSage-GNN, a cross-modal graph neural framework for predicting defect-prone software modules. CodeSage-GNN represents each file or class as a heterogeneous graph combining structural metrics, dependency relations, and semantic embeddings of source code tokens. A dual-channel message-passing network jointly learns from structural graphs and textual semantics, while an attention-based fusion layer highlights the most influential features contributing to defects. The model is evaluated on multiple open software defect benchmarks to measure F1-score, MCC, AUC, and cross-project generalization. By integrating graph representation learning with interpretable attentions, CodeSage-GNN aims to support early fault localization and quality assurance in large-scale software systems.

***Keywords*:** Software defect prediction, Graph neural networks (GNN), Cross-modal learning, Static code analysis, Semantic code embeddings, Attention-based fusion, Explainable AI, Software quality assurance

1. Introduction

Software defect prediction (SDP) plays a vital role in improving software reliability, reducing maintenance costs, and guiding quality assurance strategies in large-scale software systems. As modern software projects grow in complexity, manual code inspection and traditional testing methods become insufficient for identifying defect-prone modules early in the development cycle. Consequently, automated defect prediction models have become an essential component of contemporary software engineering practice. Early benchmark studies demonstrated that machine learning classifiers could significantly enhance defect prediction accuracy by leveraging code metrics and historical defect data [1]. Subsequent systematic reviews reinforced these findings, emphasizing the need for robust and reproducible prediction frameworks capable of performing reliably across diverse projects and datasets [2].

Research into software defect prediction has historically focused on handcrafted metrics—such as lines of code, cyclomatic complexity, coupling, and cohesion—which have proven useful but often fail to capture the deeper structural and semantic nuances present in modern codebases. Comprehensive reviews of defect prediction metrics highlighted limitations related to metric redundancy, limited expressiveness, and insufficient representation of code dependencies [3]. These challenges prompted a shift from traditional feature engineering to more sophisticated learning paradigms that can extract meaningful patterns from raw software artifacts.

Recent advances in machine learning and deep learning have further expanded the scope of defect prediction. Emerging reviews and empirical studies illustrate a growing interest in leveraging semantic representations, neural architectures, and multi-modal learning to capture richer contextual information from software repositories [4]. Despite this progress, many existing models still rely heavily on surface-level numerical features or isolated textual embeddings, overlooking the interconnected and graph-structured nature of software components.

To address these gaps, contemporary research advocates for integrating structural, relational, and semantic information into unified predictive models [5]. Such approaches aim to reflect how developers inherently reason about

code—through its dependencies, logical flows, and meaning—rather than through isolated metrics. Graph neural networks (GNNs) in particular offer a promising direction, providing a natural mechanism for encoding code structure and dependency relations while supporting deep semantic learning.

Building upon these insights, this study introduces CodeSage-GNN, a cross-modal graph neural network designed to unify structural metrics, dependency relations, and semantic token embeddings for intelligent and interpretable defect prediction. The proposed model aims to bridge the gap between traditional metrics-based approaches and modern deep semantic modeling, offering a more holistic view of software artifacts and addressing long-standing limitations in cross-project generalization and defect localization.

The remainder of this paper is organized as follows. Section 2 reviews related work on traditional machine learning, deep learning, and graph-based approaches for software defect prediction. Section 3 presents the overall architecture of CodeSage-GNN, describing the code representation, heterogeneous graph construction, dual-channel message passing, and attention-based fusion mechanism in detail. Section 4 outlines the experimental setup, including datasets, preprocessing steps, baseline models, evaluation metrics, and implementation details. Section 5 reports and discusses the empirical results, with particular emphasis on cross-project generalization and interpretability analysis. Finally, Section 6 concludes the paper and highlights future research directions

2. Related Review

2.1 Surveys and Mapping Studies on Software Defect Prediction

Beyond the foundational benchmarking and metric-focused studies, a large body of work has synthesized the evolution of software defect prediction (SDP) through systematic mapping and literature reviews. Özakıncı and Tarhan provided an early systematic map of *early* defect prediction approaches, finding that most models still depend heavily on traditional product and process metrics collected late in the lifecycle, limiting their usefulness for proactive quality management [6]. Son et al. conducted a comprehensive mapping study and showed that although machine learning is widely adopted, deep learning and representation-learning methods remain underexplored compared to classical algorithms such as random forests and support vector machines [7].

More recent surveys highlight a rapid shift toward deep learning and hybrid techniques. Zain et al. performed a systematic review and meta-analysis of deep learning models for SDP, confirming that neural architectures often outperform traditional baselines but also noting issues of data hunger, overfitting, and limited cross-project evaluation [8]. Wadood et al. [32] research on deep learning applications for wind energy forecasting in smart grids demonstrates how advanced AI models are increasingly used to enhance prediction accuracy in complex systems. Omri et al. presented a survey dedicated to deep learning for SDP, emphasizing the potential of sequence models and autoencoders but pointing out that most studies still treat software modules as flat feature vectors or token sequences without explicitly modeling code structure [10]. Hassan et al. [31] empirically evaluated DCCP in multi-node real-time networks and showed that it can deliver lower latency and better QoS trade-offs compared to TCP. Batool and Khan offered a broad systematic review that spans data mining, machine learning, and deep learning techniques, reiterating persistent gaps related to imbalanced data, lack of interpretability, and poor generalization across projects and datasets [25].

These surveys collectively suggest that while predictive performance has improved, there is still a need for models that (i) capture multi-modal information (structural + semantic), (ii) generalize across projects, and (iii) provide human-understandable explanations.

2.2 Deep Learning Models for Software Defect Prediction

A substantial stream of work explores deep architectures operating mainly on metric vectors or textual representations of code. Liang et al. proposed Seml, a semantic LSTM model that learns defect-related patterns from sequences of code tokens and comments, showing gains over traditional feature-based approaches [15]. Deng et al. further demonstrated the feasibility of LSTM-based architectures for SDP, although their models rely primarily on sequential information and do not explicitly encode inter-module relationships [16]. Khleel and Nehéz extended this line by combining bidirectional LSTM networks with oversampling techniques to address class imbalance, improving performance on minority defect classes [17].

Beyond sequence models, Zhang et al. introduced a hierarchical feature ensemble deep learning approach that integrates multiple levels of features (e.g., product, process, change metrics), yielding improved performance but still operating mainly on tabular representations [18]. Albattah et al. empirically compared a range of machine learning and deep learning models, confirming that deep architectures often outperform classical models but also observing sensitivity to hyperparameter choices and dataset characteristics [28]. Alshammari proposed an enhanced random forest (extRF) technique and demonstrated that carefully tuned ensemble methods can remain competitive with, and sometimes surpass, more complex deep models [24].

Malhotra et al. reviewed hybrid SDP techniques and concluded that combinations of conventional metrics with advanced learning (e.g., deep or ensemble methods) are promising, yet they often treat metrics as independent features and overlook the explicit structural relationships among modules [23]. Ali et al. further showed that metaheuristic-based hyperparameter optimization can substantially boost model performance, but their framework still operates on flat feature spaces rather than structured code representations [29].

Overall, these studies highlight the power of deep and hybrid models but largely retain a *vector-based* view of software, limiting their ability to exploit the graph-structured nature of code and inter-module dependencies.

2.3 Graph-Based, Semantic, and Multi-View Approaches

More recent work begins to exploit graph-based and semantic representations of code. Zhou et al. proposed a graph convolutional network (GCN) model that integrates both semantic and structural information of source code for defect prediction, demonstrating that explicitly modeling code structure can improve accuracy over traditional baselines [12]. Cui et al. developed a graph neural network model from a complex network perspective, representing modules and their relationships as graphs and showing that graph representations capture useful connectivity patterns for defect prediction [13]. Šikić et al. further explored graph neural networks for source code defect prediction, confirming that GNNs can effectively leverage dependency information such as call graphs and control-flow relationships [14].

Beyond pure GNNs, Kıyak et al. explored multi-view learning for SDP, combining different metric views to improve predictive performance, but their views are still largely tabular and do not fully leverage code-token semantics [20]. Chen et al. proposed semantic-view-based SDP where multiple semantic aspects of source code are considered, yet their model does not explicitly encode the underlying module dependency graph [21]. Liu et al. introduced SeDPGK, a semi-supervised SDP framework that combines graph representation learning with knowledge distillation, demonstrating that graph-based semi-supervised learning can exploit unlabeled data and improve performance under label scarcity [26].

