GNNCONTEXT: GNN-based Code Context Prediction for Programming Tasks

Xiaoye Zheng, Zhiyuan Wan, *Member, IEEE*, Shun Liu, Kaiwen Yang, David Lo, *Fellow, IEEE*, and Xiaohu Yang

Abstract—A code context model comprises source code elements and their relations relevant to a programming task. The capture and use of code context models in software tools can benefit software development practices, such as code navigation and search. Prior research has explored approaches that leverage either the structural information of code or interaction histories of developers with integrated development environments to automate the construction of code context models. However, these approaches primarily capture shallow syntactic and lexical features of code elements, with limited ability to capture contextual and structural dependencies among neighboring code elements. In this paper, we propose GNNCONTEXT, a novel approach for predicting code context models based on Graph Neural Networks. Our approach leverages code representation learning models to capture both the syntactic and semantic features of code elements, while employing Graph Neural Networks to learn the structural and contextual information among neighboring code elements in the code context models. To evaluate the effectiveness of our approach, we apply it to a dataset comprising 3,879 code context models that we derive from three Eclipse open-source projects. The evaluation results demonstrate that our proposed approach GNNCONTEXT can significantly outperform the state-of-the-art baseline for code context prediction, achieving average improvements of 62.79%, 56.60%, 73.50% and 81.89% in mean reciprocal rank, top- 1, top-3, and top-5 recall rates, respectively, across predictions of varying steps. Moreover, our approach demonstrates robust performance in a cross-project evaluation setting. Our code is publicly available at https://github.com/ZXXYy/CodeContextModel.

Index Terms—Code context models, programming tasks, developers, context prediction, graph neural networks

1 INTRODUCTION

DEVELOPERS spend a substantial amount of time navigating and understanding the relevant code of a software system when they perform software programming tasks. Meanwhile, in their mind, they implicitly form *code context models*, which consist of source code elements and relations between those elements relevant to specific tasks [1]. The explicit capture and use of even a portion of code context models in software tools can benefit developers and software development projects [2], such as supporting code search activities [3], facilitating code recommendations [4], [5], and improving the quality of code changes made to software systems [6]. Fig. 1a shows an example code context model of a bug-fixing task in the Mylyn project.

Previous studies propose a series of tools to help developers explicitly capture code contexts during programming tasks, such as Concern Graphs [7], Code Bubbles [8], and Code Basket [9], as well as leverage structural information in code or interaction histories of developers with integrated development environments (IDEs) to enable the automatic formation of code context models [4], [5], [10], [11]. In a most recent study, Wan et al. [11], [12] propose an approach that utilizes both structural information in code and the interaction histories of developers for proactively formation of code context models. The approach learns abstract patterns of how developers explore structurally connected code elements during programming tasks in a software system, and leverages these learned patterns to predict code elements in future code context models based on the the current interactions of developers with IDEs. The learned patterns, represented as directed graphs, leverage stereotype roles [13] to generalize the syntactic behaviors of code elements in their nodes, while capturing structural dependencies between these code elements, such as *calls* and *declares*, in their edges. Nonetheless, these patterns capture shallow syntactic and lexical information of code elements in their nodes, as well as limited contextual and structural information between neighboring code elements through their edges.

In this paper, we propose GNNCONTEXT, a novel approach for predicting code context models based on Graph Neural Networks (GNNs). Our approach uses code representation learning models to capture both the syntactic and semantic features of code elements in code context models, while utilizing GNNs to learn structural and contextual interdependencies among neighboring code elements. Specifically, GNNCONTEXT constructs abstract syntax trees (ASTs) and call graphs (CGs) for the relevant code within a code context model to capture both syntactic and semantic information. Next, program slicing [14] is performed from the point of interest (i.e., the seed node in the code context model) to identify candidate nodes for prediction, thereby expanding the code context model. The nodes in the expanded model are then embedded into low-dimensional

- *Xiaoye Zheng, Zhiyuan Wan, Shun Liu, Kaiwen Yang, and Xiaohu Yang are with the State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou 310013, China E-mail: {xiaoyez, wanzhiyuan, liushun0311, kwyang, yangxh}@zju.edu.cn.*
- *David Lo is with the School of Information Systems, Singapore Management University University, Singapore 188065. E-mail: davidlo@smu.edu.sg.*
- *Zhiyuan Wan is the corresponding author.*

Manuscript received; revised.

vectors, which serve as input to the GNN model. To enhance learning, contrastive learning loss is applied in the GNN model to generate distributed representations of the nodes. Consequently, GNNCONTEXT leverages the well-trained GNN model to predict future code elements for a given task, based on the initial code context model as input.

To evaluate the effectiveness of our approach, we curate a dataset of 3,879 code context models from the interaction histories of three Eclipse open-source projects, Mylyn[1], Platform[2], and Plug-in Development Environment (PDE)[3], spanning 12, 11 and 8 years of history, respectively. The experimental results demonstrate that GNNCONTEXT significantly outperforms the state-of-the-art baseline [11] on our dataset, achieving average improvements of 62.79%, 56.60% %, 73.50% and 81.89% in mean reciprocal rank, top-1, top-3, and top-5 recall rates, respectively, across step-1, step-2, and step-3 predictions. GNNCONTEXT also outperforms existing SOTA approaches for bug localization [15] and API recommendation [16] when evaluated on our dataset. In addition, our approach exhibits robust performance in cross-project settings, with cross-project code context predictions even outperforming within-project predictions, particularly when the training data originates from projects with extensive and highly relevant historical data.

In summary, we make the following contributions:

- We propose a novel approach to predicting code context models, which integrates code representation learning models to capture both the syntactic and semantic information of code elements in code context models, and GNNs to learn the structural and contextual information among neighboring code elements. Our code is publicly available at https://github.com/ZXXYy/CodeContextModel.
- We demonstrate that our approach can predict code context models effectively, outperforming the state-of-the-art baseline in within-project settings, and achieving robust performance in cross-project settings.
- We provide a dataset comprising 3,879 code context models to facilitate future investigations by others.[4] The dataset builds upon prior work [11], extending it with the inclusion of two additional projects and an expansion of 1,992 code context models.

The remainder of this paper is organized as follows. We begin by describing preliminaries and motivation in Section 2. We then describe our approach in Section 3, and evaluate the effectiveness of the approach for code context prediction in Section 4. Next, we discuss the implications of results in Section 5, and threats to validity in Section 6. Finally, we describe related work in Section 7, before concluding in Section 8.

2 PRELIMINARIES AND MOTIVATION

2.1 Usage Scenario

As software developers work on programming tasks, they spend considerable time navigating code to understand relevant parts of the codebase, forming code context models in their mind that represent source code elements and their relationships [1], [11]. In large-scale software projects, where developers must navigate numerous source code files to handle their programming tasks, the complexity of programming tasks significantly increases [17], [18]. Consequently, the complexity of programming tasks results in more intricate code context models, which, in turn, raise the cognitive load for developers. As a result, code context prediction becomes especially valuable when code context models are distributed across multiple source files in large codebases, as it can help reduce the cognitive burden and enhance the productivity of the developers.

Fig. 1 presents a typical usage scenario in which code context prediction approaches assist software practitioners in performing programming tasks. The task aims to fix a bug report `169123`[5] as recorded in the Mylyn project. During the bug-fixing task, a developer interacts with the IDE and navigates through the source code to understand the relevant code of the task. As a result, the developer locates two code elements potentially relevant to the task – a *class* code element, `AbstractRepositoryTaskEditor`, as well as a *method* code element, `createLabel`, which are captured by code context prediction approaches as an initial code context model as shown in Figure 1a. By learning patterns from interaction histories as shown in Fig. 1b, code context prediction approaches proactively form a future code context model (Fig. 1c) by recommending likely code elements of interest based on the initial code context model.

2.2 Motivating Example

We illustrate how the state-of-the-art (SOTA) approach [11] works for code context prediction in the usage scenario in Fig. 1, and list our observations to motivate our approach. The SOTA approach enables code context prediction by utilizing the topological patterns of code elements learned from the interaction histories of a project. Specifically, the approach first learns abstract topological patterns based on the stereotype roles of code elements, and then applies these learned patterns to predict code context models for a given task through graph pattern matching.

On the one hand, the SOTA approach leverages stereotype roles [13] to summarize the syntactic behaviors of code elements in code context models, learning patterns from the generalized code elements assigned with stereotype roles for code context prediction. However, we observe that these stereotype roles are derived from a set of predefined rules, which capture shallow syntactic and lexical information of code elements when summarizing the behavior of methods and classes. These pre-defined rules encompass 17 stereotype roles for method elements, as well as 13 stereotype roles for class elements. For instance, as shown in Fig 1d, the `createLabel` method takes *PROPERTY* as its primary stereotype role, as it returns the local variable `label` (Rule 1). Additionally, it is labeled with *COLLABORATOR* as its secondary stereotype role, considering that all of its parameters (`composite` and `attribute`) are non-primitive variables (Rule 2). The observation suggests that capturing deeper syntactic and semantic information of code elements

1. https://eclipse.dev/mylyn/
2. https://wiki.eclipse.org/Platform
3. https://www.eclipse.org/pde
4. https://zenodo.org/records/13790748
5. https://bugs.eclipse.org/bugs/show_bug.cgi?id=169123

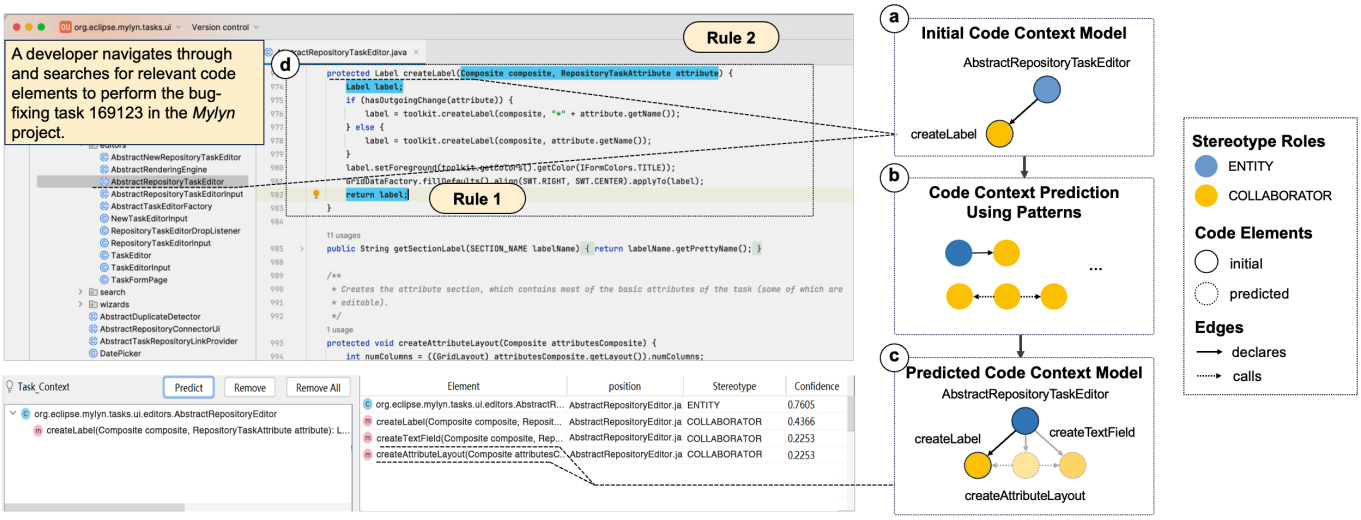


Fig. 1: Usage scenario of code context prediction.

may enhance the performance of code context prediction beyond what is achieved by the current SOTA approach. Consequently, we consider adopting code representation learning models in our approach.

