Empirical Software Engineering (2026) 31:50
https://doi.org/10.1007/s10664-025-10763-6

Learning to represent code changes

Xunzhu Tang[1] · Haoye Tian[2] · Weiguo Pian[1] · Saad Ezzini[3] · Abdoul Kader Kaboré[1] · Andrew Habib[1] · Kisub Kim[4] · Jacques Klein[1] · Tegawendé F. Bissyandé[1]

Received: 2 January 2025 / Accepted: 3 November 2025

Abstract

Code change representation plays a pivotal role in automating numerous software engineering tasks, such as classifying code change correctness or generating natural language summaries of code changes. Recent studies have leveraged deep learning to derive effective code change representation, primarily focusing on capturing changes in token sequences or Abstract Syntax Trees (ASTs). However, these current state-of-the-art representations do not explicitly calculate the intention semantic induced by the change on the AST, nor do they effectively explore the surrounding contextual information of the modified lines. To address this, we propose a new code change representation methodology, Patcherizer, which we refer to as our tool. This innovative approach explores the intention features of the context and structure, combining the context around the code change along with two novel representations. These new representations capture the sequence intention inside the code changes in the code change and the graph intention inside the structural changes of AST graphs before and after the code change. This comprehensive representation allows us to better capture the intentions underlying a code change. Patcherizer builds on graph convolutional neural networks for the structural input representation of the intention graph and on transformers for the intention sequence representation. We assess the generalizability of Patcherizer 's learned embeddings on three tasks: (1) Generating code change description in NL, (2) Predicting code change correctness in program repair, and (3) Code change intention detection. Experimental results show that the learned code change representation is effective for all three tasks and achieves superior performance to the state-of-the-art (SOTA) approaches. For instance, on the popular task of code change description generation (a.k.a. commit message generation), Patcherizer achieves an average improvement of 19.39%, 8.71%, and 34.03% in terms of BLEU, ROUGE-L, and METEOR metrics, respectively.

Keywords Code change representation · Code change correctness · Message generation

Communicated by: Lei Ma.

Extended author information available on the last page of the article

Published online: 17 December 2025

1 Introduction

A software code change represents the source code differences between two software versions. It has a dual role: on the one hand, it serves as a formal summary of the code changes that a developer intends to make on a code base; on the other hand, it is used as the main input specification for automating software evolution. Code changes are thus a key artifact that is pervasive across the software development life cycle. In recent years, building on empirical findings on the repetitiveness of code changes (Barr et al. 2014), several approaches have built machine learning models based on code change datasets to automate various software engineering tasks such as code change description generation (Linares-Vásquez et al. 2015; Buse and Weimer 2010; Cortés-Coy et al. 2014; Jiang et al. 2017; Xu et al. 2019; Liu et al. 2019, 2018, 2020c), code completion (Svyatkovskiy et al. 2019; Liu et al. 2020b, a; Ciniselli et al. 2021; Pian et al. 2022), code change correctness assessment (Tian et al. 2022c), and just-in-time defect prediction (Hoang et al. 2019; Kamei et al. 2016; Liu et al. 2023).

Early approaches relied on manually crafted features to represent code changes (Kamei et al. 2016, 2012). With the rise of deep learning, researchers began adopting representation learning techniques originally successful in text, signal, and image domains (Devlin et al. 2018; Niemeyer and Geiger 2021; Qin et al. 2021; Li et al. 2021; Tang et al. 2021; Pian et al. 2023; Wang et al. 2022a), applying them to software engineering tasks by developing neural models for code and code changes (Yin et al. 2019; Hoang et al. 2020; Feng et al. 2020; Nie et al. 2021; Jiang et al. 2021; Tian et al. 2022c; Pian et al. 2022; Liu et al. 2023). Among the most recent advances, CCRep (Liu et al. 2023) leverages pre-trained code models, contextual embeddings, and a “query back” mechanism to extract and encode changed fragments, achieving strong results in JIT defect prediction. However, CCRep does not explicitly capture the *intention* behind code edits. Two changes with nearly identical structural modifications may reflect fundamentally different purposes (e.g., disabling a feature versus refactoring a method). We argue that modeling such semantic intentions is crucial for advancing downstream applications beyond defect prediction, including commit refinement and correctness assessment.

Initially, these approaches treated code (Feng et al. 2020; Elnaggar et al. 2021; Wang et al. 2021d; Guo et al. 2021, 2021) and other code-like artifacts, such as code changes (Xu et al. 2019; Nie et al. 2021; Dong et al. 2022; Liu et al. 2020c), as a sequence of tokens and thus employ natural language processing methods to extract code in text format. Researchers have recognized the limitations of using code token sequences alone (often represented by + and− lines in the textual diff format) to capture the full semantics of code changes, as these symbols lack inherent meaning that a DL model can learn. To address this, they began incorporating the code structure, such as Abstract Syntax Trees (ASTs), to better capture the underlying structural information in source code (Zhang et al. 2019a; Alon et al. 2019, 2020; Guo et al. 2021). Therefore, recent work such as commit2vec (Cabrera Lozoya et al. 2021), C^3 (Brody et al. 2020), and CC2Vec (Hoang et al. 2020) attempted to represent code changes more structurally by leveraging ASTs as well. To get the best of both worlds, more recent work tried to combine token information with structure information to obtain a better code change representation (Dong et al. 2022). Finally, several such approaches of code change representation learning have been evaluated on specific tasks, e.g., BATS (Tian et al. 2022a) for code change correctness assessment and FIRA (Dong et al. 2022) for code change description generation.

On the one hand, token-based approaches for code change representation (Hoang et al. 2020; Xu et al. 2019; Nie et al. 2021) lack the rich structural information of source code and intention features inside the sequence is still unexplored. On the other hand, graph-based representation of code changes (Liu et al. 2020c; Lin et al. 2022) lacks the context which is better represented by the sequence of tokens (Hoang et al. 2020; Xu et al. 2019; Nie et al. 2021) of the code change itself and also the surrounding unchanged code and intention features inside graph changes is also still unexplored. In conclusion, approaches that try to combine context and AST information to represent code changes (e.g., FIRA Dong et al. 2022) do not use the intention features of either sequence or graph from the code change but rather rely on representing the code before and after the change while adding some *ad-hoc* annotations to highlight the changes for the model.

This Paper We propose a novel code change representation that tackles the aforementioned problems and provides an extensive evaluation of our approach on three practical and widely used downstream software engineering tasks. Our approach, Patcherizer, learns to represent code changes through a combination of (1) the context around the code change, (2) a novel SeqIntention representation of the sequential code change, and (3) a novel representation of the GraphIntention from the code change. Our approach enables us to leverage powerful DL models for the sequence intention such as Transformers and similarly powerful graph-based models such as GCN for the graph intention. Additionally, our model is pre-trained and hence task agnostic where it can be fine-tuned for many downstream tasks. We provide an extensive evaluation of our model on three popular code change representation tasks: (1) Generating code change description in NL, (2) Predicting code change correctness in program repair, and (3) Code change intention detection.

Overall, this paper makes the following contributions:

- ► *A novel representation learning approach for code changes*: we combine the context surrounding the code change with a novel sequence intention encoder and a new graph intention encoder to represent the intention of code changes in the code change while enabling the underlying neural models to focus on the code change by representing it explicitly. To that end, we developed: ❶*an adapted Transformer architecture for code sequence intention* to capture sequence intention in code changes taking into account not only the changed lines (added and removed) but also the full context (i.e., the code chunk before the code change application); ❷ *an embedding approach for graph intention* to compute embeddings of graph intention capturing the semantics of code changes.
- ► *A dataset of parsable code changes*: given that existing datasets only provide code changes with incomplete details for readily collecting the code before and after the code change, extracting AST diffs was challenging. We therefore developed tool support to enable such collection and produced a dataset of 90k code changes, which can be parsed using the Java compiler.
- ► *Extensive evaluation*: we evaluate our approach by assessing its performance on several downstream tasks. For each task, we show how Patcherizer outperforms carefully-selected baselines. We further show that Patcherizer outperforms the state of the art in code change representation learning.

2 Intention

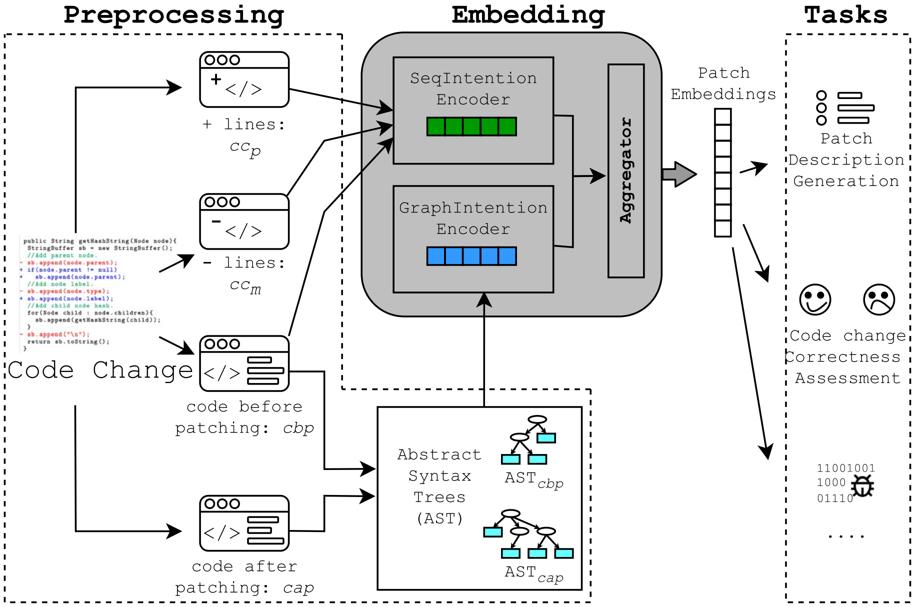
What is Intention? We define the *intention* of a code change as the semantic motivation behind the edit, e.g., fixing a bug, refactoring logic, disabling a feature, or improving readability. Unlike syntactic diffs that only describe the + / − lines or AST node operations, intention reflects the underlying purpose of the modification. For instance, consider two edits: (1) removing a method call to disable a feature, and (2) replacing a method call with another to refactor logic. Although both edits appear structurally similar in AST diffs (i.e., deleting or modifying a call node), their intentions differ fundamentally. Existing approaches such as CCRep (Liu et al. 2023) and FIRA (Dong et al. 2022) do not explicitly encode such distinctions.