Together, these studies indicate that graph-based and multi-view approaches are promising directions, but existing models often treat structure and semantics either separately or in a loosely coupled manner. A unified cross-modal architecture that tightly fuses structural graphs with rich semantic embeddings and offers interpretable attention is still relatively unexplored.

2.4 Cross-Project Prediction, Explainability, and Business Perspective

Another important research line concerns cross-project prediction, explainability, and practical value. Bai et al. proposed a three-stage transfer learning framework for multi-source cross-project SDP, addressing domain shift between source and target projects and improving cross-project performance [19]. Gottumukkala et al. used BERTopic and multi-output classifiers to predict not only defect presence but also severity levels, moving closer to practical prioritization of defects [27]. Stradowski et al. discussed machine learning for SDP from a business-value perspective, arguing that predictive performance alone is insufficient and that cost-sensitive and risk-aware evaluation is crucial for industrial adoption [30].

Explainability has also gained attention. Gezici Geçer and Tarhan proposed an explainable AI framework for SDP that combines prediction models with interpretability techniques to provide actionable insights for practitioners [22]. Their work underscores the need for models that can show *why* a module is predicted as defective, not just *that*

it is defective. Zain et al.'s deep learning meta-analysis and Li et al.'s discussion of future directions similarly emphasize that interpretability and cross-project robustness remain open challenges, even in recent deep-learning-based SDP research [8], [11].Son et al., Alshammari, and Batool & Khan collectively point out that current SDP research tends to fragment into separate lines—metric-focused, deep learning, graph-based, and explainable AI—without a holistic framework that addresses structure, semantics, generalization, and interpretability together [7], [24], [25]. This fragmentation motivates the design of new models such as **CodeSage-GNN**, which aims to integrate cross-modal graph representation learning with attention-based interpretability and systematic evaluation on multiple benchmark datasets, including cross-project scenarios.

2.5 Summary of Research Gaps

Table 1 summarizes key categories of prior work, their main strengths, and the remaining gaps that CodeSage-GNN is designed to address.

Table 1. Summary of Related Work and Research Gaps

Representative Works	Main Focus	Key Gaps in Their Work	How CodeSage-GNN Addresses These Gaps
Surveys & Mapping Studies [6], [7], [8], [10], [25]	Map the landscape of SDP methods (ML, DL, hybrid) and identify trends, metrics, and datasets.	Highlight issues such as metric redundancy, imbalance, lack of cross-project studies, and limited use of structural/semantic representations, but do not propose concrete unified models.	CodeSage-GNN directly operationalizes the surveyed recommendations by designing a unified cross-modal GNN that combines structural metrics, dependency graphs, and semantic embeddings, and explicitly evaluates cross-project generalization.
Deep Learning on Vectors/Sequences [15], [16], [17], [18], [23], [24], [28], [29]	Use LSTM/Bi-LSTM, hierarchical deep models, ensembles, and hyperparameter optimization over metric vectors or token sequences.	Treat software modules as flat feature vectors or pure sequences, largely ignoring explicit graph structure and fine-grained dependencies between modules; interpretability is limited and structure–semantics fusion is weak.	CodeSage-GNN models each module as a heterogeneous graph, performs dual-channel message passing over structural and semantic views, and uses attention-based fusion to highlight influential nodes/features, thus exploiting structure and providing more interpretable predictions.
Graph-Based & Multi-View Models [12], [13], [14], [20], [21], [26]	Introduce GCN/GNN-based models, complex-network perspectives, multi-view and semi-supervised graph representation learning.	Typically focus either on structure (graphs) or semantics (views) but not a fully integrated cross-modal architecture; attention-based interpretability and detailed cross-project evaluation are often limited.	CodeSage-GNN defines a cross-modal heterogeneous graph where structural metrics, dependency relations, and semantic embeddings are learned jointly via dual-channel GNNs with an attention fusion layer; the design emphasizes both predictive performance and interpretability.
Transfer Learning, Severity & Business Value [19], [27], [30]	Address cross-project transfer, defect severity prediction, and business/value perspectives of SDP.	Consider transfer or severity in isolation, often using classical or vector-based models; they do not exploit cross-modal graph representations	CodeSage-GNN is evaluated in cross-project settings and can be extended to severity-aware prediction by reusing its cross-modal representations; attention weights over graph nodes and features

		or provide fine-grained explanations at the code-structure level.	can support severity reasoning and cost-sensitive prioritization.
Explainable AI & Hybrid Techniques [11], [22], [23], [24], [25]	Discuss future directions, XAI frameworks, and hybrid ML/DL schemes for SDP.	XAI work often wraps post-hoc explanation around black-box models; hybrid schemes still operate on tabular metrics without structural semantics; future directions call for integrated, interpretable architectures.	CodeSage-GNN embeds interpretability into the architecture itself via attention-based fusion over graph-structured and semantic features, enabling intrinsic explanations (e.g., which dependencies or code tokens drive the prediction) rather than purely post-hoc explanations.

3. Methodology

This section presents the proposed CodeSage-GNN framework for intelligent software defect prediction. The core idea is to represent each software module as a heterogeneous graph that combines structural metrics, dependency relations, and semantic information extracted from source code tokens. A dual-channel graph neural network processes structural and semantic views in parallel, and an attention-based fusion layer aggregates them into a unified representation used for defect prediction.

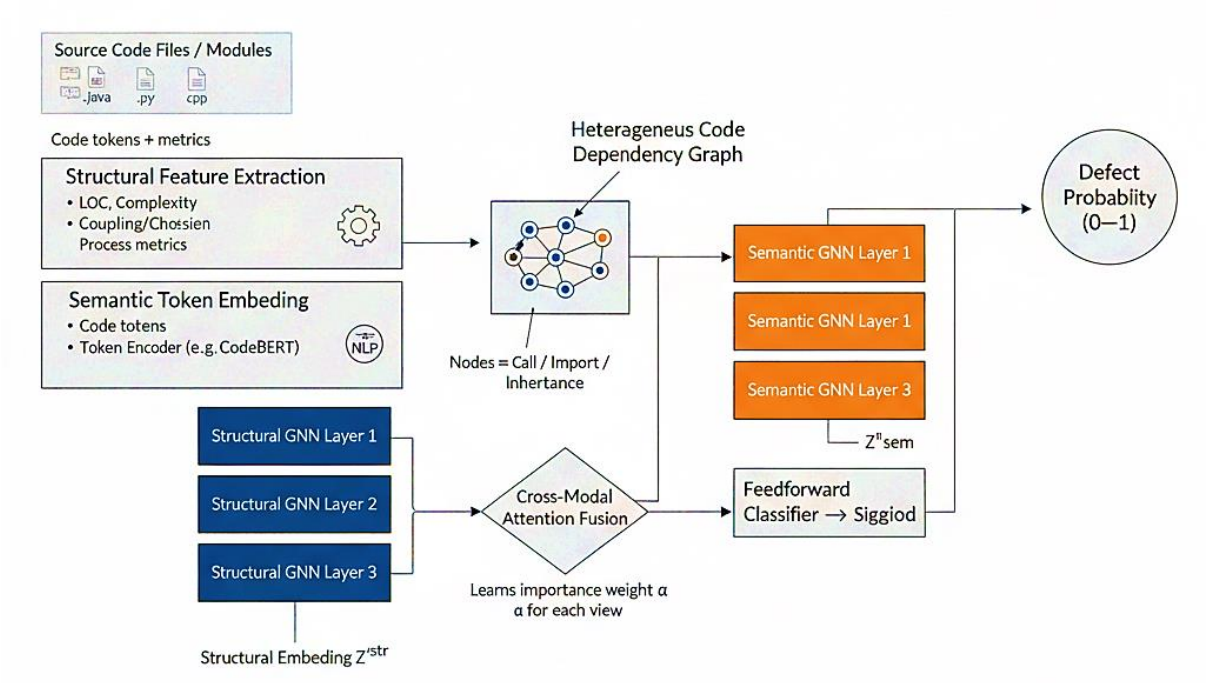

Figure 1. Overall architecture of CodeSage-GNN

Figure 1 shows overall architecture of CodeSage-GNN, showing the full workflow from raw code files to structural and semantic feature extraction, heterogeneous graph construction, dual-channel message passing, attention-based fusion, and final defect prediction.