On the other hand, the SOTA approach learns abstract patterns represented as directed graphs, facilitating code context prediction through pattern matching. In these patterns, nodes correspond to stereotype roles, while edges capture the syntactic relationships between code elements, such as *declares* and *calls*. However, we observe that the structural interdependencies reflected by these edges do not account for structural interdependencies of multi-hop neighboring code elements. The observation suggests that incorporating more comprehensive structural and contextual information from multi-hop interdependencies may further improve the performance of code context prediction beyond the current SOTA approach. As a result, we consider utilizing GNNs to capture both structural and contextual information among code elements of code context models in our approach.

2.3 Graph Neural Networks

GNNs constitute a specialized family of deep learning models specifically tailored for graph-structured data, such as the Graph Convolutional Network (GCN) [19], Relational GCN (RGCN) [20], GraphSAGE [21] and Graph Attention Network (GAT) [22]. GNN models usually leverage a neighborhood aggregation mechanism, whereby the representation of each node in a graph is progressively refined through the incorporation of information from its neighboring nodes. The mechanism allows GNNs to effectively capture and leverage the underlying structural interdependencies of graph data.

In the realm of software engineering, a variety of tasks require capturing the intricate structural interdependencies among elements in code, aiming at better comprehending the program semantics. Due to this natural alignment, GNNs are particularly well-suited to tackle such tasks, demonstrating significant potential across a diverse range of tasks, such as code completion [23], [24], code summarization [25], vulnerability detection [26], [27], [28], [29], [30], [31], code summarization [25], [32], and code recommendation [33], [34]. For instance, RepoHyper [23] employs GraphSAGE [21] to encode the semantic code graph structure for solving repository-level code completion tasks. MVD+ [26] proposes a flow-sensitive GNN to detect memory-related vulnerabilities. Ling et al. [33] utilize GNNs to capture high-order collaborative signals from API calls and recommend API usage for programmers. LeClair et al. [25] use GCN together with an RNN-based encoder to embed ASTs for code summarization.

3 Our Approach

In this section, we present the architecture design of GNNContext. Fig. 2 gives its pipeline overview consisting of two phases: the **training phase** and the **prediction phase**.

The training phase takes as input the initial code context models of programming tasks as well as their relevant source code, and produces a well-trained GNN model for code context prediction, which consists of three steps. In **Step 1**, our approach builds ASTs and CGs of source code to capture the syntactic and semantic information, and conducts program slicing [14] to generate *expanded* code context models from initial code context models (Section 3.1). In **Step 2**, our approach transforms each code element in *expanded* code context models into low-dimensional vector representations (Section 3.2). In **Step 3**, our approach makes use of both contrastive learning and GNNs to learn contextual and structural features of code context models (Section 3.3)

In the prediction phase, our approach takes as input the initial code context model of a given task, and generates the expanded code context model (**Step 1**), and conducts node embedding for the expanded model (**Step 2**) as it does in the training phase. The expanded code context model and its corresponding vector representations are then fed into the well-trained model as graph input for code context prediction.

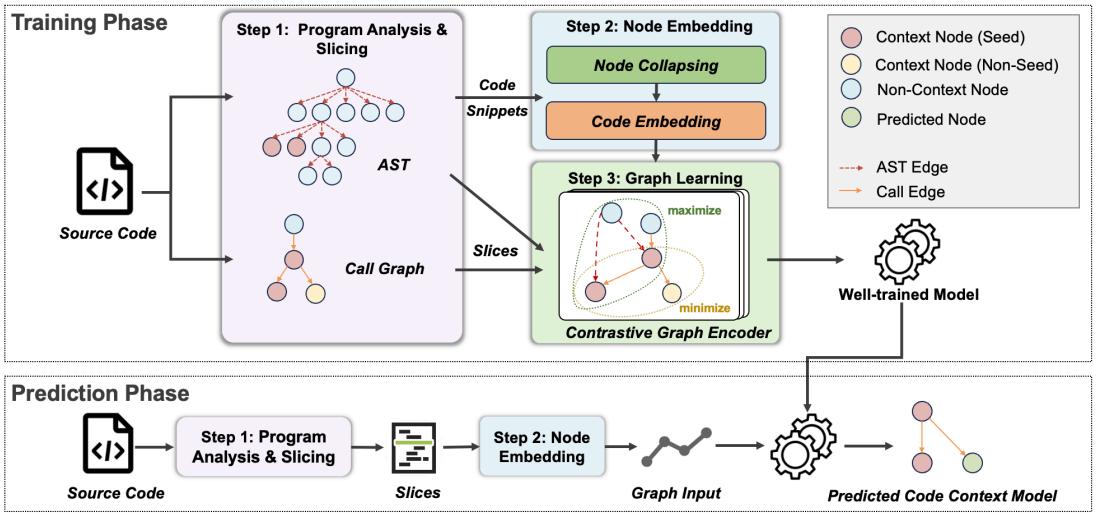


Fig. 2: GNNCONTEXT overview.

3.1 Program Analysis and Slicing

GNNCONTEXT first conducts intraprocedural and interprocedural program analysis to build ASTs and CGs from the relevant source code of programming tasks. AST reflects the syntax structure of source code files, while CG provides the control-flow information across functions in the source code files. As inspired by previous studies [35], [36], combining diverse code representations can be beneficial for deep learning models to capture the program semantics.

GNNCONTEXT then adopts program slicing [14] to perform backward and forward slicing in source code to generate expanded code context models, which starts from the code elements of interest (code elements in the initial code context model), along with the edges in ASTs and CGs of relevant source code. Previous studies indicate that source code files usually contain numerous code elements, among which, developers only access a few structurally connected or semantically relevant code elements to form code context models [1], [11]. Consequently, the program slicing considers the code elements d steps (d = 1, 2 and 3) away from the code elements in the initial code context models. Moreover, our approach labels the nodes in the expanded code context model with three types: (1) *Seed context* nodes, which exist in the initial code context model, serving as the input of a prediction model, (2) *Non-seed context* nodes, which exist in the code context model but not in the initial code context model, serving as the ground truth for a prediction model, and (3) *Non-context* Nodes, which do not exist in the code context model. Note that *context* nodes represent the nodes in a code context model, consisting of *seed context* nodes and *non-seed context* nodes.

3.2 Node Embedding

Node Collapsing. We observe a substantial imbalance in the numbers of *Context* vs. *Non-Context* nodes in expanded code context models, especially for the *field* nodes, which tend to have relatively less semantic information as compared to *method* and *class* nodes. Consequently, GNNCONTEXT collapses the *field* nodes that are declared in each *class* node into a single node. Note that the *field* nodes representing lambda expressions or anonymous inner classes are not collapsed in our approach, as they may contain crucial semantic information.

Code Embedding. GNNCONTEXT adopts the code representation learning models to generate embedding for the code snippet of each code element. We begin by tokenizing the code snippet of the code element into multiple token sequences, padding the classification token (`[CLS]`) at the start of the sequences as a start marker. Next, we employ the BGE embedding model [37], a language embedding model based on the transformer architecture that incorporates code as a textual sequence in its training data, to encode these token sequences. We then extract the embedding of the `[CLS]` token, which captures the global semantics of the code snippet, as the final embedding of the code snippet.

3.3 Graph Learning

GNNCONTEXT designs a graph learning architecture based on RGCN, utilizing contrastive learning loss to code elements in a code context more similar in the embedding space. It learns both the features of code elements and their structural interdependencies to capture the semantic and contextual information within code context models.

Model Architecture. In our architecture, we update the node embedding h_v of each node v as

$$h_i^{l+1} = \sigma\left(W_i^{(l)} h_i^{(l)} + \sum_{r \in R} \sum_{j \in N_i^r} \frac{1}{c_{i,r}} W_r^{(l)} h_j^{(l)}\right), \quad (1)$$

where $h_i^{(l)}$ denotes the representation of node i at layer l, N_i^r denotes the set of neighbor indices of node i under relation $r \in R$, $W_i^{(l)}$ denotes the learnable weight for node i at layer l, σ denotes the activation function and $c_{i,r}$ is a normalization constant. The node embedding vectors serve initial hidden representation $h^{(0)}$. The initial edge representation for edge r can be encoded by basis decomposition as

$$W_r^{(l)} = \sum_{b=1}^{B} a_{rb}^{(l)} V_b^{(l)}, \quad (2)$$

where $V_b^{(l)}$ is a set of learnable basis vectors and and $a_{rb}^{(l)}$ is also the learnable scalar weight specific to edge type and basis. Furthermore, we add inverse edges and assign different weight parameters according to different edge types, to capture contextual information of code context models.

To stabilize training, we add a self-loop for each node in the graph neural network. This maintains a consistent information flow and ensures that each node retains its features in the updated representation. With the help of the graph learning, contextual information can be captured and sensitive low information is given more attention. Moreover, we stack multiple graph convolutional layers to enable the GNN model to learn hierarchical and increasingly abstract representations of nodes, capturing complex graph structures and patterns.

Model Training. We further utilize contrastive learning loss to optimize the parameters of the GNN model, aiming at capturing the intricate differences between *Seed Context* Nodes S, *Non-Seed Context* Nodes P, and *Non-Context* Nodes N in a code context model R. The contrastive learning loss consists of two loss functions, as inspired by a prior study [38]: (i) Seed-positive pair loss function $\mathcal{L}_{pos}$, which aims to maximize the semantic similarity of S and P:

$$\mathcal{L}_{pos}(R) = \sum_{(s,p)\in S\times P} \max(0, m_{positive} - \text{sim}(v_s, v_p)) \quad (3)$$

and (ii) Seed-negative pair loss function $\mathcal{L}_{neg}$, which aims to minimize the semantic similarity of S and N:

$$\mathcal{L}_{neg}(R) = \sum_{(s,n)\in S\times N} \max(0, \text{sim}(v_s, v_n) - m_{negative}) \quad (4)$$

where v_i denotes the vector embedding of node i as output by the GNN model. $\text{sim}(v_i, v_j)$ is the cosine similarity between two vector embeddings. $m_{positive}$ and $m_{negative}$ are the hyper-parameters for the two contrastive loss functions. To this end, the contrastive learning loss function for a mini-batch $\mathcal{R}$ is

$$\mathcal{L} = \sum_{R\in\mathcal{R}} (\mathcal{L}_{pos}(R) + \mathcal{L}_{neg}(R)) \quad (5)$$

Overall, our customized loss directly enforces the relative distance between positive and negative pairs by calculating their similarity with seeds.