How do we Capture Intention? Our design incorporates two complementary encoders: (1) the *SeqIntentionEncoder* models the sequence-level edit operations, integrating surrounding context and semantic patterns in the changed tokens, enabling it to distinguish, for example, whether a removed method indicates deactivation or replacement; (2) the *GraphIntentionEncoder* focuses on the structural edit patterns between ASTs, learning embeddings of the *changes* (rather than entire ASTs), which highlights semantically meaningful transformations. By combining both, our model is able to represent not only how code was changed but also why it was changed.

Motivation for Code Change Intention Detection In addition to description generation and correctness prediction, we introduce and explore a novel downstream task: *code change intention detection*. Unlike traditional diff tools that identify syntactic operations like additions or deletions, our goal is to uncover the semantic *intent* behind a change. This capability is crucial for automating commit message refinement, supporting intelligent software analytics, and enabling advanced tooling for software reviewers. For example, being able to detect whether a patch intends to disable a feature versus merely refactor a method allows tools to generate more meaningful messages or guide human reviewers' attention to impactful modifications. We argue that intention detection should go beyond surface diffs and rely on learned semantics from both structure and context—a gap that Patcherizer fills effectively.

3 Patcherizer

Figure 1 presents the overview of Patcherizer. Code changes are first preprocessed to split the available information about added (+) and removed (−) lines, identifying the code context (i.e., the code chunk before applying the code change) and computing the ASTs of the code before and after applying the code changes (cf. Section 3.2). Then, Patcherizer deploys two encoders, which capture sequence intention semantics (cf. Section 3.3) and graph intention semantics (cf. Section 3.4). Those encoded information are aggregated (cf. Section 3.6) to produce code change embeddings that can be applied to various downstream tasks. In the rest of this section, we will detail the different components of Patcherizer before discussing the pre-training phase (cf. Section 3.7).



```
public String getHashString(Node node){
  StringBuffer sb = new StringBuffer();
  //Add parent node.
- sb.append(node.parent);
+ if(node.parent != null)
+   sb.append(node.parent);
  //Add node label.
- sb.append(node.type);
+ sb.append(node.label);
  //Add child node hash.
  for(Node child : node.children){
    sb.append(getHashString(child));
  }
- sb.append("\n");
  return sb.toString();
}
```

Fig. 1 Overview of Patcherizer

3.1 Handling Incomplete Code Snippets

In real-world software repositories, patches often appear as partial code fragments extracted from diffs, lacking full method or class definitions required for reliable parsing and structural analysis. To ensure that our model can robustly represent such incomplete code, **Patcherizer** employs a *context construction mechanism* that reconstructs semantically valid and parsable code by leveraging surrounding, unmodified code as context.

During the preprocessing phase, we identify the surrounding code chunk prior to patch application (referred to as `cbp`) and combine it with the lines to be added (`ccp`) and removed (`ccm`) to recover a coherent code segment. This reconstruction not only aids in forming a complete representation for the sequential input but also enables successful parsing of the Abstract Syntax Tree (AST) required for graph-based representation.

Specifically, when certain structural elements are missing in the diff (e.g., a method body without its signature or class declaration), our system uses a sliding window strategy to extract surrounding lines from the same file to syntactically complete the snippet. We then apply the `javalang` parser to the reconstructed code segment to generate the corresponding ASTs before and after the change. If the full reconstruction still fails due to excessive incompleteness, we fall back to using only the available sequence-level representation without graph-based embedding.

This fallback mechanism ensures that **Patcherizer** can generalize to a wide range of real-world patch inputs—whether they are complete or incomplete—by balancing robustness with semantic fidelity. It is particularly valuable in scenarios such as just-in-time commit analysis, where full project context may not be readily available.

3.2 Code Change Preprocessing

The preprocessing aims to focus on three main information within a code change for learning its representation. The code before applying the code change (which provides contextual information of the code change), plus and minus lines (which provide information about the code change operations), and the difference between *AST* graphs before and after code change (which provides information about graph intention in the code). Through the following steps we collect the necessary multi-modal inputs (code text, sequence intention, and graph intention) for the learning:

1. **Collect $+/-$ lines in the code change.** We scan each code change line. Those starting with a + are added to a *pluslist*, while those starting with a$-$ are added to a *minuslist*. Both lists record the line numbers in the code change.
2. **Reconstruct before/after code.** Besides +/- lines, a code change includes unchanged code that are part of the context. We consider that the full context is the code before applying the code change (i.e., unchanged & minuslist lines). We also construct the code after applying the code change (i.e., unchanged & pluslist lines). The reconstruction leverages the recorded line numbers for inserting each added/removed line to the proper place and ensure accuracy.
3. **Generate code ASTs before and after code change.** We apply the Javalang (Thunes 2013) tool to generate the ASTs for the reconstructed code chunks before and after applying the code change.
4. **Construct vocabulary.** Based on the code changes of the code changes in the training data, we build a vocabulary using the Byte-Pair-Encoding (*BPE*) algorithm.

At the end of this preprocessing phase, for each given code change, we have a set of inputs: $\langle cc_p, cc_m, cbp, cap, G_{cbp}, G_{cap} \rangle$, where cc_p is the sequence of added (+) lines of code, cc_m is the sequence of removed (-) lines of code, *cbp* is the **c**ode chunk **b**efore the **p**atch is applied, *cap* is the **c**ode chunk **a**fter the **p**atch is applied, G_{cbp} is the AST graph of *cbp* and G_{cap} is the graph of *cap*.

3.3 Sequence Intention Encoder

Intention features refer to the semantic signals that capture *why* a change was made (e.g., to fix a bug, refactor code, disable functionality), rather than just *what* was changed. Unlike prior approaches that focus on syntax or structural patterns, our intention encoders aim to learn such semantics through contextualized token changes (in the sequence encoder) and structural shifts (in the graph encoder).

Despite the success of AST-based methods, they often struggle to distinguish code changes that are structurally similar but semantically distinct. Our intention encoders are designed to capture such nuances. Figure 2 provides an illustrative example: although both code changes involve editing an `if` condition, one disables functionality while the other refactors logic. Traditional AST diffs treat both similarly, but their purposes are fundamentally different. Patcherizer captures these differences through intention-aware encoding.

```
if (x) {
    doSomething();
}
```

Listing 1: Disabling a Feature

```
if (checkCondition()) {
    doSomething();
}
```

Listing 2: Refactoring Logic

Fig. 2 Example illustrating how semantically distinct code changes can result in similar AST diffs. Both code changes involve replacing the condition in an "if" statement, and their corresponding AST diffs may appear structurally similar. However, their *intentions* differ: the left change disables functionality, while the right one introduces a refactor with encapsulated logic. Traditional AST-based models may treat these equivalently, while Patcherizer captures this distinction through its sequence and graph intention encoders

A first objective of Patcherizer is to build an encoder that is capable of capturing the semantics of the sequence intention in a code change. Although prior work focuses mainly on +/− lines or simply does some calculation between changed codes and their contextual contents, we postulate that code context is a relevant additional input for better encoding such differences. Figure 3 depicts the architecture of the Sequence intention encoder. We leverage the relevant subset of the preprocessed inputs (cf. Section 3.2) to pass to a Transformer embedding layer and further develop a specialized layer, named the *SeqIntention embedding layer*, which captures the intention features from the sequence.

3.3.1 Input Layer

The input, for each code change, consists of the triplet $\langle cc_m, cc_p, cbp\rangle$, where cc_m is the set of removed (−) lines, cc_p the set of added (+) lines and *cbp* is the code before code changeing, which represents the context.

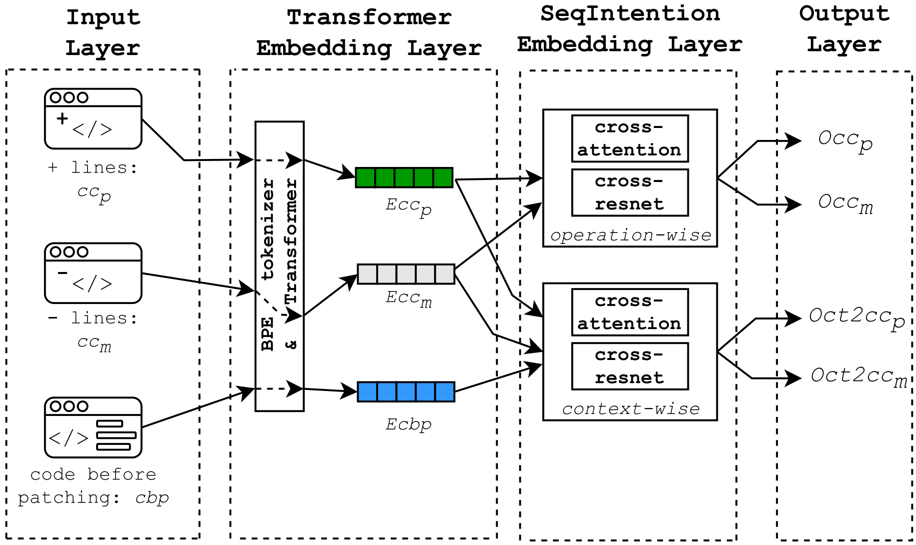


Fig. 3 Architecture for the Sequence Intention Encoder

3.3.2 Transformer Embedding Layer

To embed the sequence of code changes, we use a Transformer as the initial embedder. Indeed, Transformers have been designed to capture semantics in long texts and have been demonstrated to be effective for inference tasks (Devlin et al. 2018; Wang et al. 2022b).

We note that $cc_m \in$ *cbp*. Assuming that $cc_p = \{token_{p,1}, \ldots, token_{p,j}\}$, $cc_m = \{token_{m,1}, \ldots, token_{m,k}\}$, *cbp* = $\{token_{cbp,1}, \ldots, token_{cbp,l}\}$, where *j, k, l* represent the maximum length of cc_p, cc_m, and *cbp* respectively, we use the initial embedding layer in the Pytorch's nn.module implementation to produce first vector representations for each input information as:

$$E_X = Transformer(Init(X; \Theta_1); \Theta_2) \tag{1}$$

where *X* represents an input (either cc_m, cc_p or *cbp*); *Init* is the initial embedding function; *Transformer* is the model based on a transformer architecture; Θ_1 and Θ_2 are the parameters of *Init*( ) and *Transformer*( ), respectively.