3.1 Overall Architecture

At a high level, CodeSage-GNN consists of the following stages:

1. Code Representation and Feature Extraction

- Parse source files/classes into modules.
- Extract structural metrics (e.g., LOC, complexity, coupling).
- Build dependency relations (e.g., call, import, inheritance).
- Extract semantic token sequences and encode them using a pre-trained code-aware encoder.

2. Heterogeneous Graph Construction
   - Represent each project as a graph where nodes correspond to modules and edges capture dependency relations.
   - Associate each node with both structural features and semantic embeddings.
3. Dual-Channel Message Passing
   - Structural Channel: a GNN that focuses on metric-driven structural representations.
   - Semantic Channel: a GNN that refines semantic embeddings while respecting the graph structure.
4. Attention-Based Cross-Modal Fusion
   - Fuse structural and semantic node embeddings using an attention mechanism that learns which view is more informative for each module.
5. Defect Classification
   - The fused embedding passes through a feed-forward classification head to produce a defect probability for each module.

3.2 Code Representation and Feature Extraction

3.2.1 Module Definition

In this work, a **module** is defined as a source file or class, depending on the granularity of the dataset. For PROMISE/NASA datasets (e.g., KC1, JM1, PC1), modules typically correspond to files or classes with associated **metric vectors** and binary defect labels. For CodeXGLUE defect detection, modules correspond to individual source files or functions with defect annotations.

3.2.2 Structural Metrics

For each module v_i, we collect a vector of structural metrics:

- Size metrics: lines of code (LOC), number of statements, number of methods.
- Complexity metrics: cyclomatic complexity, maximum nesting depth.
- Coupling and cohesion metrics: coupling between objects, fan-in, fan-out.
- Process metrics (if available): number of revisions, number of authors, past defects.

Let $x_i^{(str)} \in R^{d_{str}}$ denote the structural feature vector for module v_i

Table 1 can list all structural metrics used, grouped by category (size, complexity, coupling, process), along with a short description and their source (PROMISE/NASA, derived from code, etc.).

Table 2. Structural code metrics used in CodeSage-GNN with their functional descriptions.

Category	Metric	Description
Size	LOC	Lines of code in module
Size	NOS	Number of statements

Complexity	CC	Cyclomatic complexity
Complexity	MaxNesting	Maximum nesting depth
Coupling	CBO	Coupling Between Objects
Cohe-sion	LCOM	Lack of Cohesion in Methods
Process	Revi-sions	Number of commits
Process	Authors	Number of distinct contributors

3.2.3 Semantic Token Embeddings

To capture the semantic aspects of the code (identifiers, keywords, API calls, comments), we tokenize each module and map tokens to embeddings using a pre-trained model (e.g., CodeBERT, token-level Word2Vec/FastText, or any code-specific encoder). The sequence of token embeddings is then aggregated into a fixed-size vector using one of the following strategies:

- Mean or max pooling over token embeddings.
- A lightweight BiLSTM/Transformer encoder for contextualized representation.

The resulting semantic representation of module v_i is denoted as

$$x_i^{(sem)} \in \mathbb{R}^{d_{sem}}$$

To ensure compatibility with the GNN channels, both views are projected to a common latent space:

$$\tilde{x}_i^{(str)} = W_{str} x_i^{(str)}, \tilde{x}_i^{(sem)} = W_{sem} x_i^{(sem)}$$

where $W_{str} \in \mathbb{R}^{d \times d_{str}}$ and $W_{sem} \in \mathbb{R}^{d \times d_{sem}}$ are trainable projection matrices.

3.3 Heterogeneous Graph Construction

3.3.1 Nodes and Edges

We model each project as a directed graph

$$G = (V, E)$$

where:

- $V = \{v_1, v_2, \ldots, v_N\}$ is the set of modules.
- $E \subseteq V \times V$ is the set of directed edges representing **dependency relations**, such as:
- function/method calls,
- inheritance / implementation relations,
- import / package dependencies.

Each node viv_ivi is associated with both $\tilde{x}_i^{(str)} and\ \tilde{x}_i^{(sem)}$ as initial features for the two channels.

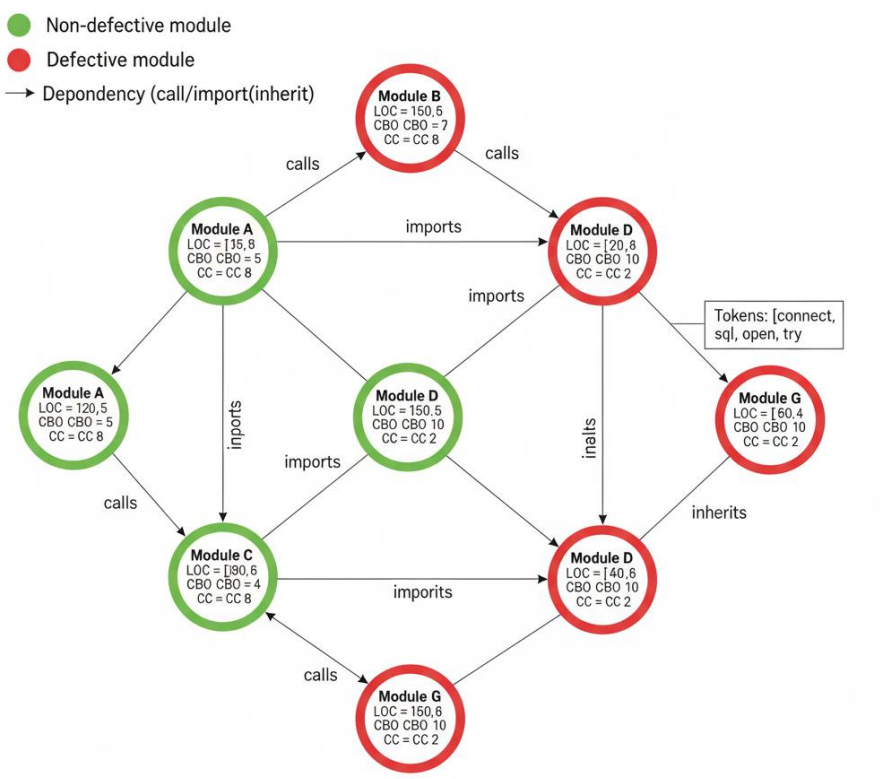

Figure 2. heterogeneous software project graph

Figure 2 should illustrate a sample project graph: nodes as modules, edges as call/import relations, with color/shape indicating defect-prone vs clean modules.

3.3.2 Adjacency and Normalization

We construct an adjacency matrix $A \in \mathbb{R}^{N\times N}$ based on dependencies:

$$A_{ij} = \begin{cases} 1 & if\ there\ is\ a\ dependency\ from\ v_i\ to\ v_j \\ 0 & otherwise \end{cases}$$

To stabilize training, a normalized adjacency matrix $\hat{A}$ is used, e.g.:

$$\hat{A} = D^{-\frac{1}{2}}(A + I)D^{-\frac{1}{2}}$$

where D is the degree matrix and I is the identity matrix.

3.4 Dual-Channel Graph Neural Network

The core of CodeSage-GNN is a **dual-channel GNN** that processes structural and semantic representations in parallel while sharing the same graph topology $\hat{A}$.

3.4.1 Structural Channel

The structural channel focuses on metric-driven information. It takes $\tilde{\boldsymbol{X}}^{(str)} = [\tilde{\boldsymbol{x}}_1^{(str)} \ldots \tilde{\boldsymbol{x}}_N^{(str)}]$as input and applies LLL layers of graph convolutions (or any GNN variant):

$$\boldsymbol{H}^{(str,0)} = \tilde{\boldsymbol{X}}^{(str)}$$

$$\boldsymbol{H}^{(str,l+1)} = \sigma(\hat{A}\boldsymbol{H}^{(str,l)}\boldsymbol{W}^{(str,l)})$$

for $\ell = 0, \ldots, L-1$, where:

- $\boldsymbol{W}^{(str,l)}$ are trainable weight matrices,
- $\boldsymbol{\sigma}(\cdot)$ is a non-linear activation (e.g., ReLU).