3.4 Code Context Model Prediction

In the prediction phase, GNNCONTEXT uses the well-trained model built in the training phase to predict code elements in the future code context model of a given task based on its initial code context model.

Specifically, our approach first generates an expanded code context models based on the initial code context model through program analysis and slicing (Section 3.1). Next, our approach embeds the nodes in the expanded code context model into low-dimensional vectors through code representation learning models (Section 3.2). Finally, our approach feeds the well-trained GNN model with the *expanded* code context model to label the non-seed nodes as *positive* or *negative* in the future code context model. For each non-seed node n in the *expanded* code context model, we calculate its similarity score SIM_n as compared to all seed nodes S in the *expanded* code context model:

$$\text{SIM}_n = \frac{1}{|S|} \sum_{s\in S} \text{sim}(v_n, v_s) \quad (6)$$

where $\text{sim}(v_i, v_j)$ is the cosine similarity between two embeddings learned by the well-trained GNN model. With respect to the similarity scores of non-seed nodes, our approach labels top k non-seed nodes as *positive*, and the remaining as *negative*. Consequently, the *positive* nodes are predicted as code elements in the future code context model.

4 EVALUATION

In this section, we focus on evaluating the effectiveness of GNNCONTEXT for code context prediction by answering the following research questions:

- **RQ1:** How does GNNCONTEXT perform in code context prediction, compared with the state of the art?
- **RQ2:** How does GNNCONTEXT perform in a cross-project setting?
- **RQ3:** To what extent do different design choices affect the performance of GNNCONTEXT?
- **RQ4:** What is the impact of different hyperparameters?
- **RQ5:** How do different design choices of the initial code context models affect prediction performance?

4.1 Dataset

4.1.1 Code Context Model Formation

To experiment with the prediction of code context models, we need a dataset of such code context models. To this end, we form a dataset of code context models[6] using the *interaction histories* captured as developers work with three open-source Eclipse projects, Mylyn, Platform and PDE. We chose to use the three projects as the data source for our investigations because (1) the three projects are widely used in previous studies on code edit recommendations, e.g., [4] and [39]; and (2) the interaction histories span over 12, 11 and 8 years in the development of Mylyn, Platform and PDE, respectively. The Eclipse Mylyn tool[7] records interaction histories as developers work on a code base. Each interaction history includes a record of the code elements that have been viewed or edited by a developer during a programming task. In line with prior work [40], which suggests that both editing and viewing histories of developers contribute to code edit recommendation, we included code elements edited or viewed by developers in the code context models, as recorded in the interaction histories in our dataset. The Eclipse Mylyn tool enables one or more interaction histories to be associated with each task performed by developers on a system, and stored with the task recorded in the Eclipse Bugzilla system.

Based on the interaction histories, we curated a dataset of code context models, by following the process aligned with that in Wan et al.'s work [11]. Specifically, for each interaction history, we first divided every two consecutive interaction events into separate working periods, provided

6. https://zenodo.org/records/13790748
7. https://www.eclipse.org/mylyn

the time gap between their occurrences exceeded three hours. Next, we extracted class, method, and field code elements from the "selection" and "edit" events in the interaction history of each working period. We further identified the structural dependencies (i.e., *declares*, *calls*, *inherits*, *implements*) between code elements by running Doxygen [41] on the corresponding snapshots of code repositories. Consequently, the extracted code elements constitute the nodes of the code context model for each working period, while the identified structural dependencies form the edges. Further details are provided in Appendix A.

As shown in Table 1, our dataset consists of 3,879 code context models, including 3,126, 446 and 307 from the Mylyn, Platform, and PDE projects, respectively. The statistics in the *Code Context Model (CCM)* column show that the sizes of code context models vary across projects, with a median of 7, 11 and 4 nodes for the Mylyn, Platform, and PDE projects, respectively. The code context models are comprised of multiple connected components, with a median of 3, 4 and 2 connected components for the Mylyn, Platform, and PDE projects, respectively, indicating that developers worked with multiple clusters of structurally connected code elements during programming tasks. The *Connected Component (CC)* column reports statistics about the range of sizes of the 13,275, 3,099 and 895 connected components that comprise the code context models of the Mylyn, Platform, and PDE projects. The statistics show that the average diameters of connected components are 0.94, 0.83 and 0.87 for the Mylyn, Platform, and PDE projects, respectively, indicating that the developers did not navigate code elements by following structural dependencies in depth during programming tasks.

4.1.2 Quality Assessment

Our dataset includes interaction histories of developers from three projects, covering a broad spectrum of programming tasks, namely bug fixing and feature enhancement [42], some of which have been utilized in previous studies on code edit recommendations for programming tasks [4] and software evolution tasks [39]. Specifically, out of the 3,126 code context models for the Mylyn project, 1,181 (37.8%) are related to feature enhancement, while 1,945 (62.2%) are associated with bug fixing. Out of the 446 code context models for the Platform project, 61 (13.6%) correspond to feature enhancement and 385 (86.3%) to bug fixing. As for the PDE project, which consists of 307 code context models, 92 (30%) are attributed to feature enhancement and 215 (70%) to bug fixing.

Furthermore, we quantified the practical value of the predicted code elements in the code context models of our dataset by examining whether these elements were edited, the number of times they were viewed, and the duration developers spent viewing them, as recorded in the interaction histories. As shown in Table 2, 54.06%, 44.83%, and 54.08% of the code elements in the context models were edited at least once for the Mylyn, Platform, and PDE projects, respectively. For the remaining viewed-only code elements, the average viewing durations were 108.75s, 108.76s, and 107.66s, with average view counts of 1.83, 1.77, and 1.65 for the Mylyn, Platform, and PDE projects, respectively. The statistics suggest that the dataset we consider as ground truth closely reflects the actual interactions of developers, highlighting the practical relevance of predicting code elements in the ground truth.

4.1.3 Preprocessing

In practice, a developer may start code search and navigation from the middle of a task. To experiment with our approach in a practical scenario, we followed the steps below to preprocess the code context models in our dataset:

Seed Generation. In this step, we aimed to generate a set of *seed* nodes for each code context model from the dataset $\mathcal{R}$, representing the corresponding initial code context model, as illustrated in Algorithm 1. Specifically, for each code context model $R \in \mathcal{R}$, we created a seed set S for it with the prediction step d as input. Specifically, considering developers may start navigating code elements in an arbitrary order during programming tasks, we randomly chose a subset of $size\ minus\ d$ code elements from R, where $size$ denotes the number of nodes in R. This process yields an initial code context model $\hat{R}$ for R, where the chosen elements serve as the *seed* nodes for experimenting with code context prediction.

Code Context Model Expansion. We followed the step of program analysis and slicing in the training phase of GNNCONTEXT (Section 3.1) to expand the initial code context models $\hat{\mathcal{R}}$. Specifically, we first performed backward and forward slicing along the AST and CG edges from the code elements in the initial code context models with depth d, where we chose d to be 1, 2 and 3. As a result, we generated three expanded code context models, R_1, R_2 and R_3, for each initial code context model $\hat{R}$ generated from the code context model R in our dataset $\mathcal{R}$.

Node Labeling. Based on the resulting *seed* nodes $\hat{R}$ of each code context model $\mathcal{R}$ in our dataset, we further labeled the nodes in the corresponding expanded code context models. Specifically, for each expanded code context model R_1, R_2 or R_3, we labeled the corresponding seed nodes within $\hat{R}$ as "*seed*", $(R - \hat{R})$ as "*positive*", and the remaining nodes as "*negative*".

Given code duplication in training and test sets may distort experimental results in machine learning models [43], [44], we measured the code duplication in our dataset as follows: (1) We identified similar code elements in each project using BLEU scores, grouping those with a BLEU score above 0.9; (2) We detected duplicate code context models across the training and test sets using Jaccard similarity, considering two code context models as duplicates if their coefficient exceeded 0.7, a commonly used threshold indicating strong similarity while allowing minor variations [45]. As a result, we identified 6, 1 and 0 duplicate code context models between training and test sets for the Mylyn, PDE, and Platform projects, respectively, accounting for 0.96%, 1.6%, and 0% of the training samples, suggesting a neglectable impact of code duplication on the experimental results.

4.2 Metrics

To measure the performance of code context prediction, we choose two metrics that are widely used in previous studies (e.g., [46], [47], [48]), Mean Reciprocal Rank (MRR), and Top-k Recall (R@k).

TABLE 1: Dataset statistics.

Project (# CCM / # CC)		Code Context Model (CCM) # Nodes	# Edges	# CC	Connected Component (CC) # Nodes	# Edges	Diameter
Mylyn (3,126 / 13,275)	Min	2	1	1	1	0	0
	Max	71	73	68	30	61	10
	Median	7	4	3	2	1	1
	Mean	10.54	8.09	4.25	2.48	1.91	0.94
	SD	9.79	9.81	4.20	2.45	3.79	1.06
Platform (446 / 3,099)	Min	2	1	1	1	0	0
	Max	119	135	80	63	117	11
	Median	11	7	4	1	0	0
	Mean	17.16	13.74	6.95	2.47	1.98	0.83
	SD	17.14	17.85	7.95	3.48	5.74	1.19
PDE (307 / 895)	Min	2	1	1	1	0	0
	Max	41	48	15	27	48	7
	Median	4	2	2	2	1	1
	Mean	6.83	5.06	2.92	2.34	1.74	0.87
	SD	6.60	7.15	2.74	2.35	3.75	1.05

TABLE 2: statistics of code elements in dataset.

Project	# Elements	# Edited Elements		Viewed-only Elements Min	Max	Median	Mean	SD
Mylyn	26,976	14,583	# Views	1	69	1	1.83	2.61
			Duration (s)	0	11,070	6	108.75	569.18
Platform	5,925	2,656	# Views	1	53	1	1.77	2.76
			Duration (s)	0	9,983	6	108.76	508.74
PDE	1,653	894	# Views	1	27	1	1.65	1.81
			Duration (s)	0	7,830	5	107.66	524.78

Algorithm 1 Seed generation.

Input: Dataset of code context models $\mathcal{R}$; prediction step d;
Output: Initial code context models $\hat{\mathcal{R}}$

```
for R ∈ R do
    size = node_number(R)
    if size > d then
        S = extract_subgraphs(R, size − d)
        R̂ = random_select(S)
        R̂ = R̂ ∪ {(R, R̂)}
    end if
end for
```

1) MRR denotes the average reciprocal ranks of the correctly predicted code elements given the initial code context models $\hat{\mathcal{R}}$, as defined in Equation 7:

$$\text{MRR} = \frac{1}{|\hat{\mathcal{R}}|}\sum_{i=1}^{|\hat{\mathcal{R}}|}\frac{1}{Rank_i} \tag{7}$$

where $Rank_i$ denotes the rank of the first correctly predicted code elements for the i-th initial code context model $\hat{R}$.