The Transformer embedding layer outputs $E_{cc_p} = [e_{p,1}, e_{p,2}, \ldots, e_{p,j}] \in \mathbb{R}^{j \times d_e}$, $E_{cc_m} = [e_{m,1}, e_{m,2}, \ldots, e_{m,k}] \in \mathbb{R}^{k \times d_e}$, $E_{cbp} = [e_{cbp,1}, e_{cbp,2}, \ldots, e_{cbp,l}] \in \mathbb{R}^{l \times d_e}$, where d_e is the size of the embedding vector.

3.3.3 SeqIntention Embedding Layer

Once the Transformer embedding layer has produced the initial embeddings for the inputs cc_p, cc_m and *cbp*, our approach seeks to capture how they relate to each other. Prior works (Shaw et al. 2018; Devlin et al. 2018; Xu et al. 2019; Dong et al. 2022) have proven that self-attention is effective in capturing relationships among embeddings. We thus propose to capture relationships between the added and removed sequences, with the objective of capturing the intention of the code change through the change operations. We also propose to pay attention to context information when capturing the semantics of the sequence intention.

Operation-wise To obtain the intention of modifications in code changes, we apply a cross-attention mechanism between cc_p and cc_m. To that end, we design a resnet architecture where the model performs residual learning of the importance of inputs (i.e., E_{cc_p} and its evolved $\sqsubseteq_p$, which will be introduced below).

To enhance E_{cc_p} into E_{cc_m}, we apply a **cross-attention** mechanism. For the *i-th* token in cc_m, we compute the matrix-vector product, $E_{cc_p} e_{m,i}$, where $e_{m,i} \in \mathbb{R}^{d_e}$ is a vector parameter for *i-th* token $\rangle$ in cc_m. We then pass the resulting vector through a softmax operator, obtaining a distribution over locations in the E_{cc_p},

$$\alpha_\rangle = SoftMax(E_{cc_p} e_{m,i}) \in \mathbb{R}^k, \tag{2}$$

where SoftMax(***x***) = $\frac{exp(x)}{\Sigma_j exp(x_j)}$. *exp(x)* is the element-wise exponentiation of the vector *x*. *k* is the length of cc_m, The attention α is then used to compute vectors for each token in cc_m,

$$\sqsubseteq_\rangle = \Sigma_{n=1}^{j} \alpha_{\rangle,\backslash} h_n. \tag{3}$$

where $h_n \in E_{cc_p}, j$ is the length of cc_p. In addition, $\sqsubseteq_{\rangle}$ is the new embedding of *i-th* token in cc_m enhanced by semantic of E_{cc_p}.

Then, we get new cc_m embedding $v_m = [\sqsubseteq_1, \ldots, \sqsubseteq_k] \in \mathbb{R}^{k \times d_e}$.

Similarly, following steps above, we can obtain new embedding of cc_p, $v_p \in \mathbb{R}^{j \times d_e}$, enhanced by the semantic of cc_m.

For the combination of v_p, v_m, E_{cc_p}, E_{cc_m}, inspired by Shi et al. (2021); He et al. (2016), we design a **cross-resnet** for combining v_p, v_m, v_{cc_p}, and v_{cc_m}. The pipeline of **cross-resnet** is shown in Fig. 4. The process is as follows:

$$\begin{aligned} O_{cc_p} &= f(n(E_{cc_p}) + \mathcal{A}\lceil\lceil(E_{cc_p}, v_p)) \\ O_{cc_m} &= f(n(E_{cc_m}) + \mathcal{A}\lceil\lceil(E_{cc_m}, v_m)) \end{aligned} \tag{4}$$

where *n*( ) is a normalization function in Devlin et al. (2018); $\mathcal{A}\lceil\lceil(\cdot)$ is the adding function, *f*( ) represents *RELU* (Glorot et al. 2011) activation function.

Finally, we obtain output O_{cc_m} and O_{cc_p}.

Context-Wise Similar to operation-wise block, we enhance the contextual information into modified lines by **cross-attention** and **cross-resnet** blocks. The computation process is as follows:

$$\begin{aligned} O_{ct2cc_p} &= f(n(E_{cc_p}) + \mathcal{A}\lceil\lceil(E_{cc_p}, E_{cbp})) \\ O_{ct2cc_m} &= f(n(E_{cc_m}) + \mathcal{A}\lceil\lceil(E_{cc_m}, E_{cbp})) \end{aligned} \tag{5}$$

where E_{cbp} is the embedding of *cbp* calculated by (1); O_{ct2cc_p} represents the vector of context-enhanced plus embedding and O_{ct2cc_m} is the vector of context-enhanced minus embedding.

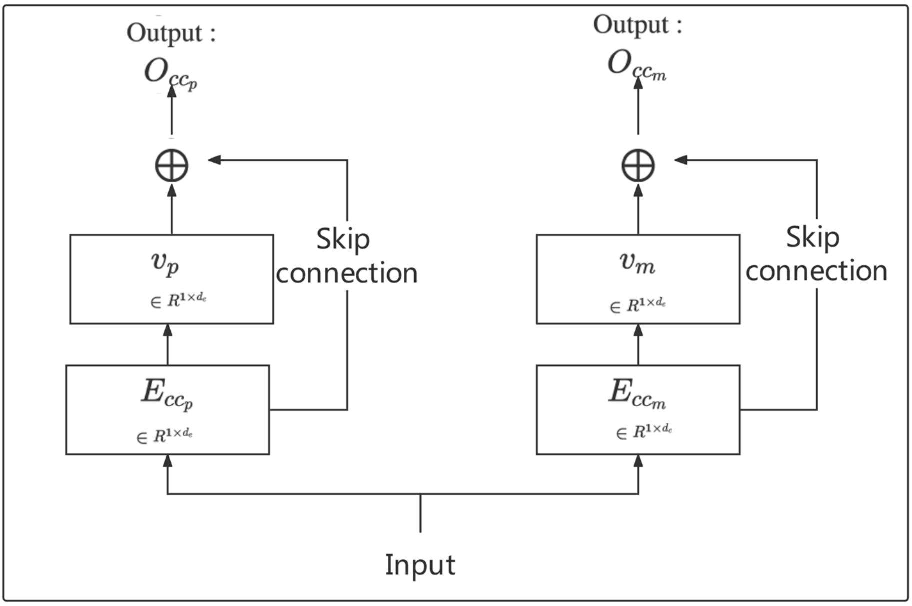


Fig. 4 cross-resnet architecture

3.4 Graph Intention Encoder

Concurrently to encoding sequence information from code changes, we propose to also capture the graph intention features of the structural changes in the code when the code change is applied. To that end, we rely on a Graph Convolutional Network (GCN) architecture, which is widely used to capture dynamics in social networks, and is effective for typical graph-related tasks such as classification or knowledge injection (Kipf and Welling 2016; Zhang et al. 2019b, c). Once the GCN encodes the graph nodes, the produced embeddings can be used to assess their relationship via computing their cosine similarity scores (Luo et al. 2018). Concretely, in Patcherizer, we use a GCN-based model to capture the graph intention features. The embedder model was trained by inputting a static graph, a graph resulting from the merge of all sub-graphs from the training set. Overall, we implement this encoding phase in two steps: building the static graph, performing graph learning and encoding the graph intention (cf. Figure 5).

3.4.1 Graph Building

To start, we consider the G_{cbp} and G_{cap} trees, which represent in graph forms.

❶ **Static Graph building:** Each code change in the dataset can be associated to two graphs: G_{cbp} and G_{cap}, which are obtained by parsing the *cbp* and *cap* code snippets. After collecting all graphs (which are unidirectional graphs) for the whole training set, we merge them into a “big” graph by iteratively linking the common nodes. In this big graph, each distinct code snippet AST-inferred graph is placed as a distinct sub-graph. Then, we will merge the graphs shown in Fig. 6 which illustrates the merging progress of two graphs: if a node $\mathcal{N}$ has the same value, position, and neighbors in both ASTs, it will be merged into one (e.g., red nodes 1 and 2). However, when a common node has different neighbors between the ASTs (e.g., red nodes 3 and 4), the merge keeps one instance of the common node but includes all neighbors connected to the merged red nodes (i.e., all green and grey nodes are now connected to red nodes 3 and 4, respectively). After iterating over all graphs, we eventually build the static graph.

However, on the one hand, some nodes in most subgraphs such as ‘prefix_operators’, ‘returnStatement’, and ‘StatementExpression’ are not related to the semantics of the code change. On the other hand, as data statistics in our study, 97.2% nodes in the initial graphs

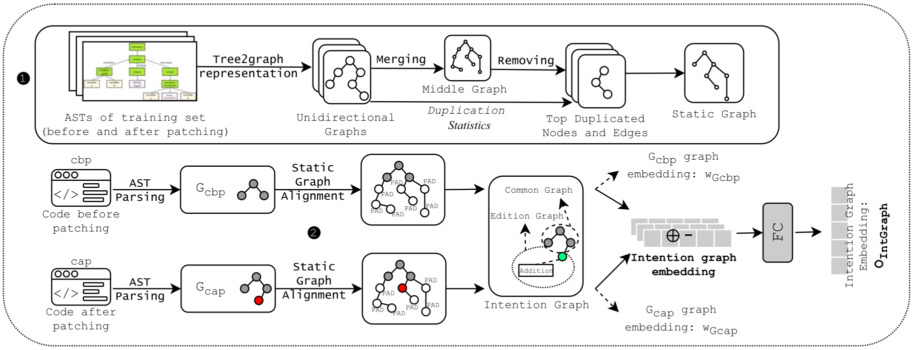

❶
Tree2graph representation
Merging
Middle Graph
Removing
Duplication Statistics
ASTs of training set (before and after patching)
Unidirectional Graphs
Top Duplicated Nodes and Edges
Static Graph
cbp
Code before patching
AST Parsing
G_{cbp}
Static Graph Alignment
PAD
❷
cap
Code after patching
G_{cap}
Common Graph
Edition Graph
Addition
Intention Graph
G_{cbp} graph embedding: w_{Gcbp}
G_{cap} graph embedding: w_{Gcap}
⊕ -
Intention graph embedding
FC
Intention Graph Embedding: $O_{IntGraph}$

Fig. 5 Architecture for the Graph Intention Encoder

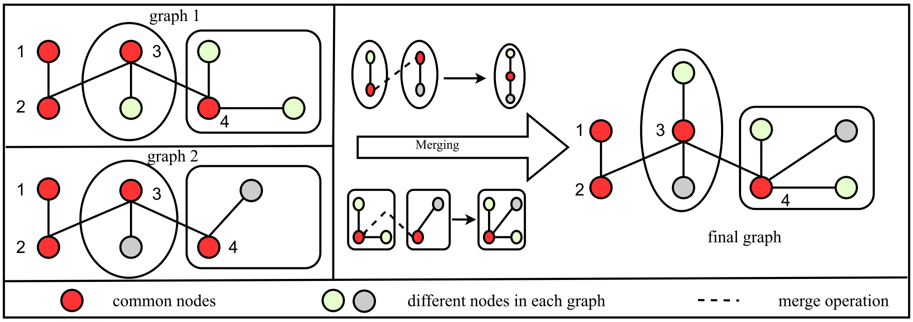


Fig. 6 An example of merging graphs

(ASTs) extracted from code are from parser tools instead of code changes, which means these nodes are rarely related with the semantic of the code change. Thus, as shown in step 1 in Fig. 5, we remove nodes whose children do not contain words in the code change to reduce the size of the graph because these nodes will be considered noise in our research.