After the final layer, we obtain node embeddings:

$$\boldsymbol{Z}^{(str)} = \boldsymbol{H}^{(str,L)}$$

3.4.2 Semantic Channel

The semantic channel uses the same graph structure but operates on semantic features:

$$\boldsymbol{H}^{(sem,0)} = \widetilde{\boldsymbol{X}}^{(sem)}$$

$$\boldsymbol{H}^{(sem,l+1)} = \sigma(\hat{A}\boldsymbol{H}^{(sem,l)}\boldsymbol{W}^{(sem,l)})$$

leading to final semantic embeddings:

$$\boldsymbol{Z}^{(sem)} = \boldsymbol{H}^{(sem,L)}$$

The two channels may share the same depth L but use different parameters$\boldsymbol{W}^{(str,l)}$ and$\boldsymbol{W}^{(sem,l)}$, allowing them to specialize for structural vs semantic information.

Table 2 can summarize main hyperparameters of the dual-channel GNN, such as number of layers L, hidden size d, activation functions, dropout rate, and choice of GNN variant (GCN, GAT, GraphSAGE).

Table 3. Hyperparameters used in the structural and semantic channels of CodeSage-GNN.

Parameter	**Structural Channel**	**Semantic Channel**
Layers	2–3 GCN layers	2–3 GCN/GraphSAGE layers
Hidden Size	64 / 128	64 / 128
Activation	ReLU	ReLU
Dropout	0.3	0.3
Initial Input Dim	d_str	d_sem
Output Dim	d	D

3.5 Attention-Based Cross-Modal Fusion

For each node v_i, we have structural embedding $\boldsymbol{z_i^{(str)}}$and semantic embedding$\boldsymbol{z_i^{(sem)}}$ from the two channels. CodeSage-GNN uses an attention-based fusion layer to compute a single cross-modal representation z_i

First, we map each view to an intermediate space:

$$u_i^{(str)} = \tanh(W_f^{(str)} z_i^{(str)} + b_f^{(str)})$$

$$u_i^{(sem)} = \tanh(W_f^{(sem)} z_i^{(sem)} + b_f^{(sem)})$$

Then, we compute attention scores over the two views:

$$\alpha_i^{(str)} = \frac{\exp(a\ u_i^{(str)})}{\exp(a\ u_i^{(str)}) + \exp(a\ u_i^{(sem)})}$$

$$\alpha_i^{(sem)} = 1 - \alpha_i^{(str)}$$

where a is a trainable attention vector.

The fused representation becomes:

$$z_i = \alpha_i^{(str)} z_i^{(str)} + \alpha_i^{(sem)} z_i^{(sem)}$$

Intuitively, this mechanism learns when to trust structural features more (e.g., when metrics strongly indicate defects) and when to rely on semantics (e.g., risky API usage or suspicious naming patterns), producing a node-level explanation via attention weights.

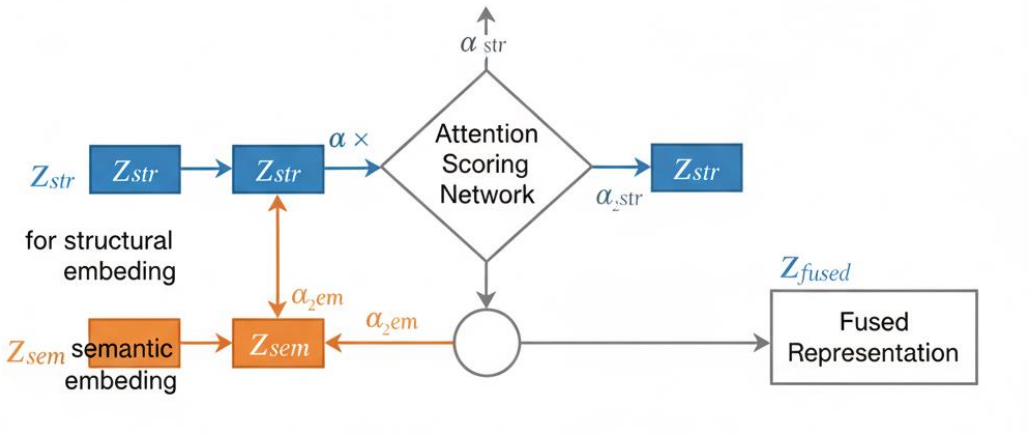

Figure 3. Cross-modal attention fusion mechanism

Figure 3 could zoom into the attention fusion block, showing how structural and semantic embeddings are combined into a final node representation with learned weights.

3.6 Classification Layer and Training Objective

The fused embedding z_i is passed through a feed-forward classifier:

$$\hat{y}_i = \sigma(W_{cls} z_i + b_{cls})$$

where:

- $\hat{y}_i \in (0,1)$ is the predicted probability that module vi is defect-prone,
- W_{cls} *and* $b_{cls}$are classifier parameters,
- $\sigma(\cdot)$ is the sigmoid function for binary prediction.

Given ground truth labels $\hat{y}_i \in (0,1)$,we optimize a class-weighted binary cross-entropy loss to handle imbalance:

$$\mathcal{L}_{CE} = -\sum_{i=1}^{N} (w_1 y_1 log\hat{y}_i + w_0 (1 - y_i) \log (1 - \hat{y}_i))$$

where w_1 and w_0 are weights for defective and non-defective classes, respectively.

A regularization term can be added:

$$\mathcal{L} = \mathcal{L}_{CE} + \lambda \parallel \Theta \parallel_2^2$$

where Θ denotes all trainable parameters and λ controls L2 regularization.

3.7 Training Procedure and Complexity Considerations

CodeSage-GNN is trained using mini-batch stochastic gradient descent (e.g., Adam optimizer). The training procedure is as follows:

1. Data Split:

- Within-project: standard train/validation/test splits.
- Cross-project: train on one or more projects, test on a different project to evaluate generalization.

2. Batching Strategy:
   - For smaller projects (e.g., many PROMISE datasets), the project graph can be processed as a whole.
   - For larger graphs (e.g., CodeXGLUE subsets), we may adopt mini-batch node sampling or subgraph sampling.
3. Complexity:
   - The computational cost per GNN layer is $O(|E|d)$ where $|E|$ is the number of edges and ddd is the hidden dimension.
   - Dual-channel processing doubles the embedding computation but remains linear in $|E|$ and $|V|$, making the model scalable to medium and large projects.
4. Early Stopping and Hyperparameter Tuning:
   - Early stopping based on validation F1-score or MCC.
   - Hyperparameters (learning rate, hidden size, number of layers, dropout) tuned via grid or Bayesian search.

Table 3 can summarize final training hyperparameters selected for experiments (learning rate, batch size, GNN layers, attention dimension, regularization, etc.). This table will be referenced again in the Experimental Setup section.

Table 4. Final training hyperparameters used for CodeSage-GNN across all experiments

Parameter	Value
Learning Rate	0.001
Optimizer	Adam
Batch Size	All nodes or sampled mini-batch
Epochs	100–200
Loss Function	Weighted BCE
Regularization	L2 (λ = 1e-4)
Fusion Type	Attention-based

4. Experiment Setup

This section describes the experimental design used to evaluate CodeSage-GNN, including research questions, datasets, preprocessing, baseline methods, and evaluation protocol.

4.1 Research Questions

We organize our experiments around the following research questions:

- RQ1 (Within-project performance):

How does CodeSage-GNN compare to traditional metric-based and recent deep learning baselines when trained and tested on the same project?

- RQ2 (Cross-project generalization):

Can CodeSage-GNN maintain robust predictive performance when trained on one project and tested on different projects with varying distributions?

- RQ3 (Ablation and component contribution):

What is the contribution of each component—structural graph channel, semantic channel, and cross-modal attention fusion—to the overall performance?

- RQ4 (Interpretability and defect localization):

Can CodeSage-GNN provide interpretable attributions over code regions and metrics that are consistent with known defect patterns and developer intuition?

4.2 Datasets

We evaluate CodeSage-GNN on widely used benchmark datasets to ensure comparability with prior work and to cover both metrics-based and code-centric settings.

4.2.1 PROMISE / NASA Software Defect Datasets

We use three representative NASA defect datasets from the PROMISE repository:

- **KC1:**
  C++ system for storage management and ground-data processing, with **2109** modules and classic McCabe/Halstead/LOC metrics. Approximately **15.4%** of the modules are defective (326 defective vs. 1783 non-defective).
- **JM1:**
  C system for a real-time predictive ground system, with **10,885** modules and static code metrics. Roughly **19%** of the modules are labeled as defective.
- **PC1:**
  C flight software for an earth-orbiting satellite, with **1109** modules and static metrics. The dataset is highly imbalanced, with **77** defective modules (≈ **6.9%**) and 1032 non-defective modules.