2) R@k evaluates the effectiveness of code context prediction in recalling *positive* nodes from the top k candidates in the prediction results, where k=1, 3 and 5, which is calculated as Equation 8:

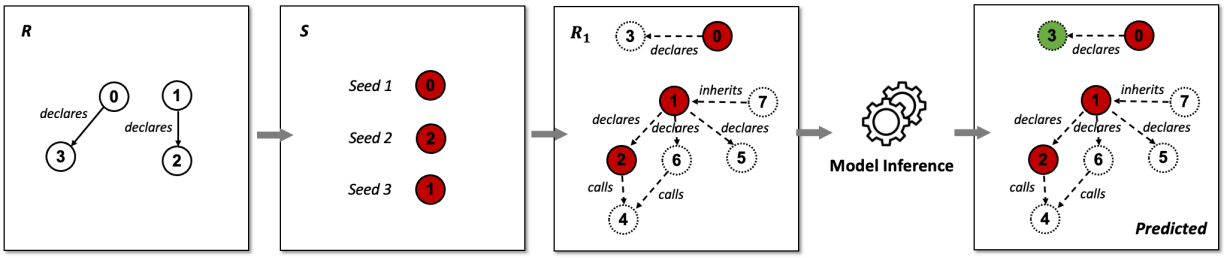
$$\text{R@k} = \frac{1}{|\hat{\mathcal{R}}|}\sum_{i=1}^{|\hat{\mathcal{R}}|}\phi(Rank_i \leq \text{k}) \tag{8}$$

where ϕ is an indicator function that equals 1 if the correctly predicted code elements for the i-th initial code context model exist in the top-k results, i.e., its rank $Rank_i$ is less than or equal to k. Note that for 2-step and 3-step predictions, the R@k metric measures whether any code elements from the initial code context models appear in the top-k results. For example, in the 3-step prediction, R@1 is 1 if any code element of the three positive ones, as labeled in the expanded code context model, ranks first in the prediction results.

4.3 Workflow in Evaluation

Figure 3 illustrates the workflow of 1-step cross-file code context model prediction of GNNCONTEXT, exemplified through a feature enhancement task[8], with its code context model R comprising two connected components distributed across two source code files. Our approach first generates the seed set S, and expands S to a depth of $d = 1$, resulting in the expanded code context model R_1. Among the nodes in R_1, nodes 0, 1, and 2 are labeled as "seed", node 3 as "positive", and the remaining nodes as "negative". Using R_1 as input, our approach computes the embedding similarities between the seed nodes and the other nodes, and subsequently ranks the nodes based on the similarities. Following model inference, node 3 is identified as the correct prediction with a rank of 2. Consequently, R@1 is $\phi(Rank \leq 1) = 0$, R@3 is $\phi(Rank \leq 3) = 1$, and MRR is $\frac{1}{Rank} = 0.5$.

8. https://bugs.eclipse.org/bugs/show_bug.cgi?id=376308



	Node	Similarity with Seeds	Rank
0	org.eclipse.mylyn.internal.resources.ui.ResourceStructureBridge	-	-
1	org.eclipse.mylyn.resources.tests.ResourceStructureBridgeTest	-	-
2	org.eclipse.mylyn.resources.tests.ResourceStructureBridgeTest.testGetProjectsInActiveContext()	-	-
3	org.eclipse.mylyn.internal.resources.ui.ResourceStructureBridge.getProjectsInActiveContext()	2.1456	2
4	org.eclipse.mylyn.resources.tests.TestProject.getProject()	1.3276	5
5	org.eclipse.mylyn.resources.tests.ResourceStructureBridgeTest.setUp()	2.0599	3
6	org.eclipse.mylyn.resources.tests.ResourceStructureBridgeTest.testGetProjectsInActiveContextWithInvalidProject()	2.4963	1
7	org.eclipse.mylyn.resources.tests.AbstractResourceContextTest	1.9411	4

Fig. 3: Example task to illustrate the workflow of code context prediction.

4.4 Baselines

To comprehensively evaluate the effectiveness of GNNCONTEXT, we compare it against three baseline approaches: the SOTA specifically designed for code context prediction [11], as well as leading SOTA approaches for bug localization [15], and API recommendation [16]. We considered approaches for bug localization and API recommendation because they share the objective of locating code relevant to a specific programming task, which aligns closely with code context prediction. We detail the adaptations made to facilitate the comparison as follows.

Code Context Prediction: We adopted the SOTA approach for code context prediction proposed by Wan et al. [11] as a baseline. Specifically, we conducted a direct comparison between the approach and GNNCONTEXT with no adaptation, given their alignment in the experimental setting.

Bug Localization: We adopted SemanticCodeBERT [15] as a baseline, which integrates semantic flow graphs with pre-trained models for bug localization. For a fair comparison, we used the pre-trained model of SemanticCodeBERT as the base, and fine-tuned it on our dataset using Momentum Contrastive Learning. Specifically, seed context nodes were used as input, non-seed context nodes as positive samples, and negative samples were randomly selected non-context nodes following the Memory Bank mechanism of SemanticCodeBERT.

API Recommendation aims to assist developers in selecting appropriate API methods or classes based on the current code context, typically with a focus on external library usage. The objective differs from code context prediction, which emphasizes suggesting the structural or semantic elements that developers are likely to view or edit next in their local codebase. To support API recommendation, prior studies have leveraged techniques such as frequent pattern mining [49], collaborative filtering [50], and more recently, learning-based models including LUPE [51] and GAPI [16], which represent the state of the art in this area. Given that the dataset preprocessing implementation of LUPE is not publicly available, we selected GAPI as the baseline of API recommendation for comparison. GAPI formulates API recommendation as a graph-based learning problem, and uses GNNs to capture high-order collaborative signals from API call interactions and project structures. To enable a fair comparison in our code context prediction setting, we made two adaptations to GAPI. First, we employed a k-step expanded graph to represent structural relationships among code elements in the initial context. Second, we randomly selected one code element as a seed node and treated the remaining context nodes as positive samples.

4.5 Implementation

We implemented GNNCONTEXT in Python using PyTorch [52], trained in a batch-wise fashion until converging and the batch size is set to 32. The number of the maximum epoch is set to 50. The dimension of the vector representation of node embedding is set to 1024. The hidden layer dimension of the graph neural network is set to 1024. Adam [53] optimization algorithm is used to train the GNN model with the learning rate of $1e$-6. Our experiments are conducted on a machine with 64 cores of 2.9GHz Intel(R) Xeon(R) Gold 6226R CPU and 1 GeForce RTX 3090 24GB GPU installed with Ubuntu 20.04 with CUDA 11.1. The training of models costs approximately 20, 49 and 282 hours for 1-step, 2-step, and 3-step predictions, respectively. The inference of models costs 0.016, 0.019, and 0.020 seconds on average for 1-step, 2-step, and 3-step predictions, respectively. The average inference time of less than 0.02 seconds demonstrates the feasibility of our approach for real-time use.

4.6 Experiments

4.6.1 RQ1: Effectiveness

To evaluate the effectiveness of GNNCONTEXT, we compare it with the SOTA approach [11].

TABLE 3: Evaluation results of GNNCONTEXT (with and without optimization) compared with three baselines.

Approach	1-Step Prediction				2-Step Prediction				3-Step Prediction			
	MRR	R@1	R@3	R@5	MRR	R@1	R@3	R@5	MRR	R@1	R@3	R@5
SOTA proposed by Wan et al. [11]	0.291	0.230	0.306	0.349	0.397	0.348	0.394	0.416	0.343	0.285	0.353	0.379
SemanticCodeBERT [15]	0.292	0.205	0.303	0.371	0.252	0.119	0.305	0.380	0.250	0.133	0.282	0.352
GAPI [16]	0.317	0.205	0.433	0.529	0.529	0.377	0.634	0.714	0.480	0.346	0.533	0.673
GNNCONTEXT w/o node collapsing	0.529	0.419	0.573	0.653	0.563	0.458	0.615	0.686	0.503	0.369	0.570	0.660
GNNCONTEXT w/ node collapsing	0.560	0.449	0.610	0.681	0.612	0.510	0.660	0.753	0.509	0.393	0.558	0.646

Setup. Regarding the dataset, we utilized the code context models from the Mylyn project, which is aligned with Wan et al.'s work [11]. Given the inherent sequential nature of our dataset, we initially arranged the Mylyn dataset in chronological order prior to partitioning. Subsequently, we allocated the first 80% of the dataset for training purposes, and reserved the final 20% for testing. We further subdivided the training set into a training subset and a validation subset with a 9:1 ratio during the training phase of the GNN model.

We implemented the SOTA approach and explored its parameters to replicate the performance levels reported in the original work [11]. In particular, we focused on the *MinSupp* parameter, which represents the minimum support threshold for pattern discovery. After evaluating a range of values between 0.01 and 0.1, we identified that setting *MinSupp* to 0.015 produced a set of 144 abstract patterns, which closely aligns with the 142 patterns reported in the original work.

Furthermore, we considered two variations of our approach in the comparison: (1) GNNCONTEXT without node collapsing optimization, and (2) GNNCONTEXT with node collapsing optimization (see Section 3.3 for details of node collapsing). This allows for a more comprehensive evaluation of the impact of node collapsing optimization on the overall performance.

Results. Table 3 presents the results of a comprehensive comparison of the performance of GNNCONTEXT in code context prediction, both with and without node collapsing optimization, against three baselines.

In comparison to the SOTA approach for code context prediction, GNNCONTEXT exhibits a substantial improvement across all four performance metrics (MRR, R@1, R@3, and R@5) and three prediction steps (1-step, 2-step, and 3-step), encompassing a total of 12 evaluation scenarios. Specifically, GNNCONTEXT achieves MRR scores of 0.529, 0.563, and 0.503, and R@5 scores of 0.653, 0.686, and 0.660 for 1-step, 2-step, and 3-step predictions, respectively. The experimental results demonstrate that GNNCONTEXT consistently outperforms the baseline in code context prediction across varying prediction steps, achieving average improvements of 62.79%, 56.60%, 73.50% and 81.89% in MRR, R@1, R@3, and R@5, respectively. One possible explanation for why 2-step prediction outperforms both 1-step and 3-step predictions is that it strikes an optimal balance between prediction difficulty and input size. Specifically, 2-step prediction targets one of two positive code elements with an expanded context model containing a sufficient number of code elements, making the prediction task less challenging than 1-step prediction. At the same time, 2-step prediction reduces the noise introduced by irrelevant code elements in the larger expanded code context models used in 3-step prediction.

In comparison to the leading SOTA approaches for bug localization and API recommendation, GNNCONTEXT significantly outperforms SemanticCodeBERT, which achieves MRR scores of only 0.292, 0.252, and 0.250 across the three prediction steps. GNNCONTEXT outperforms GAPI in 1-step, 2-step, and 3-step predictions, particularly excelling in the MRR and R@1 metrics. Specifically, for 1-step prediction, GNNCONTEXT achieves substantial improvements of 66.9% in MRR and 104.4% in R@1 compared to GAPI, which is likely due to the integration of historical interaction data in GNNCONTEXT. Nonetheless, for 2-step and 3-step predictions, GAPI surpasses GNNCONTEXT in R@5, indicating that the significance of historical interaction data diminishes with larger code contexts as the number of prediction steps increases. The comparison results suggest the potential to enhance the performance of GNNCONTEXT in multi-step code context predictions through a more nuanced integration of structural code information and historical interaction data.