In the remainder of this paper, we refer to the final graph as the *static graph* $\mathcal{G}$=$(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges.

❷ **Graph Alignment to the Static Graph:** GCN requires that the input graphs are all of the same size (Kipf and Welling 2016). Yet, the graphs built using the graphs of *cbp* and *cap* do not have as many nodes and edges as the static graph used for training the GCN network. Consequently, we propose to use the global static graph to initialize all specific ast-diff graphs for unique code change. Given the global static graph (huge graph mentioned before) $\mathcal{G}_{global} = (\mathcal{V}_g, \mathcal{E}_g)$ and an AST graph $\mathcal{G}_{local} = (\mathcal{V}_l, \mathcal{E}_l)$, we leverage the *VF* graph matching algorithm (Cordella et al. 1999) to find the most similar sub-graph with $\mathcal{G}_{local}$ in $\mathcal{G}_{global}$:

$$subGraph = VFG(\mathcal{G}_{local}, \mathcal{G}_{global}). \tag{6}$$

where $VFG(\cdot)$ is the function representing the VF matching algorithm (Networkx 2018). The matched sub-graph is a subset of both $\mathcal{G}_{local}$ and $\mathcal{G}_{global}$: some nodes of *subGraph* will be in $\mathcal{G}_{local}$ but not in $\mathcal{G}_{global}$. We then align $\mathcal{G}_{local}$ to the same size of $\mathcal{G}_{global}$ as follows: we use the [PAD] element to pad the node of subGraph to the same size of $\mathcal{G}_{global}$ and then we obtain $\mathcal{G}_{l_{PAD}}$. Therefore, $\mathcal{G}_{l_{PAD}}$ keeps the same size and structure of the static graph $\mathcal{G}_{global}$. Eventually, all graphs are aligned to the same size of $\mathcal{G}_{global}$ and the approach can meet the requirements for GCN computation for graph learning.

3.5 Graph Intention Encoding Details

The graph intention encoding process involves comparing and merging Abstract Syntax Trees (ASTs) from the original and modified code. Our approach uses a tree-based differencing algorithm inspired by GumTree (Falleri et al. 2014) but optimized for Java ASTs. The AST comparison and merging process follows three key phases:

1. **Node Mapping**: We identify mappings between nodes in the original and modified ASTs using a combination of structural similarity (based on node types and parent-child

relationships) and token-level similarity metrics. This hybrid approach achieves approximately 92% accuracy when evaluated against manually labeled AST diffs from our dataset.

2. **Change Detection**: Based on these mappings, we detect node operations (addition, deletion, update, move) by analyzing both mapped and unmapped nodes. For unmapped nodes in the modified AST, we classify them as additions; for unmapped nodes in the original AST, we classify them as deletions; for mapped nodes with different values, we classify them as updates.
3. **Graph Construction**: We construct a unified graph where matched nodes from both ASTs are merged into single nodes with special attributes indicating whether they were preserved, added, or deleted. This creates a comprehensive representation that explicitly encodes the transformation between the original and modified code.

The remaining 8% of cases where the AST differencing is imperfect typically involve complex refactorings with significant structural changes. To mitigate the impact of these inaccuracies on downstream tasks, our dual-encoding approach complements the graph intention encoding with sequence intention encoding, providing robustness when AST-based differencing is imperfect. As demonstrated in our ablation studies (RQ-2), when graph intention encoding contains inaccuracies, the sequence intention encoding can effectively compensate, maintaining strong performance across all downstream tasks.

The graph merging process preserves the original hierarchical relationships while adding special edges to represent the transformation operations. This allows our model to learn patterns of how code structures evolve rather than just focusing on token-level changes, contributing significantly to the improved performance across all downstream tasks as demonstrated in our experiments.

3.5.1 Graph Learning

Inspired by Zhang et al. (2019b), we build a deep graph convolutional network based on the undirected graph formed following the above construction steps to further encode the contextual dependencies in the graph. Specifically, for a given undirected graph $\mathcal{G}_{l_{PAD}} = (\mathcal{V}_{l_{PAD}}, \mathcal{E}_{l_{PAD}})$, let $\mathcal{P}$ be the renormalized graph laplacian matrix (Kipf and Welling 2016) of $\mathcal{G}_{l_{PAD}}$:

$$\begin{aligned} \mathcal{P} &= \hat{\mathcal{D}}^{-1/2}\hat{\mathcal{A}}\hat{\mathcal{D}}^{-1/2} \\ &= (\mathcal{D}+\mathcal{L})^{-1/2}(\mathcal{A}+\mathcal{L})(\mathcal{D}+\mathcal{L})^{-1/2} \end{aligned} \tag{7}$$

where $\mathcal{A}$ denotes the adjacency matrix, $\mathcal{D}$ denotes the diagonal degree matix of the graph $\mathcal{G}_{l_{PAD}}$, and $\mathcal{L}$ denotes the identity matrix. The iteration of GCN through its different layers is formulated as:

$$\mathcal{H}^{(l+1)} = \sigma(((1-\alpha)\mathcal{P}\mathcal{H}^{(l)} + \alpha\mathcal{H}^{(0)})((1-\beta^{(l)})\mathcal{L} + \beta^{(l)}\mathcal{W}^{(l)})) \tag{8}$$

where α and $\beta^{(l)}$ are two hyper parameters, σ denotes the activation function and $\mathcal{W}^{(\updownarrow)}$ is a learnable weight matrix. Following GCN learning, we use the average embedding of the graph to represent the semantic of structural information in code snippet:

$$w_G = \frac{1}{L}\Sigma_{i=1}^{L}(\mathcal{H}_\rangle). \tag{9}$$

Thus, at the end of the graph embedding, we obtain representations for G_{cbp} and G_{cap}, i.e., $w_{G_{cbp}}$ and $w_{G_{cap}} \in \mathrm{R}^{1\times d_e}$.

3.5.2 Graph Intention Encoding

Once we have computed the embeddings of the code snippets before and after code changeing, (i.e., the embeddings of *cbp* and *cap*), we must get the representation of their differences to encode the intention inside the graph changes. To that end, similarly to the previous cross-resnet for sequence intention, we design a **graph-cross-resnet** operator which ensembles the semantic of $w_{G_{cbp}}$ and $w_{G_{cap}}$. Figure 7 illustrates this crossing. In this graph-cross-resnet, the model can choose and highlight a path automatically by the backpropagation mechanism. The *GraphIntention* is therefore calculated as follows:

$$\begin{aligned} path1 &= w_{G_{cbp}} \\ path2 &= \mathcal{A}\lceil\lceil(w_{G_{cbp}}, w_{G_{cap}}) \\ path3 &= w_{G_{cap}} \\ O_{GraphIntention} &= \mathcal{FC}(f(\mathcal{A}\lceil\lceil(path1, path2, path3)), \Theta_3) \end{aligned} \tag{10}$$

where $\mathcal{FC}$ is a fully-connected layer and Θ_3 is the parameter of $\mathcal{FC}$.

At the end, the graph-cross-resnet component outputs the sought graph intention embedding: *GraphIntention*.

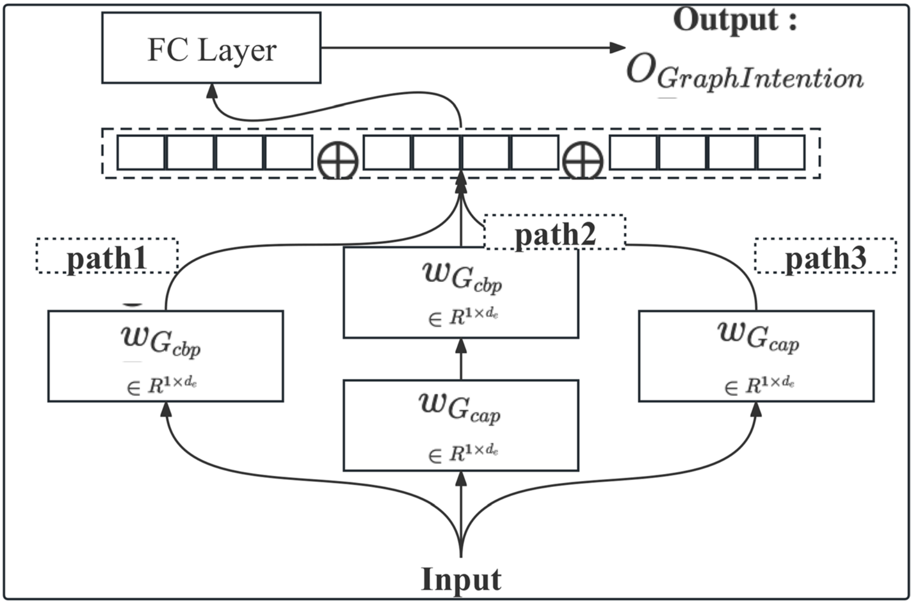

FC Layer
Output :
$O_{GraphIntention}$
path1
path2
path3
$w_{G_{cbp}} \in R^{1\times d_e}$
$w_{G_{cbp}} \in R^{1\times d_e}$
$w_{G_{cap}} \in R^{1\times d_e}$
$w_{G_{cap}} \in R^{1\times d_e}$
Input

Fig. 7 graph-cross-resnet architecture

3.6 Aggregator: Aggregating multi-source Input Embeddings

With the sequence intention encoder and the graph intention encoder, we can produce for each code change several embeddings of different input modalities (code sequences and graphs) that must be aggregated into a single representation.