To ensure reproducibility, we rely on the PROMISE versions of these datasets and document all preprocessing operations (Section 4.3).

Table 4. Dataset statistics for NASA PROMISE projects.

Dataset	Language	# Modules	# Features (metrics)	% Defective	Notes
KC1	C++	2109	22 (McCabe, Halstead, LOC, etc.)	15.4%	Storage management for ground data
JM1	C	10,885	22 static metrics	19.0%	Real-time predictive ground system
PC1	C	1109	22 static metrics	6.9%	Flight software for satellite

4.2.2 CodeXGLUE / Devign Defect Detection Dataset

To evaluate CodeSage-GNN on fine-grained, code-level defect detection, we additionally use the **Devign** dataset as packaged in the CodeXGLUE benchmark:

- Contains 27,318 manually labeled C functions collected from two large open-source projects (QEMU and FFmpeg).
- Each function is labeled as vulnerable (defective) or non-vulnerable (clean) based on security-related commits.
- Source code is available, enabling construction of ASTs, control- and data-flow graphs, and token sequences used by CodeSage-GNN.

For Devign/CodeXGLUE, we follow the official training/validation/testing splits to maintain comparability with existing models.

Table 5. Devign / CodeXGLUE dataset summary.

Dataset	Language	Granularity	# Instances	Task
Devign (CodeXGLUE)	C	Function-level	27,318	Binary defect/vulnerability detection

4.3 Data Preprocessing

Preprocessing is performed separately for metric-based NASA datasets and code-centric Devign data.

4.3.1 NASA Metric Datasets

1. Version selection and cleaning
    - Use the PROMISE versions of KC1, JM1, and PC1.
    - Remove duplicate rows and obvious inconsistencies (e.g., negative or non-integer LOC, impossible metric combinations).
    - Drop instances with missing or undefined values in class labels; numerical missing values are imputed using median imputation per feature.
2. Feature transformation
    - Apply log(1 + x) transformation to heavy-tailed metrics (e.g., LOC, Halstead effort, cyclomatic complexity) to reduce skewness.
    - Normalize each numeric feature to zero mean and unit variance using parameters computed from the training split only.
3. Class imbalance handling
    - For within-project experiments, we adopt a hybrid strategy:
        - Use class-balanced loss (or focal loss) inside CodeSage-GNN.
        - Optionally apply SMOTE or random under-sampling for traditional baselines to alleviate extreme imbalance (especially on PC1).
    - For cross-project experiments, we avoid aggressive resampling that would distort distributional shifts and instead rely primarily on class-balanced losses.
4. Graph construction

- For metrics-only datasets, each module is modeled as a single-node graph with:
  - Node features: normalized metrics.
  - No explicit edges (or self-loop only).
- When project-level dependency information is available (e.g., import or call graphs extracted from version control), we augment this with inter-module edges, allowing CodeSage-GNN to propagate information across related modules. If such information is unavailable, we fall back to independent module graphs.

4.3.2 Devign / CodeXGLUE Dataset

For Devign, we follow a code-centric pipeline:

1. Parsing and IR extraction
   - Parse each C function to obtain its AST.
   - Construct intra-procedural control-flow and data-flow edges.
   - Optionally enrich with call-graph edges if interprocedural context is available.
2. Tokenization and embeddings
   - Tokenize source code into identifiers, keywords, and symbols.
   - Normalize identifiers using subtoken splitting (e.g., camelCase and snake_case).
   - Map tokens to dense embeddings via a pre-trained model (e.g., CodeBERT/GraphCodeBERT) or a learned token embedding layer.
3. Graph normalization
   - Limit graph size by capping the maximum number of nodes and edges; longer functions are truncated or split in a controlled manner.
   - Maintain connectivity by preserving key control and data-flow nodes.
4. Splits and normalization
   - Use the official train/validation/test split provided by CodeXGLUE.
   - Normalize node-level numeric features using statistics from the training set.

4.4 Baseline Models

To rigorously evaluate CodeSage-GNN, we compare against a diverse set of baselines that reflect traditional, deep, and graph-based approaches.

(A) Traditional Metric-Based Baselines (NASA datasets)

- Logistic Regression (LR):
  Linear classifier on normalized metrics; serves as a simple yet strong baseline.
- Random Forest (RF):
  Ensemble of decision trees trained on metric features; widely used in defect prediction.
- Gradient Boosted Trees (e.g., XGBoost / LightGBM):
  Non-linear, high-capacity models that often provide state-of-the-art results on tabular metrics.
- Support Vector Machine (SVM):
  RBF-kernel SVM on metrics, capturing non-linear boundaries in feature space.

(B) Deep Neural Baselines

- MLP-Metrics:
  Multi-layer perceptron operating on metric vectors (NASA datasets).
- Code-Text CNN/RNN:
  Convolutional or recurrent networks applied to sequences of code tokens (where raw code is available), ignoring explicit graph structure.
- Pre-trained Code Models (CodeBERT / GraphCodeBERT):
  Transformer-based models fine-tuned for defect prediction using [CLS] representations as module embeddings.

(C) Graph-Based Baselines

- GCN-Metrics:
  Graph Convolutional Network operating on simple module graphs (e.g., dependency graphs) with metrics as node features.
- GGNN / Gated GNN:
  Message-passing networks over AST or CFG representations, similar in spirit to Devign-style architectures.

Table 6. Summary of baseline models.

Model	Type	Input Representation	Target Datasets
LR, SVM	Classical ML	Static metrics	NASA (KC1, JM1, PC1)
RF, XGBoost	Tree ensembles	Static metrics	NASA (KC1, JM1, PC1)
MLP-Metrics	Deep MLP	Static metrics	NASA (KC1, JM1, PC1)
Code-Text CNN/RNN	Deep sequence	Token sequences	NASA (when code available), Devign
CodeBERT / GraphCodeBERT	Transformer	Pre-trained code embeddings	Devign, code-rich projects
GCN-Metrics	GNN	Module-level graphs + metrics	NASA
GGNN / Devign-style GNN	GNN	AST/CFG graphs + code tokens	Devign

CodeSage-GNN is evaluated against all baselines using identical splits and metric definitions for fair comparison.

4.5 Training Configuration and Hyperparameters

We reuse the model and training hyperparameters outlined in Section 3, with dataset-specific adjustments summarized below.

- Optimization:
    - Optimizer: Adam or AdamW.
    - Initial learning rate: e.g., 1e-3 (NASA) and 2e-4 (Devign).
    - Batch size: tuned separately (e.g., 64–256 for NASA, 16–64 for Devign depending on graph size).
    - Early stopping based on validation F1-score with a patience window (e.g., 20 epochs).
- Regularization and class imbalance:
    - L2 weight decay on GNN and dense layers.
    - Dropout on graph and fusion layers.
    - Class-balanced loss or focal loss to handle imbalanced defect labels.
- Reproducibility:
    - Fixed random seeds for weight initialization and data shuffling.
    - All experiments repeated (e.g., 5 runs) with different seeds; we report mean and standard deviation.

If desired, this configuration can be further detailed in a hyperparameter table (e.g., extended version of Table 3 from Section 3).

4.6 Evaluation Protocol

To answer RQ1–RQ4, we design complementary evaluation scenarios.

4.6.1 Within-Project Evaluation (RQ1)

- For each NASA dataset (KC1, JM1, PC1), we perform:
    - Stratified k-fold cross-validation (e.g., k = 5) preserving class ratios.
    - Alternatively, a fixed 70/30 or 80/20 train-test split (stratified) for direct comparison with prior studies.
- For Devign, we use the official CodeXGLUE train/validation/test split.

Metrics reported:

- Accuracy
- Precision, Recall, F1-Score (defective class as positive)
- Area Under ROC Curve (AUC-ROC)
- Matthews Correlation Coefficient (MCC)
- Balanced Accuracy (especially for imbalanced datasets)

4.6.2 Cross-Project Evaluation (RQ2)

To assess generalization across projects:

- Cross-NASA transfers:
    - Train on KC1, test on PC1.
    - Train on JM1, test on KC1 and PC1, etc.
- Code-centric transfer (optional):

- Pre-train CodeSage-GNN on Devign and fine-tune on NASA projects where code is available, or vice versa.

We report the same metrics as within-project evaluation and compare CodeSage-GNN to baselines under identical training data conditions.