Furthermore, the node collapsing optimization enhances the performance of GNNCONTEXT, yielding improvements of 5.8%, 8.7%, and 1.2% in MRR scores for 1-step, 2-step, and 3-step predictions, respectively. The experimental results indicate the effectiveness of merging *field* nodes during node collapsing, which leads to improvements in the performance of code context prediction. The improvements can be attributed to two key factors: first, reducing the numbers of nodes in the expanded code context models helps alleviate the imbalance between positive and negative nodes; second, consolidating *field* nodes with less semantic information simplifies the structure of expanded code context models, allowing the GNN model more effectively capture meaningful patterns.

Answer to RQ1: Our approach demonstrates a significant improvement over the SOTA baseline across 1-step, 2-step, and 3-step predictions, achieving average improvements of 62.79%, 56.60%, 73.50% and 81.89% in MRR, R@1, R@3, and R@5, respectively. Additionally, node collapsing optimization further boosts the performance of our approach, increasing MRR scores by 5.8%, 8.7%, and 1.2% for 1-step, 2-step, and 3-step predictions, respectively. GNNCONTEXT also outperforms the leading SOTA approaches for bug localization and API recommendation.

TABLE 4: Evaluation results on cross-project code context prediction.

Case	Category	Training Set	Test Set	MRR	R@1	R@3	R@5
Case 1	Within-Project Baselines	PDE	PDE	0.5310	0.3548	0.6452	0.7742
Case 2		Platform	Platform	0.5119	0.3371	0.6180	0.7303
Case 3	Cross-Project Predictions	Mylyn	Platform	0.5898	0.4646	0.6657	0.7450
Case 4	with Smaller Training Data	Mylyn	PDE	0.6035	0.4774	0.6914	0.7531
Case 5	Cross-Project Predictions	Mylyn+PDE	Platform	0.5973	0.4844	0.6544	0.7394
Case 6	with Larger Training Data	Mylyn+Platform	PDE	0.6047	0.4650	0.6914	0.7654

4.6.2 RQ2: Cross-Project Prediction

To evaluate the practical value of GNNCONTEXT, we conducted experiments in a cross-project setting to analyze how different datasets impact the performance of our approach in code context prediction.

Setup. We designed six experimental cases, as detailed in Table 4: (1) Cases 1 and 2 serve as within-project prediction baselines, where the GNN model is trained using data from the same project. Note that the Mylyn project is excluded from the within-project setting, as the experiment is identical to that in RQ1. (2) Cases 3 and 4 represent cross-project predictions, differing in test sets, but using the Mylyn project as the training set. The Mylyn project is selected as the training set due to its sufficient data volume. (3) In Cases 5 and 6, we further expanded the training sets in a cross-project setting as compared to Cases 3 and 4, respectively, aiming to evaluate the impact of additional cross-project training data on prediction performance. For each project, the data was split chronologically, with the first 80% used for training and the remaining 20% reserved for testing. Additionally, the training set was further divided into a training set and a validation set in a 9:1 ratio for each experimental case.

Results. Table 4 presents the results of 1-step code context prediction across six experimental cases. For each case, the training set, test set, and performance metrics (MRR, R@1, R@3, R@5) are listed.

Within-Project Baselines: In Cases 1 and 2, the performance of within-project predictions serves as the baseline. When the GNN model is trained and tested on the same project, it achieves solid performance, with an MRR of 0.5310 and R@5 of 0.7742 for PDE (Case 1), as well as an MRR of 0.5119 and R@5 of 0.7303 for Platform (Case 2). The experimental results indicate that, even within the same project, the model is capable of predicting code context model with relatively high effectiveness.

Cross-Project Predictions with Smaller Training Data: Case 3 and 4 involve cross-project predictions, where the model trained on data from the Mylyn project is tested on the Platform and PDE projects, respectively. The two cases indicate the ability GNN model to generalize when applied to a different project. In Case 3, the MRR increases to 0.5898, and R@5 to 0.7450, outperforming the within-project baseline in Case 2, demonstrating that the Mylyn project provides sufficient information for predicting relevant code elements in code context models of the Platform project. Similarly, Case 4 achieves an MRR of 0.6035 and R@5 of 0.7531 for PDE, surpassing the baseline performance in Case 1. The experimental results indicate that cross-project code context prediction can be highly effective when the training data of GNN models is sufficiently rich and relevant.

Cross-Project Predictions with Larger Training Data: In Case 5 and 6, the training set is expanded by combining Mylyn with PDE for Case 5 and with Platform for Case 6. The two cases aim to test whether increasing the size of the training set with data from another project improves cross-project predictions. The experimental results indicate slight improvements in performance compared to Case 3 and 4. For example, in Case 5, where the model is trained on "Mylyn + PDE" and tested on Platform, the MRR rises to 0.5973, slightly higher than the 0.5898 seen in Case 3. Similarly, in Case 6, the MRR improves to 0.6047 when testing on PDE, compared to the 0.6035 in Case 4. The experimental results suggest that adding more data, particularly from a different project, marginally improves the predictive power of GNN models.

We also made several counterintuitive observations. In Cases 1 and 2, with smaller training sets, the prediction outperforms the within-project prediction for the Mylyn project (0.681 in Table 3) in terms of R@5, despite the larger training set in the latter case. The observation suggests that, contrary to intuition, smaller training sets may yield better performance in code context prediction, indicating that model generalization and reduced overfitting may be more critical than simply increasing the training data size. Furthermore, in Cases 3 and 4, where Mylyn serves as the training set and Platform and PDE are used as test sets for cross-project predictions, the model outperforms the 1-step within-project prediction for the Mylyn project across all metrics (Table 3). This counterintuitive observation suggests that cross-project prediction could be more effective than within-project prediction, potentially due to statistical bias introduced by the relatively small test sets for the PDE and Platform projects.

> **Answer to RQ2:** GNNCONTEXT exhibits promising performance in cross-project settings. Cross-project code context predictions even outperform within-project predictions, especially when the training data comes from a project with rich and highly relevant historical data. Expanding the training set with additional data yields modest performance gains.

4.6.3 RQ3: Ablation Study

We altered different components of GNNCONTEXT to evaluate their individual contributions and to examine how various design decisions impact the performance of GNNCONTEXT, as measured by the evaluation metrics. We

used the same dataset as in RQ1 (Section 4.6.1) for the ablation study.

Setup. To evaluate the impact of different code representations in node embedding on code context prediction, we replace our code encoder (BGE embedding) with CodeBERT embedding [54], a widely-used bimodal pre-trained model for both programming and natural languages. This allows us to evaluate the contribution of alternative embedding methods to the overall performance of GNNCONTEXT.

For graph learning, we first construct a variant of our approach that excludes the GNN graph encoder. The variant directly compares the node embeddings of *seed* and *non-seed* nodes, recommending the *non-seed* nodes most similar to the *seed* nodes for code context prediction. Next, we replace our graph encoder (RGCN) with three well-established GNN models—**GCN** [19], **GAT** [22], and **GraphSAGE** [21] to evaluate the contribution of graph learning in enhancing code context prediction. GCN scales linearly with the number of graph edges, learning hidden representations that incorporate both the local graph structure and node features. GAT introduces attention mechanisms, allowing the model to assign different importance weights to each neighboring node. GraphSAGE samples a fixed-size subset of neighbors for each node, aggregating their features to improve scalability without relying on all neighbors.

Results. Table 5 presents how each node embedding technique and GNN model combination performs in terms of MRR, R@1, R@3, and R@5.

CodeBERT: Using no GNN at all results in the lowest MRR score of 0.333, suggesting that while CodeBERT is a strong bimodal pre-trained model for programming and natural languages, it lacks the ability to fully capture the structural information inherent in code context models when used alone. The introduction of GNN models significantly boosts performance, with RGCN being the most effective, achieving an MRR of 0.476. The experimental results suggest the importance of incorporating relational information in code context predictions, as RGCN is specifically designed to model such information. On the other hand, GCN shows a moderate improvement (MRR 0.367), while GAT and GraphSAGE underperform, with MRR scores of 0.324 and 0.347, respectively. The experimental results indicate that attention mechanisms in GAT and sampling strategies in GraphSAGE are less effective when used with CodeBERT embeddings, potentially because they do not fully exploit the pre-trained representations.

BGE Embedding: In contrast, BGE embedding consistently outperforms CodeBERT across all GNN models. Even without a GNN model, BGE embedding achieves an MRR of 0.442, which is higher than 0.333 of CodeBERT in the same setup, demonstrating that BGE embedding is better suited for representing code context even without additional graph-based learning. The combination of BGE embedding with RGCN produces the highest MRR, reaching 0.529. GCN and GraphSAGE also perform relatively well with BGE embedding, achieving MRR scores of 0.453 and 0.449, respectively. Interestingly, GAT yields the lowest MRR of 0.334, mirroring its poor performance with CodeBERT, indicating that attention-based mechanisms may not provide a significant advantage in the code context prediction task when compared to other GNN architectures.

TABLE 5: Evaluation results on different node embedding techniques and GNN models.

Node Embedding	GNN Models	MRR	R@1	R@3	R@5
CodeBERT	w/o GNN	0.333	0.218	0.370	0.445
	RGCN	0.476	0.368	0.510	0.571
	GCN	0.367	0.250	0.408	0.470
	GAT	0.324	0.266	0.403	0.478
	GraphSAGE	0.347	0.232	0.374	0.461
BGE Embedding	w/o GNN	0.442	0.315	0.498	0.581
	RGCN	0.529	0.419	0.573	0.653
	GCN	0.453	0.323	0.517	0.596
	GAT	0.334	0.271	0.424	0.485
	GraphSAGE	0.449	0.321	0.502	0.607

TABLE 6: Effect of number of GNN layers in graph learning.

# Layers	MRR	R@1	R@3	R@5
1	0.505	0.384	0.559	0.648
2	0.519	0.406	0.567	0.658
3	0.529	0.419	0.573	0.653
4	0.528	0.413	0.575	0.665
5	0.544	0.440	0.576	0.660

Answer to RQ3: BGE embedding consistently outperforms CodeBERT across all GNN models, indicating the inherent ability of BGE embedding to capture code context features. RGCN tends to be the most effective model for both embeddings, with BGE+RGCN achieving the highest MRR (0.529), highlighting the capacity of RGCN to integrate node embeddings with relational graph structures.

4.6.4 RQ4: Sensitivity Analysis

We performed a further investigation into the impact of different hyperparameters on the performance of GNNCONTEXT. For consistency, we utilized the same dataset as in RQ1 (Section 4.6.1).