Concretely, we use the $\mathcal{A}\lceil\lceil$ aggregation function to merge the *SeqIntention* embeddings (combination of O_{cc_p}, O_{cc_m}, O_{ct2cc_p} and O_{ct2cc_m} - cf. Equations 4 and 5) and *GraphIntention* embedding $O_{GraphIntention}$ before outputing the final representation $E_{Patcherizer}$. Actually, we use $E_{Patcherizer}$ as the representation of code change out of the model.

3.7 Pre-training

Patcherizer is an approach that is agnostic to downstream tasks. We propose to build a pre-trained model using a large corpus of code changes. The objective is to enable the model to learn contextual and structural semantics of code edits, thereby enhancing the quality and robustness of code change representations.

The pre-training task is formulated as masked token prediction. Following the popular bidirectional objective from masked language modeling (MLM; Devlin et al. 2018), we randomly mask a subset of tokens in the input sequence and train the model to predict these masked tokens using their surrounding context. Formally, this can be expressed as computing the conditional probability $P(x_i|x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$.

Inspired by previous pre-training works on code representation learning (Feng et al. 2020; Elnaggar et al. 2021; Zhang et al. 2019d), we adopt the MLM objective for the encoder. Unlike traditional MLM models that rely solely on BERT-style encoders, we employ an encoder-decoder architecture where the decoder is a left-to-right Transformer (Vaswani et al. 2017), similar to GPT-style models, which are more suitable for autoregressive generation tasks.

Specifically, we use our proposed Patcherizer encoder to produce latent embeddings of code changes. These embeddings are then passed to a standard Transformer-based decoder that shares the same vocabulary. The decoder autoregressively generates tokens, starting from an initial $<$s$>$ token, using the following formulation:

$$\text{index} = \arg\max\left(p(y_t|y_{t-1}, \dots, y_1, E_{Patcherizer})\right) \tag{11}$$

where y_t is the token to be predicted at position *t*, and $E_{Patcherizer}$ is the encoded representation from the Patcherizer encoder. The decoder outputs a distribution over the vocabulary, from which we select the token with the highest probability.

3.8 Fine-tuning for Different Tasks

Patcherizer serves as a task-agnostic encoder that generates semantically rich embeddings of code changes. After pre-training on a large corpus of code edits using the masked token prediction task (cf. Section 3.7), we fine-tune Patcherizer on several downstream tasks relevant to software maintenance. These tasks include code change description generation, code change correctness assessment, and code change intention detection.

3.8.1 Code Change Description Generation

This task involves generating a natural-language description (e.g., commit message) given a code change. The fine-tuning setup mirrors that of pre-training: we employ an encoder-decoder architecture, where the encoder is the pre-trained Patcherizer and the decoder is a Transformer-based autoregressive generator.

The dataset consists of paired samples of code changes and corresponding natural-language descriptions. During training, we input the code change to the encoder and generate the description token-by-token with the decoder. The learning objective is to maximize the likelihood of generating the correct sequence:

$$\text{index} = \arg\max\left(p(y_t|y_{t-1}, \cdots, y_1, E_{Patcherizer})\right) \tag{12}$$

where y_t is the target token at time *t* and $E_{Patcherizer}$ is the encoded representation from the Patcherizer encoder.

3.8.2 Code Change Correctness Assessment

This task aims to predict whether a given code change is a correct fix for a reported bug. It is formulated as a binary classification problem.

We use the Patcherizer encoder to generate an embedding for the code change and use a separate pre-trained BERT (Devlin et al. 2018) model to embed the associated bug report. The two embeddings are concatenated and passed to a fully connected classification layer:

$$\hat{y}_i = \text{sigmoid}(E_{patch_i} \oplus E_{bugReport_i}) \tag{13}$$

where E_{patch_i} is the embedding of the code change, $E_{bugReport_i}$ is the embedding of the bug report, and $\oplus$ denotes concatenation.

We train the classifier using binary cross-entropy loss:

$$\mathcal{L}_a = -\sum_{i=1}^{n}\left(y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)\right) \tag{14}$$

where y_i is the ground truth label and $\hat{y}_i$ is the predicted probability.

3.8.3 Code Change Intention Detection

This task focuses on identifying the primary semantic intention behind a given code change, classifying it into categories such as `add`, `remove`, or `update`. While traditional diff tools can highlight superficial syntactic operations, they fail to capture the deeper semantics or purpose of a change. For instance, a change may involve both an addition and removal, but its true intention may be to *disable a feature*, *refactor a loop*, or *resolve a bug*—information that is not directly evident from raw diffs.

Code change intention detection serves several practical use cases. It enables:

- **Automated commit message refinement**, where semantically meaningful summaries can be synthesized.

- **Fine-grained maintenance analytics**, aiding in the analysis of developer behavior and software evolution patterns.
- **Automated patch review assistance**, where intention-aware prioritization of patches can improve reviewer efficiency.

To perform this task, we train a classifier over Patcherizer-generated embeddings, which capture both context-aware sequence semantics and structural graph-based information. Because the embeddings encode deep semantics, no external input beyond the code change is needed for classification.

To visualize the separation of intentions in embedding space, we apply t-SNE (t-distributed Stochastic Neighbor Embedding), which projects high-dimensional embeddings into 2D space. The projection is optimized by minimizing the KL-divergence between the high- and low-dimensional pairwise similarities:

$$\mathrm{KL}(P \parallel Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{15}$$

with:

$$p_{ij} = \frac{\exp(-\|x_i - x_j\|^2/2\sigma^2)}{\sum_{k \neq l} \exp(-\|x_k - x_l\|^2/2\sigma^2)}, \quad q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l}(1 + \|y_k - y_l\|^2)^{-1}} \tag{16}$$

where x_i, x_j are high-dimensional embeddings, y_i, y_j their low-dimensional projections, and σ the perplexity.

The visualizations and downstream performance demonstrate that Patcherizer effectively distinguishes between different code change intentions. By uncovering the *purpose* behind edits, Patcherizer brings semantic reasoning to automated code analysis pipelines.

4 Experimental Design

We provide the implementation details (cf. Sec. 4.1), discuss the research questions (cf. Sec. 4.2), and present the baselines (cf. Sec. 4.3), the datasets (cf. Sec. 4.4), and the metrics (cf. Sec. 4.5).

4.1 Implementation

In the pre-training phase used for the Sequence Intention Embedding step, we apply a beam search (Vijayakumar et al. 2016) for the best performance in predicting the masked words. The beam size was set to 3. The dimension of the hidden layer output in models is set to 512, and the default value of dropout rate is set to 0.1. For the Transformer, we apply 6 heads for the multi-header attention module and 4 layers for the attention.

For the Graph Intention Embedding step, we use javalang (Thunes 2013) to parse code fragments and collect ASTs. We build on graph manipulation packages (i.e., networkx Networkx 2018, and dgl Wang et al. 2019) to represent these ASTs into graphs.

Patcherizer ’s training involves the Adam optimizer (Kingma and Ba 2014) with learning rate 0.001. All model parameters are initialized using Xavier algorithm (Glorot and Bengio 2010). All experiments are performed on a server with an Intel(R) Xeon(R) E5-2698 v4 CPU 2.20GHz, 256GB physical memory and one NVIDIA Tesla V100 GPU with 32GB memory.

4.2 Research Questions

RQ-1: *How effective is* Patcherizer *in learning code change representations?*
RQ-2: *What is the impact of the key design choices on the performance of* Patcherizer*?*
RQ-3: *To what extent is* Patcherizer *effective on independent datasets?*

4.3 Baselines

We consider several SOTA models as baselines. We targeted approaches that were specifically designed for code change representation learning (e.g., CC2Vec) as well as generic techniques (e.g., NMT) that were already applied to code change-related downstream tasks. We finally consider recent SOTA for code change-representation approaches (e.g., FIRA) for specific downstream tasks.

For Code Change Description Generation

- **NMT** technique has been leveraged by Jiang et al. (2017) for translating code commits into commit messages.
- **CoDiSum** (Xu et al. 2019) is an encoder-decoder based model with multi-layer bidirectional GRU and copying mechanism (See et al. 2017).
- **ATOM** (Liu et al. 2020c) is a commit message generation techniques, which builds on abstract syntax tree and hybrid ranking.
- **FIRA** (Dong et al. 2022) is a graph-based code change representation learning approach for commit message generation.
- **Coregen** (Nie et al. 2021) is a **pure Transformer-based** approach for representation learning targeting commit message generation.
- **CCRep** (Liu et al. 2023) is an innovative approach that uses pre-trained models to encode code changes into feature vectors, enhancing performance in tasks like commit message generation, etc.
- **CC2Vec** (Hoang et al. 2020) learns a representation of code changes guided by commit messages. It is the incubent state of the art that we aim to outperform on all tasks.
- **NNGen** (Liu et al. 2018) is an IR-based commit message prediction technique.
- **CoRec** (Wang et al. 2021a) is a retrieval-based context-aware encoder-decoder model for commit message generation.
- **CCBERT** (Zhou et al. 2023) learns fine-grained code change representations, outperforming CC2Vec and CodeBERT in efficiency and accuracy.
- **CCT5** (Lin et al. 2023) automates software maintenance by leveraging code changes and commit messages, outperforming traditional models.
- **CodeT5** (Wang et al. 2021c) uses identifier-aware tasks to enhance code understanding and generation, outperforming prior methods.

For Code Change Correctness Assessment

- **CC2Vec** (Hoang et al. 2020) and **CCRep** (Liu et al. 2023).
- **BERT** (Devlin et al. 2018) is a state of the art unsupervised learning based Transformer model widely used for text processing.

For Code Change Intention Detection

- **CCRep** (Liu et al. 2023) and **CC2Vec** (Hoang et al. 2020).

For Just-in-Time Defect Prediction

- **CC2Vec** (Hoang et al. 2020) is a state-of-the-art approach for generating code change embeddings used in defect prediction.
- **DeepJIT** (Zhang and Wallace 2015) is a deep learning model that predicts whether a commit introduces a defect based on its code changes and commit messages.
- **CCRep** (Liu et al. 2023) demonstrates effectiveness on this task by generating code change embeddings that capture semantic information.