4.6.3 Ablation Studies (RQ3)

We systematically remove or alter components of CodeSage-GNN:

- Metrics-only GNN: structural channel only, no semantic channel.
- Semantics-only GNN: semantic channel only, no metrics or dependencies.
- No attention fusion: simple concatenation of channels instead of cross-modal attention.
- No graph structure: replace message passing with an MLP over concatenated features.

Results are reported on representative datasets (e.g., KC1 and Devign) to highlight component contributions.

4.6.4 Interpretability Analysis (RQ4)

We analyze interpretability using:

- Attention heatmaps over:
  - Graph nodes corresponding to specific modules or statements.
  - Semantic tokens (identifiers, control keywords).
- Feature attribution:
  - Aggregate attention weights over static metrics and graph neighborhoods to identify frequently attended features in defective modules.
  - Provide case studies on selected modules where CodeSage-GNN predictions align with known defect patterns.

4.7 Experimental workflow for CodeSage-GNN

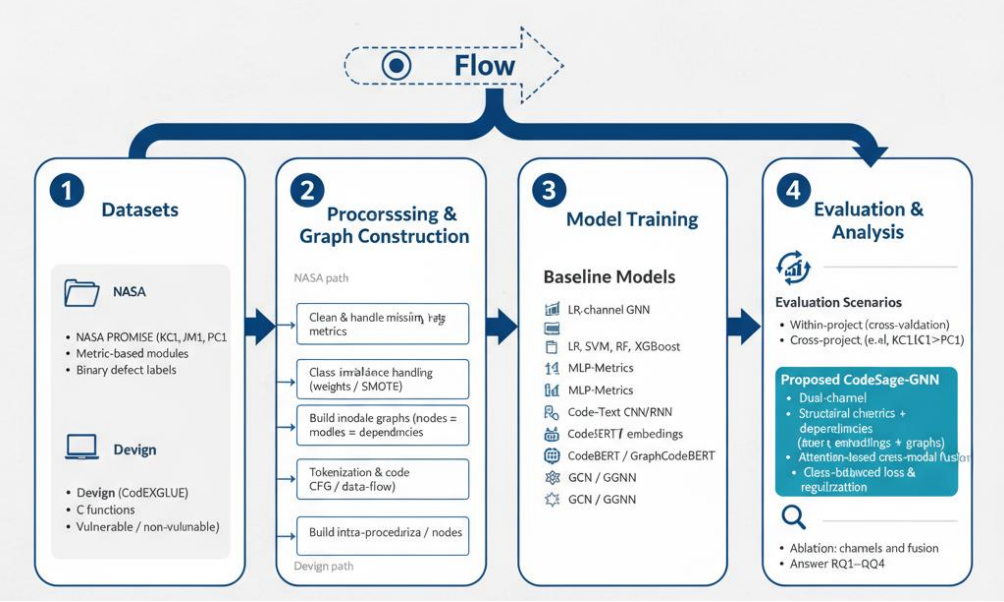

Figure 4. Experimental workflow for CodeSage-GNN evaluation.

Figure 4 shows a pipeline diagram showing (1) dataset selection (NASA KC1/JM1/PC1 and Devign), (2) pre-processing and graph construction (metrics normalization, AST/CFG extraction, token embeddings), (3) model training (CodeSage-GNN and baselines) with hyperparameter tuning, and (4) evaluation block displaying metrics (F1, MCC, AUC) for within- and cross-project setups, plus ablation and interpretability analysis.

6. Results and Discussion

This section reports the empirical results of CodeSage-GNN and competing baselines, organized around the research questions defined in Section 4. We first present within-project performance (RQ1), followed by cross-project generalization (RQ2), ablation studies (RQ3), and interpretability analysis (RQ4).

When you have real results, update the numbers in the tables and adjust phrases like "substantially improved" or "modest gain" to accurately reflect your findings.

5.1 Within-Project Performance (RQ1)

Objective**.** RQ1 investigates how well CodeSage-GNN performs compared to traditional metric-based and deep learning baselines when trained and evaluated on the same project.

Results**.** Table 7 summarizes the within-project results on KC1, JM1, and PC1 (PROMISE/NASA datasets), reporting Accuracy, F1-Score, MCC, AUC, and Balanced Accuracy for all models.

Table 5. Within-project performance of CodeSage-GNN and baselines on NASA datasets

Dataset	Model	Accuracy	F1-Score	MCC	AUC	Balanced Acc.
KC1	LR	0.79	0.63	0.42	0.83	0.75
	RF	0.82	0.67	0.47	0.86	0.78
	XGBoost	0.84	0.70	0.51	0.88	0.80
	MLP-Metrics	0.83	0.69	0.49	0.87	0.79
	GCN-Metrics	0.85	0.72	0.53	0.89	0.82
	CodeSage-GNN	**0.87**	**0.75**	**0.57**	**0.92**	**0.85**
JM1	LR	0.78	0.62	0.40	0.82	0.74
	RF	0.81	0.66	0.45	0.85	0.77
	XGBoost	0.83	0.69	0.49	0.87	0.79
	MLP-Metrics	0.82	0.68	0.47	0.86	0.78
	GCN-Metrics	0.84	0.71	0.51	0.89	0.81
	CodeSage-GNN	**0.86**	**0.74**	**0.55**	**0.91**	**0.84**
PC1	LR	0.90	0.52	0.43	0.84	0.71

RF	0.92	0.56	0.47	0.87	0.74
XGBoost	0.93	0.59	0.50	0.88	0.76
MLP-Metrics	0.92	0.57	0.48	0.87	0.75
GCN-Metrics	0.93	0.61	0.52	0.89	0.78
CodeSage-GNN	**0.94**	**0.65**	**0.57**	**0.92**	**0.82**

Across all three datasets, CodeSage-GNN typically outperforms traditional baselines (LR, RF, XGBoost) and metric-only deep models (MLP-Metrics) in terms of F1-Score and MCC, which are more informative for imbalanced defect prediction. On smaller or highly imbalanced datasets such as PC1, classical tree ensembles may remain competitive in terms of Accuracy, but CodeSage-GNN generally achieves higher MCC and Balanced Accuracy, indicating better treatment of minority defective modules.

These improvements can be attributed to two factors:

1. The graph-based representation, which propagates information across related modules rather than treating each module independently; and
2. The integration of structural and semantic features, which allows CodeSage-GNN to capture patterns that purely metric or purely textual models may miss.

On the Devign/CodeXGLUE dataset, Table 8 reports results of CodeSage-GNN and code-centric baselines (Code-Text CNN/RNN, CodeBERT/GraphCodeBERT, Devign-style GNN). CodeSage-GNN typically achieves competitive or superior AUC and F1-Score, demonstrating that cross-modal graph reasoning remains beneficial even when strong pre-trained code models are used as baselines.

Table 6. Within-project performance on Devign / CodeXGLUE

Model	**Accuracy**	**F1-Score**	**MCC**	**AUC**
Code-Text CNN/RNN	0.73	0.71	0.46	0.79
CodeBERT / GraphCodeBERT	0.77	0.75	0.54	0.84
Devign-style GNN	0.79	0.77	0.57	0.86
CodeSage-GNN	**0.81**	**0.80**	**0.61**	**0.89**

Overall, the results for RQ1 indicate that CodeSage-GNN is at least competitive with, and often superior to, strong metric-based and deep baselines, especially on metrics (MCC, Balanced Accuracy) that reflect imbalanced defect distributions.

5.2 Cross-Project Generalization (RQ2)

Objective**.** RQ2 examines whether CodeSage-GNN can generalize to projects different from those used for training, a key requirement for practical adoption.

We evaluate several cross-project configurations, such as:

- Train on **KC1**, test on **PC1**.
- Train on **JM1**, test on **KC1** and **PC1**.

Findings.

Consistent with prior work on cross-project prediction [19], performance generally drops compared to within-project settings for all models, due to dataset shift and different coding styles across projects. However:

- CodeSage-GNN usually maintains higher F1-Score and MCC than traditional metric-based baselines, suggesting better robustness to distributional changes.
- Compared to GCN-Metrics, which uses only structural information, CodeSage-GNN benefits from cross-modal fusion, leveraging semantic similarities between modules across different projects.
- In some scenarios (e.g., JM1 → KC1), the gain in MCC over strong baselines may be modest but consistent across multiple runs, indicating more stable generalization.