Setup. The investigation focused on two aspects of hyperparameters: (1) the number of RGCN layers, ranging from 1 to 5, which denotes propagation iterations, and (2) the margin hyperparameter in the loss function used for contrastive graph learning. Specifically, we evaluated the performance of our approach with the positive contrastive loss margin ($m_{positive}$) in the range of 0.8 to 1, and the negative contrastive loss margin ($m_{negative}$) from 0 to 0.2.

Results. Table 6 presents the experimental results of how varying the number of GNN layers impacts the performance of our approach in terms of MRR, R@1, R@3, and R@5. The results demonstrate a clear trend where increasing the number of GNN layers contributes to improvements in the ability of our approach to predict code contexts. As the GNN model progresses from one to three layers, there is a steady improvement in MRR and R@1, reflecting enhanced effectiveness in identifying the most relevant code elements. The improvement also indicates that deeper GNN architectures can better capture hierarchical and complex relationships in graphs, which are crucial for accurately modeling code contexts. However, the improvements in R@3 and R@5 are relatively modest as the number of layers increases, especially beyond three layers. The plateauing effect in improvements

TABLE 7: Effect of different margin values in the loss function of contrastive graph learning.

$m_{positive}$	$m_{negative}$	MRR	R@1	R@3	R@5
1.0	0	0.529	0.419	0.573	0.653
	0.1	0.528	0.419	0.567	0.652
	0.2	0.532	0.424	0.575	0.648
0.9	0	0.528	0.419	0.568	0.655
	0.1	0.530	0.417	0.584	0.663
	0.2	0.532	0.421	0.586	0.661
0.8	0	0.528	0.414	0.579	0.658
	0.1	0.531	0.416	0.587	0.658
	0.2	0.535	0.424	0.589	0.668

suggests that, while the GNN model in our approach gains in depth and complexity with additional layers, its ability to capture relevant nodes in code context models beyond the top-ranked predictions does not scale proportionally. In summary, the optimal number of layers for our approach appears to be between three and five, where the tradeoff between depth and generalization is balanced.

Table 7 presents the impact of different margin values for positive and negative samples in the loss function used for contrastive graph learning on the performance of our approach. On the one hand, smaller $m_{positive}$ values (closer to 0.8) generally yield better recall, particularly in R@3 and R@5, indicating that overly large positive margins may cause our approach to separate positive and negative samples too aggressively, reducing its ability to generalize across a larger set of relevant code elements. On the other hand, increasing $m_{negative}$ values generally improves performance, especially for MRR and R@1, suggesting that a higher negative margin helps our approach better discriminate between positive and negative samples, leading to more effective retrieval of relevant code elements in code context models. Consequently, the configuration with $m_{positive} = 0.8$ and $M_{negative} = 0.2$ yields superior performance, indicating that the margin combination optimizes the separation of positive and negative samples, thus preserving robust performance across the evaluation metrics.

Answer to RQ4: Increasing the number of GNN layers enhances the performance of GNNCONTEXT, with peak MRR and R@1 achieved at five layers. However, the gains in recall metrics (R@3 and R@5) plateau beyond three layers. Reducing positive margins enhances R@3 and R@5, whereas raising negative margins boosts MRR and R@1. By employing a configuration of five stacked GNN layers, along with a positive margin of 0.8 and a negative margin of 0.2, GNNCONTEXT achieves optimal performance across the evaluation metrics.

4.6.5 RQ5: Effect of Initial Code Context Model Design

In this RQ, we investigated the impact of different choices of initial code context models on the 1-step code context prediction performance of GNNCONTEXT, using the identical dataset utilized in RQ1 (Section 4.6.1).

Setup. We considered three design choices for the initial code context models, with the setup for each detailed below:

TABLE 8: Impact of access order of code elements in initial code context models.

Setting	MRR	R@1	R@3	R@5
Chronological order	0.529	0.408	0.578	0.665
Random order #1	0.529	0.419	0.573	0.653
Random order #2	0.521	0.404	0.568	0.658
Random order #3	0.522	0.413	0.563	0.653

TABLE 9: Impact of incorporation of developers' previous experience into initial code context models.

Setting	MRR	R@1	R@3	R@5
Experience informed	0.308	0.176	0.328	0.473
Non-experience informed	0.734	0.649	0.786	0.840

1) Access order of code elements. We compared the prediction performance of GNNCONTEXT using two selection strategies for the seed elements in a code context model R, which consists of $size$ code elements: (1) We selected the first $size-1$ elements according to the chronological order in the corresponding interaction history; (2) We randomly selected $size-1$ code elements, with the process repeated three times.

2) Model size. We compared the prediction performance of GNNCONTEXT with initial context models of varying sizes, where 10% to 90% of the code elements in each code context model R were randomly selected, with a 10% interval. The varying model sizes correspond to different stages of a programming task: smaller models (10-30%) simulate the early stages, while larger models (70-90%) represent the later stages.

3) Incorporation of previous experience of individual developers. We built initial code context models based on what developers had already accessed within the previous 14 days from the developing time of each code context model, which represents a typical development cycle in agile software development [55]. Specifically, for each code context model, we selected code elements that appeared in both the code context model and the recent 14-day interaction histories of the developer as the initial code context model. For comparison, we also built non-experience-based initial code context models by randomly selecting the same number of code elements as the corresponding experience-based initial code context models from each code context model.

Results. Table 8 presents the evaluation results regarding the impact of the access order of code elements in the initial code context models on the prediction performance of GNNCONTEXT. The prediction performance remains largely consistent across varying access orders, suggesting that the order in which code elements are accessed has minimal effect on the performance of our approach. A possible explanation is that developers begin with any code element relevant to a task and navigate bidirectionally between related elements, thereby diminishing the significance of their access order.

Figure 4 presents the evaluation results regarding the impact of initial code context model sizes on the prediction performance of GNNCONTEXT. The top-k recall values exhibit a consistent rise-and-fall trend as the size of the

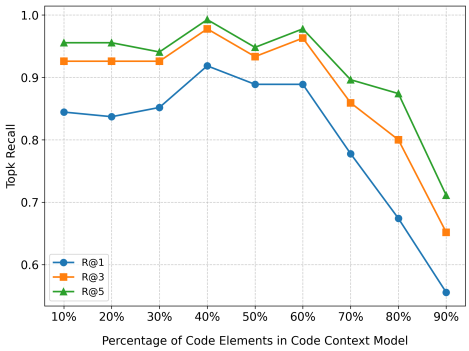


Fig. 4: Impact of initial code context model sizes.

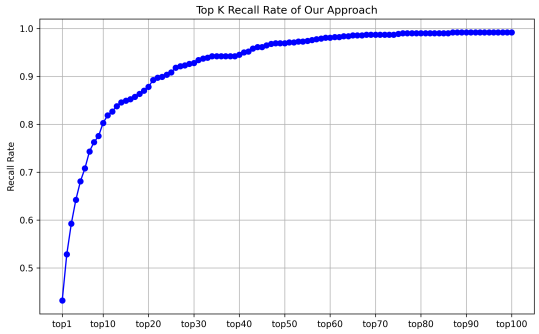


Fig. 5: Top-k recall rates of code context prediction in GNNCONTEXT.

initial code context model increases. For smaller initial code context models (10%-30%), recall values (R@1, R@3, R@5) improve with increasing model size, indicating the potential effectiveness of our approach in the early stage of a programming task. As the model size reaches 40%-60%, recall values peak, reflecting the optimal performance of our approach. However, beyond this range, prediction performance begins to stabilize; further increases in model size yield diminishing returns. When the initial code context model size exceeds 60%, recall values decline, with R@1 experiencing a notable drop. Despite the increased information provided by larger initial code context models, our approach becomes less focused, struggling to prioritize relevant code elements.

Table 9 presents the evaluation results regarding the impact of incorporating developers' previous experience into initial code context models on the prediction performance of GNNCONTEXT. The results indicate that the experience-informed setting for initial code context models consistently underperforms across all metrics, compared to the non-experience-informed setting, suggesting that the incorporation of developers' prior experience does not improve, and may even hinder the performance of our approach. While prior experience of developers might seem beneficial, it may not always align with the specific task at hand, resulting in reduced focus on the most relevant code elements, thereby hindering the ability of our approach to make accurate predictions.

Answer to RQ5: The order in which code elements are accessed has a negligible impact on the performance of GNNCONTEXT. The top-k recall values demonstrate a consistent rise-and-fall trend as the size of the initial code context model increases. The integration of developers' prior experience does not lead to performance improvements and may, in fact, impede the performance of GNNCONTEXT.

5 DISCUSSION

We reflect on the reasons behind the effectiveness of our approach, and identify opportunities for improvement.

5.1 Context Nodes Difficult to Recall

We investigated how well our approach can recall relevant context nodes in the code context models as the number of recommendations (k) increases. For each value of k (k = 1, 10, 20, ..., 100), we computed the recall rate R@k, which is detailed in Fig. 5. The y-axis ranges from 0.4 to 1.0, where a value of 1.0 indicates perfect recall, meaning that all context nodes are retrieved. The plot in Fig. 5 starts with a lower recall rate slightly above 0.4 for top-1 results, and progressively increases to near 1.0 as k approaches 100, indicating better recall rates with more top-k results. Nonetheless, our approach does not achieve perfect recall at k = 100 (R@100=0.992), suggesting that there are still some context nodes that are not retrieved.

We further examined the five tasks with context nodes difficult to recall from the test set (623 tasks) from the Mylyn project. First, we observed larger sizes of expanded code context models for the five tasks as compared to the remaining in the test set, with an average of 248 nodes (minimum: 168, median: 261, maximum: 283). We then categorized the tasks with respect to the syntactic features of their context nodes difficult to recall into two types: (1) A *function* node, which is declared in a *class* node (seed) that owns numerous functions (4 tasks); and (2) A *function* node, which is called by a *function* node (seed) that calls numerous functions (1 task). The context nodes difficult to recall tend to have similar embeddings as the relative non-context nodes of the corresponding code context models due to their similarity in their syntactic features. Future work could focus on enhancing the discriminative power of embeddings for functions that are declared in the same class or called by the same function. This might involve developing more sophisticated embedding techniques or incorporating additional contextual information to better distinguish between highly similar *context* and *non-context* nodes.

5.2 More Granular Node Classification

We explored the reasons why our approach outperforms the SOTA approach in node embedding. Our hypothesis was that our approach captures richer features of code elements in the code context models through effective code representation learning. Specifically, from the Mylyn dataset used in RQ1, we selected the *class* nodes labelled with `COMMAND` as their stereotypes in the SOTA approach. We applied a kNN clustering algorithm to these nodes [56], iterating through cluster numbers from 2 to 10 to determine

the optimal cluster number. Consequently, two distinct clusters were identified for the *class* nodes with the COMMAND stereotype, with a Silhouette score of 0.972 [57], suggesting that our approach enables a more granular classification of nodes in the code context models as compared to the SOTA approach. Future work could further explore advanced code representation learning models to capture more nuanced representations of code elements in code context models, potentially incorporating additional information such as code dependencies or execution flow.