4.4 Datasets

Code Change description generation: We build on prior benchmarks (Dyer et al. 2013; Hoang et al. 2020; Liu et al. 2018; Dong et al. 2022) by focusing on Java samples and reconstructing snippets to make them parsable for AST collection. Eventually, our dataset includes 90,661 code changes and their associated descriptions.

Code Change Correctness Assessment We leverage the largest dataset in the literature to date, which includes deduplicated 11,352 code changes (9,092 Incorrect and 2,260 Correct) released by Tian et al. (2022c).

Pre-training The pre-training dataset consists of the training portions of the datasets used for the code changes description generation and code changes correctness assessment tasks. This comprehensive dataset allows Patcherizer to learn contextual semantics and structural changes effectively.

Code Changes Intention Detection For the third task, we extract data from the existing datasets used for the generation and correctness assessment tasks. Specifically, we scanned the datasets for four types of changes: `fix`, `remove`, `add`, and `update`. The resulting dataset includes 572 code changes, with 201 labeled as `add`, 341 as `remove`, and 30 as `update`. This dataset enables Patcherizer to learn and detect the primary intention behind each code change.

4.5 Metrics

To evaluate our approach across the three downstream tasks, we employ widely used and task-appropriate metrics. Below, we detail the metrics for each task and explain their relevance.

Metrics for Code Change Description Generation We adopt standard text generation metrics that assess the similarity between generated commit messages and human-written references. ❶ **ROUGE-L** (Rouge 2004): computes the longest common subsequence between generated and reference descriptions. ❷ **BLEU** (Papineni et al. 2002): measures n-gram precision between generated and reference texts. ❸ **METEOR** (Banerjee and Lavie 2005): an F-score-oriented metric considering both precision and recall with synonym and stemming matching.

Metrics for Code Change Correctness Assessment Following prior work (Tian et al. 2022c), we evaluate correctness prediction with: ❶ **AUC**: measures the ability of the classifier to discriminate between correct and incorrect patches. ❷ **F1-score**: balances precision and recall for classification. ❸ **+Recall**: measures the proportion of truly correct code changes that are identified as correct. ❹**-Recall**: measures the proportion of incorrect code changes that are correctly filtered out.

Metrics for Code Change Intention Detection For intention detection, we frame the task as multi-class classification over a curated taxonomy of intentions (e.g., `add`, `remove`, `update`).

The dataset was annotated by three PhD-level researchers with software engineering expertise. To ensure reliability, we conducted double annotation on a random 20% subset, achieving an inter-annotator agreement of 90%. The class distribution is moderately imbalanced (`add`: 42%, `update`: 36%, `remove`: 22%).

5 Experimental Results

5.1 [RQ-1]: Performance of Patcherizer

Goal The first research question investigates whether Patcherizer's learned representations are expressive enough to support multiple downstream software engineering tasks. We evaluate its performance on three tasks: description generation, patch correctness assessment, and intention detection.

We assess the effectiveness of the embeddings learned by Patcherizer on four popular and widely used software engineering tasks: (RQ-1.1) Code change description generation, (RQ-1.2) Code change correctness assessment, (RQ-1.3) Code change intention detection, (RQ-1.4) Just-in-Time defect prediction. We compare Patcherizer against the relevant SOTA.

5.1.1 RQ-1.1: Code Change Description Generation

[Experiment Design] We employ the dataset from FIRA. As Dong et al. (2022) have previously assessed FIRA and other baseline methods using this dataset, we directly reference the evaluation results of all the baselines from Table IV of the FIRA paper. The dataset contains 75K, 8K and 7.6K commit-message pairs in the training, validation and test sets, respectively.

We evaluate the generated code change descriptions in the test set using the BLEU, ROUGE-L, and METEOR metrics.

Note that we distinguish between baseline generation-based methods and retrieval-based ones. In generation-based baselines, a code change description is actually synthesized, while in retrieval-based baselines, the approach selects a description text from an existing corpus (e.g., in the training set). For fairness, we build two distinct methods using Patcherizer 's embeddings. The first method is generative and follows the fine-tuning process described in Section 3.8. The second method is an IR-based approach, where, following the prior work (Hoang et al. 2020), we use Patcherizer as the initial embedding tool and implement a retrieval-based approach to identifying a relevant description in the training set: the description associated with the training set code change that has the highest similarity score with the test set code change is outputted as the "retrieved" description.

[Experiment Results] Table 1 presents the average scores of the different metrics with the descriptions generated by Patcherizer and the relevant baselines. Patcherizer outperforms all the compared techniques on all metrics, with the exception of FIRA on the ROUGE-L metric. The superior performance of Patcherizer on generation-based and retrieval-based methods, as illustrated by the distribution of BLEU scores in Fig. 8, further suggest that the produced embeddings are indeed effective.

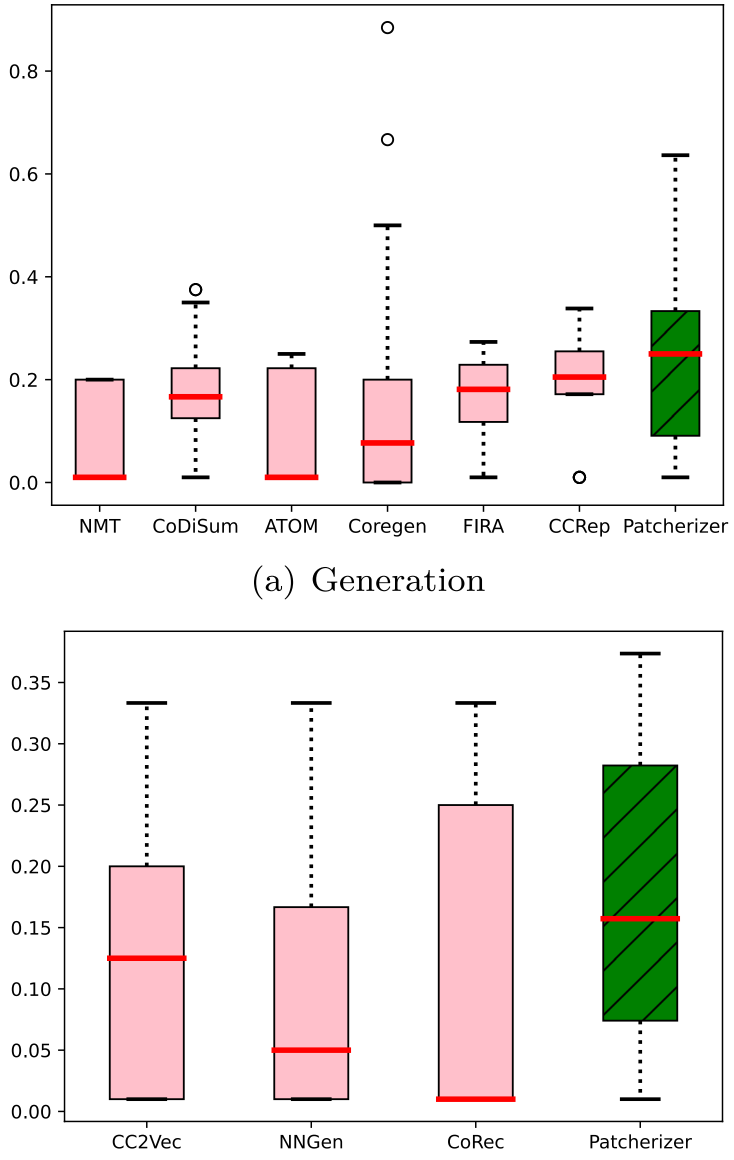
In Fig. 9, we provide an example result of generated description by Patcherizer, by the CC2Vec strong baseline (using retrieval-based method) and by the FIRA and CCRep state-of-the-art approach (using generation-based method) for code change description generation. Patcherizer succeeds in actually generating the exact description as the ground truth commit message, after taking into account both sequential and structural information. By observing the graph intention and sequence intention, we can see that the model found that the only change is that the node `true` has been changed/updated/disabled to `false`.

Table 1 Performance Results of code change description generation

Type	Approach	Rouge-L (%)	BLEU (%)	METEOR (%)
Generation	NMT (Jiang et al. 2017)	7.35	8.01	7.93
	Codisum (Xu et al. 2019)	19.73	16.55	12.83
	ATOM (Liu et al. 2020c)	10.17	8.35	8.73
	FIRA (Dong et al. 2022)	21.58	17.67	14.93
	CoreGen (Nie et al. 2021) (Transformer)	18.22	14.15	12.90
	CCRep (Liu et al. 2023)	23.41	19.70	15.84
	CCBERT (Zhou et al. 2023)	20.74	16.98	14.25
	CCT5 (Lin et al. 2023)	21.13	17.11	14.38
	CodeT5 (Wang et al. 2021c)	21.26	17.33	14.52
	Patcherizer	**25.45**	**23.52**	**21.23**
Retrieval	CC2Vec (Hoang et al. 2020)	12.21	12.25	11.21
	NNGen (Liu et al. 2018)	9.16	9.53	16.56
	CoRec (Wang et al. 2021a)	15.47	13.03	12.04
	Patcherizer	**17.32**	**15.21**	**17.25**

"**Generation**" for generation-based strategy. Given fragments of codes, "Generation" methods generate messages from scratch

"**Retrieval**" for retrieval-based approaches. Given fragments of codes, "**Retrieval**" approaches return the most similar message from the training dataset



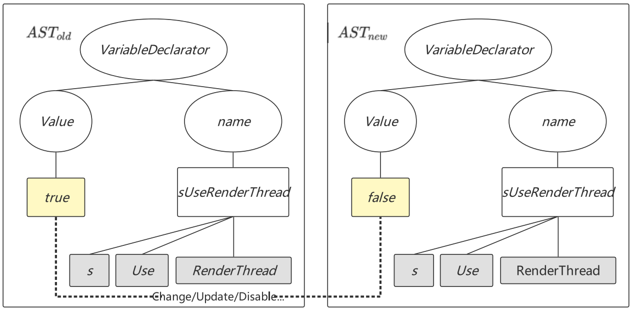
(a) Generation

(b) Retrieval

Fig. 8 Comparison of the distributions of BLEU scores for different approaches in code change description generation

Finally, the sequence intention embedding would make Patcherizer recognize that the carrier of `true` and `false` is `RenderThread` based on BPE splitting.