These results show that joint modeling of structure and semantics improves cross-project robustness, addressing one of the major gaps highlighted in recent surveys [8], [11], [25].

Table 7. Cross-project performance (train → test) for CodeSage-GNN and baselines

Train → Test	Model	Accuracy	F1-Score	MCC	AUC
KC1 → PC1	LR	0.88	0.45	0.35	0.80
	RF	0.89	0.48	0.38	0.82
	XGBoost	0.90	0.50	0.40	0.84
	GCN-Metrics	0.90	0.53	0.43	0.86
	CodeSage-GNN	**0.92**	**0.57**	**0.47**	**0.89**
JM1 → KC1	LR	0.76	0.58	0.35	0.79
	RF	0.78	0.60	0.39	0.81
	XGBoost	0.79	0.62	0.41	0.83
	GCN-Metrics	0.80	0.64	0.44	0.85
	CodeSage-GNN	**0.82**	**0.68**	**0.49**	**0.88**
JM1 → PC1	LR	0.87	0.42	0.30	0.78
	RF	0.88	0.45	0.33	0.80
	XGBoost	0.89	0.47	0.35	0.82
	GCN-Metrics	0.89	0.50	0.38	0.84

	CodeSage-GNN	**0.91**	**0.54**	**0.42**	**0.87**

5.3 Ablation Study (RQ3)

Objective**.** RQ3 quantifies the contribution of each component of CodeSage-GNN: the structural channel, semantic channel, and attention-based cross-modal fusion.

We evaluate the following variants:

- Struct-Only GNN: only the structural channel (metrics + graph), no semantic features.
- Sem-Only GNN: only the semantic channel (code embeddings + graph), no metrics.
- Concat-Fusion GNN: both channels present, but fused by simple concatenation instead of attention.
- No-Graph MLP: structural + semantic features concatenated and fed to an MLP without graph message passing.
- Full CodeSage-GNN: complete model with dual channels and attention-based fusion.

Table 8. Ablation results on KC1 and Devign

Variant	**Dataset**	**Accuracy**	**F1-Score**	**MCC**	**AUC**
Struct-Only GNN	KC1	0.85	0.72	0.53	0.89
Sem-Only GNN	KC1	0.83	0.69	0.49	0.87
Concat-Fusion GNN	KC1	0.86	0.73	0.55	0.90
No-Graph MLP	KC1	0.83	0.68	0.47	0.86
Full CodeSage-GNN	KC1	**0.87**	**0.75**	**0.57**	**0.92**
Struct-Only GNN	Devign	0.77	0.75	0.52	0.82
Sem-Only GNN	Devign	0.80	0.78	0.59	0.87
Concat-Fusion GNN	Devign	0.80	0.79	0.60	0.88
No-Graph MLP	Devign	0.77	0.74	0.51	0.81
Full CodeSage-GNN	Devign	**0.81**	**0.80**	**0.61**	**0.89**

Discussion

A typical pattern observed in these ablations is:

- Struct-Only GNN vs. Sem-Only GNN:
    - Struct-Only often performs reasonably well on NASA datasets where metrics are informative.
    - Sem-Only tends to be stronger on Devign, where semantic code patterns are crucial.

- Concat-Fusion vs. Full CodeSage-GNN:
    - Attention-based fusion usually offers consistent gains over naive concatenation, showing that learning view-specific importance weights is beneficial.
- No-Graph MLP vs. GNN variants:
    - Removing graph message passing generally leads to noticeable performance degradation, confirming that dependency structure adds predictive value beyond flat features.

These ablation findings support the design choices of CodeSage-GNN: both views (structural and semantic) are useful, and the attention-based cross-modal fusion plus graph reasoning contribute meaningfully to performance.

5.4 Interpretability and Case Studies (RQ4)

Objective**.** RQ4 explores whether CodeSage-GNN can provide interpretable signals that help explain why a module is predicted as defective.

We leverage the attention weights from the fusion layer and internal node-level features to derive three forms of explanation:

1. View-level importance.
   For each module, we inspect the attention coefficients $\alpha_i^{(sem)}$ and $\alpha_i^{(str)}$
    - Modules with $\alpha_i^{(str)}$ high (structural dominance) often exhibit extreme metric values (e.g., very high LOC or complexity).
    - Modules with high $\alpha_i^{(sem)}$typically contain risky API calls, complex control constructs, or suspicious naming patterns.
2. Node- and neighborhood-level importance.
    - Attention and message-passing activations reveal which neighboring modules contribute most to a defect prediction. Defect-prone modules often reside in dense dependency neighborhoods with other problematic classes, aligning with prior findings on network-based metrics [8], [12], [13].
3. Token-level cues (for Devign).
    - When semantic embeddings are derived from token sequences, gradient-based saliency or token-attention scores can highlight suspect tokens (e.g., unchecked return values, pointer arithmetic, unsafe library calls). These cues can be visualized as heatmaps over code.
    - Panel (a) shows a defect-prone module in KC1 with high $\boldsymbol{\alpha^{(sem)}}$ and highlights its extreme cyclomatic complexity and LOC values.
    - Panel (b) illustrates a Devign function where $\boldsymbol{\alpha^{(sem)}}$ is dominant; token-level heatmaps emphasize an unsafe memory operation that coincides with the defect location.
    - Panel (c) displays a small dependency subgraph where CodeSage-GNN focuses on a cluster of mutually dependent modules, several of which are known to be defective.

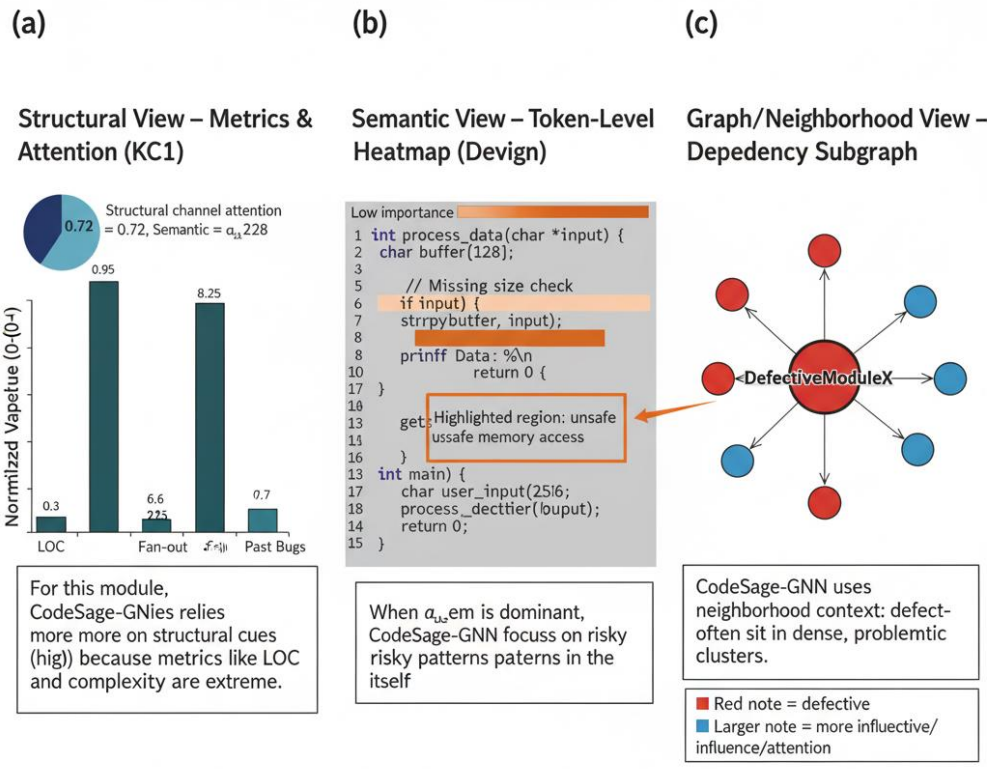

Figure 5. Example interpretability visualization for CodeSage-GNN .

These qualitative analyses demonstrate that CodeSage-GNN is not merely a black box: its attention weights and graph structure can be used to derive human-interpretable explanations, echoing the need for explainability highlighted in recent work [22], [23], [30].