5.3 Cross-file Code Context Prediction

Navigating multiple source code files without direct structural dependencies presents a considerable challenge for developers during programming tasks. To understand the potential of GNNCONTEXT in such a challenging scenario, we evaluated its performance on 344 code context models that include cross-file code elements from the Mylyn project. GNNCONTEXT achieves R@1 of 0.372, R@3 of 0.497, R@5 of 0.573, and MRR of 0.468, which are significantly lower than the corresponding metrics for the remaining code context models comprising within-file code elements, where R@1 = 0.477, R@3 = 0.667, R@5 = 0.753, and MRR = 0.603. The evaluation results indicate that GNNCONTEXT performs reasonably well in both within-file and cross-file scenarios, but there is significant room for improvement in the future, particularly in handling cross-file code contexts where its performance lags behind within-file predictions.

6 THREATS TO VALIDITY

The threats to ***internal validity*** stem from two factors. First, the expansion of code context models relies on ASTs and CGs, which may miss certain structural relationships in source code (e.g., data-flow). Additionally, static analysis can overestimate *call* relationships, leading to the inclusion of irrelevant code elements in expanded code context models. Future work could focus on integrating more sophisticated program analysis techniques, such as enhanced data-flow analysis and hybrid static-dynamic techniques, to capture a broader range of relationships and improve precision in code context modeling. Second, design choices in our approach were evaluated using two node embedding models and four GNN models, but there may be more suitable strategies that could further improve performance. Future research could explore alternative node embedding techniques and experiment with advanced GNN architectures.

The main threat to ***external validity*** lies in the generalizability of our experiment results. We analyzed 5,256 code context models from three distinct Java open-source projects, using a combined dataset for experimental evaluation. While the dataset is specific to Java code, this reflects a difference at the dataset level, and the experimental results might vary when applied to other programming languages (e.g., C/C++). However, our approach is designed to be easily adaptable to different programming languages, due to its reliance on generalizable code representation techniques. Due to the limited availability of interaction history data for software projects, the evaluation was conducted using only three Eclipse open-source projects. These three projects, however, have been extensively employed in prior studies on code edit recommendation (e.g., [4], [39]), thereby providing a solid foundation for the generalizability of our approach. Although the experimental results of our approach may vary across different software projects, our approach can be applied to any project once its interaction history becomes available.

7 RELATED WORK

Some researchers conducted empirical studies to investigate code contexts of software development tasks [1], [58], [59]. The findings include developers form code contexts in their minds when performing software development tasks [1], they tend to frequently switch code contexts between various activities [58], and they perceive context switch leads to a loss of productivity [58]. Moreover, Souti et al. [59] identified six patterns of how developers organize and manage their code contexts during development tasks.

Previous studies have proposed various tools to help developers explicitly capture code contexts after the relevant code elements have been identified or navigated during their work [7], [8], [9]. Concern Graphs allow developers to manually capture relevant code elements and their relationships within the code context [7]. Code Bubbles introduces an innovative IDE interface that enables developers to create views of code fragments related to the tasks being performed [8]. Code Basket provides a canvas where developers can organize code elements, thereby externalizing their mental models [9]. Unlike such tools that emphasize the explicit capture of code contexts, our paper seeks to facilitate the effective prediction of code context models, with the potential for the proactive formation of code context models.

Other previous studies focus on automating the formation of code context models, e.g., by suggesting relevant code elements in code context models. Some studies use the structural information of source code to make code context suggestions, such as Suade [60]. Other studies leverage development task history data to recommend relevant code elements [4], [5], [10]. In a recent study, Wan et al. [11] combined structural and interaction history data, leveraging the topological patterns of code elements for code context prediction. Our approach shares similarities with that of Wan et al. in combining structural and interaction history data. However, in contrast to their approach, we leverage code representation models and GNNs to capture syntactic, semantic, and contextual information in code context models, enabling more effective prediction.

8 CONCLUSION AND FUTURE WORK

This paper presented a novel approach for predicting code contexts using GNNs, named GNNCONTEXT, which leverages code representation learning models to capture syntactic and semantic features of code elements, alongside GNNs to capture structural and contextual interdependencies among code elements in code context models. We compile a dataset consisting of 3,879 code context models derived from historical data of three Eclipse open-source projects to evaluate the effectiveness of our approach. The

evaluation results demonstrate that GNNCONTEXT significantly outperforms the state-of-the-art approach, achieving substantial improvements in mean reciprocal rank and top recall rates. Future work could explore more sophisticated embedding techniques and incorporate additional information, such as code dependencies, execution flow, and the evolution of code contexts throughout software development, to improve the performance of code context prediction.

ACKNOWLEDGEMENT

This research was supported by the National Science Foundation of China (No. 62472383 and No. 62102358), and the Open Research Fund of the State Key Laboratory of Blockchain and Data Security, Zhejiang University.

REFERENCES

[1] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich, "Developers' code context models for change tasks," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 7–18. [Online]. Available: https://doi.org/10.1145/2635868.2635905

[2] G. C. Murphy, "Beyond integrated development environments: Adding context to software development," in *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '19. IEEE Press, 2019, p. 73–76. [Online]. Available: https://doi.org/10.1109/ICSE-NIER.2019.00027

[3] C. Mcmillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: Searching for relevant functions and their usages in millions of lines of code," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, oct 2013. [Online]. Available: https://doi.org/10.1145/2522920.2522930

[4] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: Association for Computing Machinery, 2006, p. 1–11. [Online]. Available: https://doi.org/10.1145/1181775.1181777

[5] R. Robbes and M. Lanza, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212. [Online]. Available: https://doi.org/10.1007/s10515-010-0064-x

[6] D. Čubranić and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. USA: IEEE Computer Society, 2003, p. 408–418.

[7] M. P. Robillard and G. C. Murphy, "Concern graphs: Finding and describing concerns using structural program dependencies," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 406–416. [Online]. Available: https://doi.org/10.1145/581339.581390

[8] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, "Code bubbles: A working set-based interface for code understanding and maintenance," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 2503–2512. [Online]. Available: https://doi.org/10.1145/1753326.1753706

[9] B. Biegel, S. Baltes, I. Scarpellini, and S. Diehl, "Codebasket: Making developers' mental model visible and explorable," in *Proceedings of the Second International Workshop on Context for Software Development*, ser. CSD '15. IEEE Press, 2015, p. 20–24.

[10] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[11] Z. Wan, G. C. Murphy, and X. Xia, "Predicting code context models for software development tasks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020, p. 809–820. [Online]. Available: https://doi.org/10.1145/3324884.3416544

[12] Y. Wang, Y. Lin, Z. Wan, and X. Yang, "Task context: A tool for predicting code context models for software development tasks," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2023, pp. 156–160.

[13] L. Moreno and A. Marcus, "Jstereocode: Automatically identifying method and class stereotypes in java code," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: Association for Computing Machinery, 2012, p. 358–361. [Online]. Available: https://doi.org/10.1145/2351676.2351747

[14] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[15] Y. Du and Z. Yu, "Pre-training code representation with semantic flow graph for effective bug localization," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 579–591.

[16] C. Ling, Y. Zou, and B. Xie, "Graph neural network based collaborative filtering for api usage recommendation," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 36–47.

[17] (2024) How cognitive load affects software development efficiency. Accessed: 2025-03-13. [Online]. Available: https://www.anytopic.io/podcast/43ce2778-9c9b-490f-9c43-adc8e131ea39

[18] D. Helgesson, E. Engström, P. Runeson, and E. Bjarnason, "Cognitive load drivers in large scale software development," in *2019 IEEE/ACM 12th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 2019, pp. 91–94.

[19] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations*, 2017. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl

[20] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. vannbsp;den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2018, p. 593–607. [Online]. Available: https://doi.org/10.1007/978-3-319-93417-4_38

[21] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 1025–1035.

[22] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," *International Conference on Learning Representations*, 2018, accepted as poster. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[23] H. N. Phan, H. N. Phan, T. N. Nguyen, and N. D. Bui, "Repohyper: Better context retrieval is all you need for repository-level code completion," *arXiv preprint arXiv:2403.06095*, 2024.

[24] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, 2020.

[25] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th international conference on program comprehension*, 2020, pp. 184–195.

[26] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, X. Wu, C. Tao, T. Zhang, and W. Liu, "Learning to detect memory-related vulnerabilities," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–35, 2023.

[27] D. Hin, A. Kan, H. Chen, and M. A. Babar, "Linevd: statement-level vulnerability detection using graph neural networks," in *Proceedings of the 19th international conference on mining software repositories*, 2022, pp. 596–607.

[28] L. Šikić, A. S. Kurdija, K. Vladimir, and M. Šilić, "Graph neural network for source code defect prediction," *IEEE access*, vol. 10, pp. 10 402–10 415, 2022.

[29] S. Cao, X. Sun, X. Wu, D. Lo, L. Bo, B. Li, and W. Liu, "Coca: Improving and explaining graph neural network-based vulnerability detection systems," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[30] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/49265d2447bc3bbfe9e76306ce40a31f-Paper.pdf

[31] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 664–676. [Online]. Available: https://doi.org/10.1145/3468264.3468580

[32] S. Liu, Y. Chen, X. Xie, J. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid gnn," *arXiv preprint arXiv:2006.05405*, 2020.

[33] C. Ling, Y. Zou, and B. Xie, "Graph neural network based collaborative filtering for api usage recommendation," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 36–47.

[34] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," 2018.

[35] A. Mazuera-Rozo, A. Mojica-Hanke, M. Linares-Vásquez, and G. Bavota, "Shallow or deep? an empirical study on detecting vulnerabilities using deep learning," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 276–287.

[36] J. K. Siow, S. Liu, X. Xie, G. Meng, and Y. Liu, "Learning program semantics with code representations: An empirical study," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 554–565.

[37] K. Luo, Z. Liu, S. Xiao, and K. Liu, "Bge landmark embedding: A chunking-free embedding method for retrieval augmented long-context large language models," *arXiv preprint arXiv:2402.11573*, 2024.

[38] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *2006 IEEE computer society conference on computer vision and pattern recognition (CVPR'06)*, vol. 2. IEEE, 2006, pp. 1735–1742.

[39] S. Lee, S. Kang, S. Kim, and M. Staats, "The impact of view histories on edit recommendations," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 314–330, 2015.

[40] ——, "The impact of view histories on edit recommendations," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 314–330, 2014.

[41] D. V. Heesch, "Doxygen: Source code documentation generator tool," 2008. [Online]. Available: http://www.doxygen.org

[42] (2017) definition of "severity" levels in bugzilla. Accessed: 2025-03-13. [Online]. Available: https://bugs.gentoo.org/619108

[43] Y. Zhao, L. Li, H. Wang, H. Cai, T. F. Bissyandé, J. Klein, and J. Grundy, "On the impact of sample duplication in machine-learning-based android malware detection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–38, 2021.

[44] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN international symposium on new ideas, new paradigms, and reflections on programming and software*, 2019, pp. 143–153.