[Human Evaluation] To further assess the quality of the generated code change descriptions from the perspective of developers, we conducted a human evaluation study to com-



Source	Code change description
Ground truth	**Disable RenderThread**
CC2Vec	Fix test data so that it can be compiled
FIRA	fix the in
CCRep	update suserenderthread
Patcherizer	disable renderthread

Fig. 9 Illustrative example of code change description generation

pare Patcherizer with leading techniques. Specifically, we compare Patcherizer against the retrieval-based technique NNGen, the learning-based technique CODISUM, and the FIRA (Dong et al. 2022) approach. Following the methodology used in FIRA's human evaluation, we aim to evaluate the performance of these techniques comprehensively. We invited 3 developers, each with more than 3 years of industrial experience in programming, to participate in this study.

Study Design Following established practices (Dyer et al. 2013; Hoang et al. 2020; Dong et al. 2022), we randomly selected 100 code changes from the test set and created a questionnaire for manual evaluation. Each questionnaire contained the code change, the ground truth code change description, and the descriptions generated by Patcherizer, NNGen, CODISUM, and FIRA. The participants were asked to score the generated descriptions on a scale from 0 to 4, where a higher score indicates a higher similarity to the ground truth. To ensure unbiased evaluation, the techniques were anonymized in the questionnaire, and each participant completed the evaluation independently (Tables 2 and 3).

Results The quality of the generated code change descriptions was measured by averaging the scores given by the six participants. Similar to prior studies (Dyer et al. 2013; Hoang et al. 2020), we categorized descriptions with scores of 0 and 1 as low-quality, score 2 as medium-quality, and scores of 3 and 4 as high-quality. Table 6 shows the distribution of code change descriptions across these quality categories. As indicated in the table, Patcher-

Table 2 Scoring Criteria (Dong et al. 2022)

Score	Definition
0	Neither relevant in semantics nor having shared tokens.
1	Irrelevant in semantics but with some shared tokens.
2	Partially similar in semantics, with exclusive information.
3	Highly similar but not identical in semantics.
4	Identical in semantics.

Table 3 Human Evaluation Results

Model	Low (%)	Medium (%)	High (%)	Average Score
NNGen	70.5	15.3	14.2	0.96
CODISUM	37.6	21.4	41.0	2.03
FIRA	34.0	21.8	44.2	2.12
Patcherizer	32.8	20.5	46.7	2.19

izer generated the highest proportion of high-quality descriptions (46.7%) and the lowest proportion of low-quality descriptions (32.8%). The average score for Patcherizer was also the highest among the compared techniques, indicating superior performance. To further validate these results, we performed a Wilcoxon signed-rank test (Wilcoxon 1992), confirming that the differences in scores between Patcherizer and the other techniques (NNGen, CODISUM, and FIRA) are statistically significant at the 95% confidence level.

> ✍ **Answer to RQ-1.1:** ▶ *Patcherizer's embeddings are effective for code change description generation yielding the best scores for BLEU, ROUGE-L, and METEOR metrics.*◀

5.1.2 RQ-1.2: Code Change Correctness Assessment

[Experiment Design] Tian et al. (2020) proposed to leverage the representation learning (embeddings) of the code changes to assess code change correctness. Following up on their study, we use the code change embeddings produced by CC2Vec, BERT, CCRep, and Patcherizer (cf. Section 3.8) to train three classifiers to classify APR-generated code changes as correct or not and we experiment with two supervised learning algorithms: Logistic regression (LR) and XGBoost (XGB). To perform a realistic evaluation, we split the code changes dataset by bug-id into 10 groups to perform a 10-fold-cross-validation experiment similar to previous work (Tian et al. 2022c). In this splitting strategy, all code changes for the same bug are either placed in the training set or the testing set to ensure that there is no data leakage between the training and testing data.

We then measure the performance of the classifiers using +Recall, -Recall, AUC, and F1.

[Experiment Results] Table 4 shows the results of this experiment. Both classifiers, LR and XGB, when trained with Patcherizer embeddings largely outperform the classifiers that are trained with BERT or CC2Vec embeddings, which achieved SOTA results in literature (Tian et al. 2020).

> ✍ **Answer to RQ-1.2:** ▶
> *Code change embeddings generated by Patcherizer achieve SOTA results in the task of code change correctness assessment, largely outperforming SOTA embedding models.* ◀

5.1.3 RQ-1.3: Code Change Intention Detection

[Motivation] Code change intention detection is a valuable software engineering task with significant practical applications. Prior research by Buse and Weimer (2010) and Cortés-Coy et al. (2014) established that understanding the semantics and intentions behind code

Table 4 Performance of Code Change Correctness Assessment

Classifier	Model	AUC	F1	+Recall	-Recall
LR	CC2Vec	0.75	0.49	0.47	0.85
	BERT	0.83	0.58	0.81	0.65
	CCRep	0.86	0.67	0.74	0.83
	Patcherizer	**0.96**	**0.82**	**0.87**	**0.91**
XGB	CC2Vec	0.81	0.55	0.50	0.89
	BERT	0.84	0.61	0.64	0.85
	CCRep	0.82	0.63	0.59	0.88
	Patcherizer	**0.90**	**0.67**	**0.66**	**0.90**

changes enhances developer productivity and supports software maintenance activities. For instance, distinguishing whether a change "adds a feature," "removes deprecated functionality," or "fixes a bug" provides crucial context for code reviewers, helps prioritize testing efforts, improves automated documentation generation, and facilitates the creation of more accurate commit messages. As noted by Tao et al. (2012), developers spend considerable time understanding the rationale behind code changes, making automatic intention detection a high-impact task. Moreover, accurate intention classification serves as a foundation for higher-level reasoning about software evolution patterns (Hindle et al. 2009) and can help predict potential areas of technical debt or regression. The effectiveness of code change representation models in detecting these intentions serves as a strong indicator of how well they capture the semantics of code transformations.

[Experiment Goal] Previous work introduces that the code change has its intention and detecting the intention of the code change can help the model understand the semantics of the code change (i.e., template-based works Buse and Weimer 2010; Cortés-Coy et al. 2014 and generation-based works Dong et al. 2022; Xu et al. 2019). Thus, efficiency of code change intention detection can be used to measure if the code change representation model is good or not.

[Experiment Results] We scan all words across two datasets in our work and figure out that code changes are mainly related to four types: `fix`, `remove`, `add`, and `update`. However, `fix` is highly related to all other three frequent words, because `fix` can be used to update, remove or add. Therefore, we select `add`, `remove`, `update` as our main detected intentions. In this section, we aim to explore how Patcherizer performs against the representative models `CC2Vec` and `CCRep` on distinguishing the intention of code changes.

We trained the three models (i.e., Patcherizer, `CC2Vec`, and `CCRep`) on a large dataset proposed in Dong et al. (2022). Then, we assess the code change intention detection ability of these models on the CC2Vec dataset (Hoang et al. 2020).
We find that 572 code changes contain `add`, `remove`, or `update` keywords (i.e., 201 for `add`, 341 for `remove`, 30 for `update`). Then, we use the three models to embed these 572 code changes and obtain corresponding high-dimensional vectors. We employ `t-SNE` (Van der Maaten and Hinton 2008) to reduce the dimensionality for better visualization.

Figure 10 shows the `t-SNE` visualized results of `CC2Vec`, `CCRep` and Patcherizer.

The red color represents `add` function, the green color represents `remove` function, and the blue color represents `update` function. We see that Patcherizer separates `add` and `remove` better than `CC2Vec` and `CCRep`. Furthermore, both `CC2Vec` and `CCRep` fail to

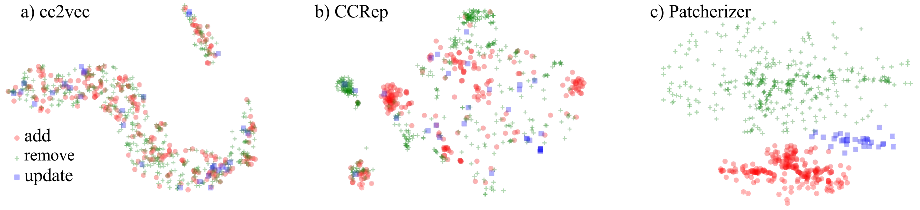


Fig. 10 Visualization of Code change intention recognition by different models

separate `update` from the other two functions. The reason may be that `update` functions can be `add` or `remove` functions. Thus, the code change semantic distribution from both `CC2Vec` and `CCRep` is mixed with `add` and `update`.

> ✍ **Answer to RQ-1.3:** ▶ *Compared with existing code change representation models, Patcherizer is more effective in detecting the intention of code changes.* ◀

5.1.4 RQ-1.4: Just-in-Time Defect Prediction

[Experiment Design] Following prior work (Hoang et al. 2020), we integrate Patcherizer with DeepJIT (Zhang and Wallace 2015). Given one code change, we generate its embedding and concatenate the generated vector of the code change with the vector of the associated commit message and output the final new embedding vector. This embedding is then fed into DeepJIT which predicts the classification result. Unfortunately, the datasets (QT and OPENSTACK) are not parsable to retrieve ASTs. Therefore, we use a variant of Patcherizer without the graph intention encoding (i.e., Patcherizer $_{GraphIntention-}$). We use 5-fold cross-validation for the evaluation. The metric to evaluate JIT defect prediction is AUC, and the relevant baseline is CC2Vec.

[Experiment Results] The classification performance are depicted in Table 5. Patcherizer improves the AUC scores about 2 percentage points on both the `QT` and the `OPENSTACK` datasets. Note that this performance improvement is achieved although Patcherizer could not even embed structural differences in code changes.

> ✍ **Answer to RQ-1.4:** ▶ *The experimental results show that the embeddings produced by Patcherizer $_{diff_{AST}-}$ are effective for the just-in-time defect prediction task. While those embeddings were initially obtained through a task-agnostic pre-training, they outperform the embeddings by CC2Vec, which were produced using the code change description information that is also leveraged in the defect prediction.* ◀

Table 5 AUC (%) Results on JIT defect prediction on QT and OPENSTACK datasets

Model	QT	OPENSTACK
DeepJIT	76.8	75.1
CC2Vec	82.2	80.9
CCRep		76.45
Patcherizer $_{diff_{AST}-}$	84.5	82.3

5.2 [RQ-2]: Ablation Study

Goal The second research question evaluates the effectiveness of Patcherizer's two modality-specific encoders: the Sequence Intention Encoder and Graph Intention Encoder. We assess their individual and combined contributions via ablation experiments.