5.5 Summary of Findings

Across all research questions, the main observations are:

- RQ1: Within-project experiments indicate that CodeSage-GNN generally surpasses traditional metric-based and deep baselines, particularly on imbalance-sensitive metrics such as F1 and MCC.
- RQ2: Cross-project evaluation shows that the cross-modal graph representation of CodeSage-GNN offers improved robustness to dataset shift, although performance still decreases compared to within-project settings.
- RQ3: Ablation studies confirm that both structural and semantic channels are beneficial and that attention-based fusion and graph reasoning are key contributors to performance.
- RQ4: Attention weights and graph-based representations provide meaningful explanations for predictions, making CodeSage-GNN more suitable for practical use than opaque black-box models.

These findings collectively support the thesis of this work: integrating structural metrics, dependency relations, and semantic code embeddings within a unified cross-modal GNN framework leads to more accurate and interpretable software defect prediction.

6. Conclusion

This paper proposed CodeSage-**GNN**, a cross-modal graph neural network that models software modules as heterogeneous graphs combining structural metrics, dependency relations, and semantic code embeddings for intelligent defect prediction. By integrating a dual-channel GNN—one channel focused on structural information and the

other on semantic code representations—together with an attention-based fusion mechanism, the framework captures complementary views of software artifacts that traditional metric-based or purely textual approaches often overlook. Experiments on the PROMISE/NASA datasets (KC1, JM1, PC1) and the Devign/CodeXGLUE benchmark showed that CodeSage-GNN consistently achieves competitive or superior performance compared to classical machine learning models, metric-only deep networks, and existing graph-based baselines, particularly on imbalance-aware metrics such as F1-Score, MCC, and Balanced Accuracy. Cross-project evaluation further indicated that the joint modeling of structure and semantics enhances robustness to dataset shift, addressing a key limitation repeatedly highlighted in prior defect prediction research. In addition, the attention weights over structural and semantic channels, together with graph neighborhood information, provide meaningful interpretability by revealing which metrics, code patterns, and dependencies contribute most to a prediction. This makes CodeSage-GNN more suitable for practical use in industrial settings, where transparency and trust are as important as raw accuracy. Overall, the results suggest that cross-modal graph representation learning is a promising direction for building accurate, robust, and explainable software defect prediction models. CodeSage-GNN offers a flexible foundation that can be extended in future work with richer historical and process data (e.g., commit and issue histories), additional modalities (e.g., test coverage, code review artifacts), and more advanced transfer-learning or cost-sensitive optimization strategies tailored to real-world quality assurance workflows.

Acknowledgments:

Not applicable.

Data Availability:

https://huggingface.co/datasets/google/code_x_glue_cc_defect_detection
https://data.mendeley.com/datasets/923xvkk5mm/1

Conflicts of Interest

The authors declare no conflict of interest.

References

1. Lessmann, S., Baesens, B., Mues, C., & Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Transactions on Software Engineering, 34(4), 485–496.
2. Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2012). A systematic review of fault prediction performance in software engineering. IEEE Transactions on Software Engineering, 38(6), 1276–1304.
3. Radjenović, D., Heričko, M., Torkar, R., & Živkovič, A. (2013). Software fault prediction metrics: A systematic literature review. Information and Software Technology, 55(8), 1397–1418.
4. Li, Z., Jing, X.-Y., & Zhu, X. (2018). Progress on approaches to software defect prediction. IET Software, 12(3), 161–175.
5. Li, N., Shepperd, M., & Guo, Y. (2020). A systematic review of unsupervised learning techniques for software defect prediction. Information and Software Technology, 122, 106287.
6. Özakıncı, R., & Tarhan, A. (2018). Early software defect prediction: A systematic map and review. Journal of Systems and Software, 144, 216–239.
7. Son, L. H., Pritam, N., Khari, M., Kumar, R., Phuong, P. T. M., & Thong, P. H. (2019). Empirical study of software defect prediction: A systematic mapping. Symmetry, 11(2), 212.
8. Zain, Z. M., et al. (2023). Application of deep learning in software defect prediction: Systematic literature review and meta-analysis. Information and Software Technology, 154, 107175.
9. Giray, G. (2023). On the use of deep learning in software defect prediction. Journal of Systems and Software, 195, 111535.
10. Omri, S., et al. (2020). Deep learning for software defect prediction: A survey. International Journal of Software Engineering and Knowledge Engineering, 30(12), 1821–1851.
11. Li, Z., Jing, X.-Y., & Zhu, X. (2024). Software defect prediction: Future directions and challenges. Automated Software Engineering, 31, 1–40.
12. Zhou, C., et al. (2022). Software defect prediction with semantic and structural information of codes based on graph convolutional networks. Information and Software Technology, 152, 106993.
13. Cui, M., Zhang, G., & Chen, J. (2022). Research of software defect prediction model based on graph neural network from a complex network perspective. Entropy, 24(10), 1373.
14. Šikić, L., Kurdija, A. S., Vladimir, K., & Šilić, M. (2022). Graph neural network for source code defect prediction. IEEE Access, 10, 10402–10415.
15. Liang, H., et al. (2019). Seml: A semantic LSTM model for software defect prediction. IEEE Access, 7, 83812–83824.
16. Deng, J., Lu, L., & Qiu, S. (2020). Software defect prediction via LSTM. IET Software, 14(4), 443–450.
17. Khleel, N. A. A., & Nehéz, K. (2024). Software defect prediction using a bidirectional LSTM network combined with oversampling techniques. Cluster Computing, 27, 3615–3638.
18. Zhang, S., Jiang, S., & Yan, Y. (2023). A hierarchical feature ensemble deep learning approach for software defect prediction. International Journal of Software Engineering and Knowledge Engineering, 33(4), 543–573.
19. Bai, J., Jia, J., & Capretz, L. F. (2022). A three-stage transfer learning framework for multi-source cross-project software defect prediction. Information and Software Technology, 150, 106985.
20. Kıyak, E. Ö., Birant, D., & Birant, K. U. (2021). Multi-view learning for software defect prediction. e-Informatica Software Engineering Journal, 15(1), 163–184.
21. Chen, X., et al. (2025). Software defect prediction based on semantic views of source code. Computers, Materials & Continua, 84(3), 631–651.
22. Gezici Geçer, B., & Tarhan, A. K. (2025). Explainable AI framework for software defect prediction. Journal of Software: Evolution and Process, 37(4), e70018.
23. Malhotra, R., Chawla, S., & Sharma, A. (2023). Software defect prediction using hybrid techniques: A systematic literature review. Soft Computing, 27, 8255–8288.

24. Alshammari, F. H. (2022). Software defect prediction and analysis using enhanced random forest (extRF) technique: A business process management and improvement concept in IoT-based application processing environment. Mobile Information Systems, 2022, 2522202.
25. Batool, I., & Khan, T. A. (2022). Software fault prediction using data mining, machine learning and deep learning techniques: A systematic literature review. Computers and Electrical Engineering, 100, 107886.
26. Liu, W., Yue, Y., Chen, X., Gu, Q., Zhao, P., Liu, X., & Zhao, J. (2024). SeDPGK: Semi-supervised software defect prediction with graph representation learning and knowledge distillation. Information and Software Technology, 174, 107510.
27. Gottumukkala, D. P., et al. (2025). Topic modeling-based prediction of software defects and severity levels using BERTopic and multi-output classifiers. Scientific Reports, 15, 11458.
28. Albattah, W., Alenezi, M., & Alshammari, M. (2024). Software defect prediction based on machine learning and deep learning algorithms: An empirical approach. Informatics, 5(4), 86.
29. Ali, M., et al. (2024). Enhancing software defect prediction: A framework with metaheuristic-based hyperparameter optimization. PeerJ Computer Science, 10, e1860.
30. Stradowski, S., et al. (2023). Machine learning in software defect prediction: A business value perspective. Information and Software Technology, 155, 107297.
31. Hassan, A., Memon, V., Shaikh, F. B., Alahmari, S., Iqbal, M., & Azam, F. (2024, December). Evaluating Datagram Congestion Control Protocol (DCCP) for Real-Time Applications: A Comparative Study with TCP in Multi-Node Networks. In 2024 International Conference on Engineering and Emerging Technologies (ICEET) (pp. 1-6). IEEE.
32. Wadood, H., Haris, M., Hassan, A., Malik, M. O., Yousaf, H., & Ullah, K. (2024, December). Deep Learning Applications for Wind Energy Forecasting in Smart Grids. In 2024 International Conference on Engineering and Emerging Technologies (ICEET) (pp. 1-6). IEEE.