[45] C. Wheelan, *Naked statistics: Stripping the dread from the data*. WW Norton & Company, 2013.

[46] E. Shi, Y. Wang, W. Gu, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "Cocosoda: Effective contrastive learning for code search," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2198–2210.

[47] X. Ling, L. Wu, S. Wang, G. Pan, T. Ma, F. Xu, A. X. Liu, C. Wu, and S. Ji, "Deep graph matching and searching for semantic code retrieval," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 15, no. 5, pp. 1–21, 2021.

[48] W. Li, H. Qin, S. Yan, B. Shen, and Y. Chen, "Learning code-query interaction for enhancing code searches," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 115–126.

[49] S. Abid, S. Shamail, H. A. Basit, and S. Nadi, "Facer: An api usage-based code-example recommender for opportunistic reuse," *Empirical Software Engineering*, vol. 26, no. 6, p. 110, 2021.

[50] P. T. Nguyen, J. Di Rocco, C. Di Sipio, D. Di Ruscio, and M. Di Penta, "Recommending api function calls and code snippets to support software development," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2417–2438, 2021.

[51] P. T. Nguyen, C. Di Sipio, J. Di Rocco, D. Di Ruscio, and M. Di Penta, "Fitting missing api puzzles with machine translation techniques," *Expert Systems with Applications*, vol. 216, p. 119477, 2023.

[52] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[53] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 12 2014.

[54] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong (YIMING), L. Shou (), B. Qin, T. Liu, D. Jiang (), and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of EMNLP 2020*, September 2020. [Online]. Available: https://www.microsoft.com/en-us/research/publication/codebert-a-pre-trained-model-for-programming-and-natural-languages/

[55] J. Sutherland and J. Sutherland, *Scrum: the art of doing twice the work in half the time*. Crown Currency, 2014.

[56] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer, "Knn model-based approach in classification," in *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003. Proceedings*. Springer, 2003, pp. 986–996.

[57] K. R. Shahapure and C. Nicholas, "Cluster quality analysis using silhouette score," in *2020 IEEE 7th international conference on data science and advanced analytics (DSAA)*. IEEE, 2020, pp. 747–748.

[58] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz, "The work life of developers: Activities, switches and perceived productivity," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1178–1193, 2017.

[59] S. Chattopadhyay, N. Nelson, Y. R. Gonzalez, A. A. Leon, R. Pandita, and A. Sarma, "Latent patterns in activities: A field study of how developers manage context," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 373–383. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00051

[60] M. P. Robillard, "Topology analysis of software dependencies," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 4, aug 2008. [Online]. Available: https://doi.org/10.1145/13487689.13487691

[61] E. Murphy-Hill, R. Jiresal, and G. C. Murphy, "Improving software developers' fluency by recommending development environment commands," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2393596.2393645

APPENDIX A
CODE CONTEXT MODEL FORMATION

Fig. 6 provides an overview of the process used to curate the dataset.

A.1 Data Extraction

The top part of Fig. 6 describes the extraction of data from *interaction histories* of each project we considered. The Eclipse Mylyn tool[9] records *interaction histories* as a developer works on the code base of the project. Each interaction history includes a sequence of *interaction events* with code elements that are viewed and edited by the developer. Mylyn captures multiple attributes of an interaction event[10], among which we consider four attributes as shown in Table 10. Mylyn enables one or more interaction histories to be associated with each task performed by developers on a project. The interaction histories are stored with the tasks recorded in the Eclipse Bugzilla system.

TABLE 10: Information captured in an interaction event.

Attribute	Description
StartDate	Time stamp for the occurrence of the event.
Kind	Determines the type of interaction that took place, e.g., *edit*, *selection*, and *command* events that initiate from the user.
StructureHandle	A unique identifier for the element being interacted with.
StructureKind	The content type of the element being interacted with, e.g., Java code elements and XML files.

Filtering Bug Reports. To gather interaction histories, we first collected 8,180, 47,918, and 8,372 *FIXED* bug reports of the Mylyn, Platform, and PDE projects, respectively, from the Eclipse Bugzilla bug tracker up to September 2023. We further filtered the bug reports that did not have one or more interaction histories associated with the reports, leaving 4,723 bug reports to consider. The bug reports have an average of 1.4 interaction histories attached (Min: 1, Max: 42, Median: 1, SD: 1.1).

Extracting Interaction Histories. We extracted the final interaction history associated with each of these bug reports, which is an archived XML file.

A.2 Code Context Model Formation

The bottom part of Fig. 6 describes the formation of code context models from the collected data.

Breaking Interaction Histories. For code context models, we are interested in representing the models that developers usually keep in their minds as they work with code for a task. As a result, we need to break interaction histories into units that more likely represent a period of time in which a developer is working with the code and for which they may have formed a working mental code context model. To capture such units, we use the concept of a *working period* [11], consisting of the portion of the interaction history within a continuous period. Specifically, for each interaction history, we broke every two consecutive interaction events into two working periods if the time gap between their occurrence is over t, where t is determined by a sensitivity analysis. Our sensitivity analysis examines a range of t, from 1 hour (as observed by a previous study [61]) to 5 hours, with intervals of 0.5 hours. We evaluate the number of working periods derived from the interaction histories for different values of t. The sensitivity analysis reveals that the number of working periods decreases as t increases and stabilizes once t exceeds 2.5 hours. Consequently, we selected 3 hours as the value for t to break interaction histories into working periods.

Extracting Code Elements. We only considered interaction histories with events directly recording interaction with code elements ("selection" and "edit" events about class, method, and field code elements), as opposed to documentation or XML files. As a result, we have 6,126 working periods left. In addition, we removed 258 outlier working periods with their number of code elements lying outside the interval $[Q1 - 3IQR,\ Q3 + 3IQR]$[11]. For example, for the Mylyn project, $Q1 = 5$, $Q3 = 30$, and $IQR = 25$.

Structural dependencies between code elements are not available in interaction histories. To capture structural information, we need to be able to relate each interaction history to the version(s) of code active when the interaction history was collected. Thus, for each working period, we 1) resolved the git repository for extracted code elements, 2) extracted event timestamps from the interaction history, and 3) associated each working period with code snapshot(s).

Resolving Git Repositories. We resolved git repositories of accessed code elements for each working period as the project code is stored across several git repositories. This step excluded 1,724 working periods that access only coarse-grained code elements (*directory* or *file*), which lack structural relations, or involve code elements from unavailable code repositories (e.g., dependency libraries), or access only code elements that were not committed to the repository when the interaction history was collected. For the example working period in Fig. 6, we resolved two related git repositories, `mylyn.tasks` and `mylyn.commons`.

Extracting Event Timestamps. We extracted the *StartDate* attribute of each interaction event from an interaction history as the timestamp of the interaction event, and identified the timestamp of the first event, which can help locate the commit(s) of related repositories before each working period. In terms of the example working period in Fig. 6, the timestamp of the first event is `2010-02-21 11:35:53`.

Associating with Commit(s). By using the timestamp of the first event in an interaction history, we associated the interaction history with one or multiple commits in the related git repositories. Specifically, we retrieved the most recent commit in the git repository prior to the timestamp of the first event in the interaction history. In terms of the example working period in Fig. 6, we associated it with two commits, the commit `78457484` in `mylyn.tasks` and the commit `3ead864c` in `mylyn.commons`.

9. https://www.eclipse.org/mylyn
10. https://wiki.eclipse.org/Mylyn/Integrator_Reference
11. Q1 is the 25th quartile; Q3 is the 75th quartile; IQR (Interquartile Range) is defined as the difference between the 25th and 75th quartile and served as a measure of statistical dispersion.

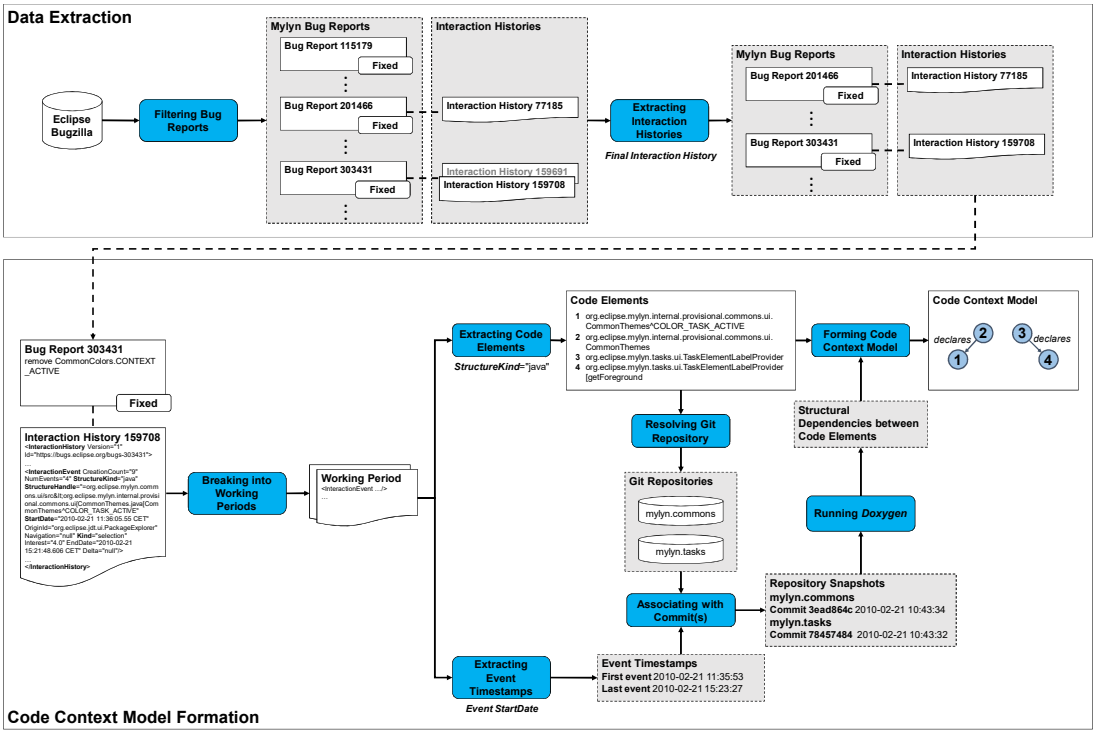


Fig. 6: Process of dataset curation.

Running Doxygen. We used Doxygen [41] to identify structural relations between code elements. Specifically, we run Doxygen for each code snapshot of each commit associated with the working period. In this paper, we consider four structural relations: *declares*, *calls*, *inherits*, and *implements*. Figure 6 illustrates that using Doxygen we identify two *declares* relations between the code elements.

Forming Code Context Models. We formed a code context model for each working period. The extracted code elements form the nodes of the code context model for a working period, while the identified structural dependencies form the edges of the code context model. Fig. 6 presents the code context model for the example working period, with four nodes and two directional edges labeled by structural relations. This code context model, which consists of two connected components, is the only working period associated with the Bug Report 303431 in the Mylyn project.