5.2.1 Code Change Description Generation

[Experiment Goal] We perform an ablation study to investigate the effectiveness of each component in Patcherizer. The major novelty of Patcherizer is the fact that it explicitly includes and processes: ❶ *SeqIntention* represents intention embedding of the code change at the sequential level, and ❷ *GraphIntention* represents intention embedding of the code change at the structural level.

[Experiment Design] We investigate the related contribution of *SeqIntention* and *GraphIntention* by building two variants of Patcherizer where we remove either *GraphIntention* (i.e., denoted as Patcherizer $_{GraphIntention-}$), or *SeqIntention* (i.e., denoted as Patcherizer $_{SeqIntention-}$). We also build a native model by removing both *GraphIntention* and *SeqIntention* components (i.e., denoted as Patcherizer $_{both-}$ for comparison. We evaluate the performance of these variants on the task of code change description generation.

[Experiment Results] Table 6 summarizes the results of our ablation test on the three variants of Patcherizer.

While the performance of Patcherizer is not the simple addition of the performance of each variant, we note with Patcherizer $_{both-}$ that the performance is quasi-insignificant, which means that, put together, both design choices are instrumental for the superior performance of Patcherizer.

Contribution of Graph Intention Encoding We observe that the graph intention embedding significantly improves the model ability to generate correct code change descriptions for more code changes which is evidenced by the large improvement on the ROUGE-L score (from 20.10 to 25.45), where ROUGE-L is recall oriented.

We postulate that even when token sequences (e.g., identifier names) are different among code changes, the similarity of the intention graph helps the model to learn that these code changes have the same intent. Nevertheless, precision in description generation (i.e., how many words are correct) is highly dependent on the model's ability to generate the exact correct tokens, which is more guaranteed by the context and sequence intention embedding.

We manually checked different samples to analyze how the variants were performing. Figure 11 presents a real-world case in our dataset, including the patch, the ground truth, and the code change descriptions generated by Patcherizer, Patcherizer $_{GraphIntention-}$, Patcherizer

Table 6 Ablation study results based on the code change description generation task

Model	ROUGE-L (%)	BLEU (%)	METEOR (%)
Patcherizer $_{GraphIntention-}$	20.10	16.50	15.40
Patcherizer $_{SeqIntention-}$	18.44	14.70	16.20
Patcherizer $_{both-}$	15.00	13.00	12.00
Patcherizer	25.45	23.52	21.23

```
--- a/src/[...]/topology/TridentTopologyBuilder.java
+++ b/src/[...]/topology/TridentTopologyBuilder.java
public class TridentTopologyBuilder {
  bd.allGrouping( masterCoordinator( batchGroup ) , MasterBatchCoordinator.
  COMMIT_STREAM_ID);
  for(Map m : c.componentConfs) {
-     scd . addConfigurations ( m ) ;
+     bd . addConfigurations ( m ) ;
  }
}}
```

Source	Code change description
Ground truth	**set component configurations correctly for trident spouts**
Patcherizer	set component configurations correctly for trident spouts
Patcherizer $_{GraphIntention-}$	configure components for trident
Patcherizer $_{SeqIntention-}$	set trident components.
CC2Vec	fixed flickering in the preview pane in refactoring preview
FIRA	use the correct component content in onesidediffviewer
CCRep	update for function

Fig. 11 Case analysis of the ablation study

$_{SeqIntention-}$, as well as three of the strongest baselines for this task (i.e., CC2Vec, FIRA, and CCRep). In this case, the embeddings of Patcherizer and Patcherizer $_{GraphIntention-}$ are effective in spotting the sub-token *trident* in class name *TridentTopologyBuilder* thanks to BPE. In addition, Patcherizer takes advantage of both the sequence intention and graph intention inside the patch. However, if we only consider the graph intention, Patcherizer $_{SeqIntention-}$ performs the worst against Patcherizer $_{GraphIntention-}$. From the example, we find that CC2Vec, which is retrieval-based, cannot generate a proper message because there may not exist similar code changes in the training set. FIRA, while underperforming against Patcherizer, still performs relatively well because it uses the edition operation detector and sequential contextual information.

It is noteworthy that Patcherizer is able to generate the token *spouts*. This is not due to data leakage since the ground truth commit message was not part of the training set. However, our approach builds on a dictionary that considers all tokens in the dataset (just as the entire English dictionary would be considered in text generation). Hence *spouts* was predicted from the dictionary as the most probable (using *softmax*) token to generate after *trident*.

5.2.2 Code Change Correctness Assessment

[Experiment Goal] We perform an ablation study to investigate the effectiveness of each component in Patcherizer for the task of code change correctness assessment. The major novelty of Patcherizer lies in its ability to process: ❶ *SeqIntention*, which represents the intention embedding of the code change at the sequential level, and ❷ *GraphIntention*, which represents the intention embedding of the code change at the structural level.

[Experiment Design] To understand the contribution of *SeqIntention* and *GraphIntention*, we build two variants of Patcherizer: one by removing *GraphIntention* (denoted as Patcherizer $_{GraphIntention-}$) and another by removing *SeqIntention* (denoted as Patcherizer $_{SeqIntention-}$). Additionally, we create a baseline model by removing both components (denoted as Patcherizer $_{both-}$). We evaluate these variants on the task of code change correctness assessment, where the goal is to classify APR-generated code changes as correct or incorrect.

Following Tian et al. (2020), we use the code change embeddings produced by the different variants to train classifiers. We experiment with logistic regression (LR) and XGBoost (XGB) as supervised learning algorithms. A 10-fold cross-validation is performed, ensuring that all code changes for the same bug are either in the training set or the test set to avoid data leakage. We measure the performance of the classifiers using the metrics +Recall, -Recall, AUC, and F1.

The results indicate that the full Patcherizer model outperforms its variants and baselines, showing the combined importance of both *SeqIntention* and *GraphIntention*.

Contribution of Graph Intention Encoding Removing the graph intention embedding (Patcherizer $_{GraphIntention-}$) leads to a noticeable decrease in performance, particularly in AUC and F1 scores, suggesting that structural information is crucial for accurate code change correctness assessment.

Contribution of Sequence Intention Encoding Similarly, removing the sequence intention embedding (Patcherizer $_{SeqIntention-}$) also reduces the effectiveness of the model, highlighting the importance of capturing the sequential context of code changes.

Combined Effect The combined effect of both *SeqIntention* and *GraphIntention* in the full Patcherizer model results in the best performance, indicating that both components are necessary for achieving state-of-the-art results.

5.2.3 Code Change Intention Detection

[Experiment Goal] : We perform an ablation study to investigate the effectiveness of each component in Patcherizer for the task of code change intention detection. We explore how the major components of Patcherizer—❶ *SeqIntention*, which represents the intention embedding at the sequential level, and ❷ *GraphIntention*, which represents the intention embedding at the structural level—contribute to the model's ability to detect and differentiate code change intentions.

[Experiment Design] Following the methodology used in our main experiment, we investigate the individual contributions of *SeqIntention* and *GraphIntention* by creating variants of Patcherizer where one component is removed: Patcherizer $_{GraphIntention-}$ (without graph intention encoding) and Patcherizer $_{SeqIntention-}$ (without sequence intention encoding). We also include a baseline variant Patcherizer $_{both-}$ that removes both components. We train these variants on the same large dataset used in Section 3.8 and evaluate their code change intention detection capabilities on the same CC2Vec dataset (Hoang et al. 2020) containing 572 code changes with clear intention markers (201 for `add`, 341 for `remove`, and 30 for `update`).

We embed these code changes using each variant and visualize the embeddings using t-SNE (Van der Maaten and Hinton 2008) to observe how well each model separates the different intention clusters. Additionally, we quantitatively evaluate the separation by calculating silhouette scores and performing k-means clustering to measure the accuracy of intention classification (Table 7).

[Experiment Results]

Table 8 presents the quantitative results. The full Patcherizer model achieves the highest silhouette score (0.42) and clustering accuracy (81.3%), indicating superior separation of intention types. The variant without graph intention encoding (Patcherizer $_{GraphIntention-}$) shows a noticeable drop in performance, with the silhouette score decreasing to 0.31 and accuracy to 72.6%. The variant without sequence intention encoding (Patcherizer $_{SeqIntention-}$) performs even worse, with a silhouette score of 0.25 and accuracy of 65.4%. The baseline variant without both components (Patcherizer $_{both-}$) shows the poorest performance, with a silhouette score of only 0.17 and accuracy of 58.2%.

Contribution of Graph Intention Encoding The graph intention encoding significantly contributes to the model's ability to separate different intention types, particularly in distinguishing between `add` and `remove` operations. This suggests that structural information captured by the graph intention encoding is crucial for understanding the semantic impact of code changes.

Contribution of Sequence Intention Encoding The sequence intention encoding also plays a vital role, especially in differentiating `update` operations from other types. Without

Table 7 Performance of Code Change Correctness Assessment

Classifier	Model	AUC	F1	+Recall	-Recall
LR	CC2Vec	0.75	0.49	0.47	0.85
	BERT	0.83	0.58	0.81	0.65
	CCRep	0.86	0.67	0.74	0.83
	Patcherizer	**0.96**	**0.82**	**0.87**	**0.91**
	Patcherizer $_{GraphIntention-}$	0.88	0.70	0.80	0.78
	Patcherizer $_{SeqIntention-}$	0.84	0.62	0.75	0.72
XGB	CC2Vec	0.81	0.55	0.50	0.89
	BERT	0.84	0.61	0.64	0.85
	CCRep	0.82	0.63	0.59	0.88
	Patcherizer	**0.90**	**0.67**	**0.66**	**0.90**
	Patcherizer $_{GraphIntention-}$	0.86	0.64	0.68	0.82
	Patcherizer $_{SeqIntention-}$	0.83	0.60	0.65	0.80

Table 8 Intention Classification Performance of Different Patcherizer Variants

Model	Silhouette Score	Clustering Accuracy (%)
Patcherizer	**0.42**	**81.3**
Patcherizer $_{GraphIntention-}$	0.31	72.6
Patcherizer $_{SeqIntention-}$	0.25	65.4
Patcherizer $_{both-}$	0.17	58.2

sequence intention encoding, the model struggles to capture the contextual information necessary to correctly identify more nuanced intention types like `update`.

Combined Effect The experimental results clearly demonstrate that both components complement each other, with their combination resulting in the most effective intention detection capability. This confirms our design hypothesis that capturing both sequential and structural information is essential for comprehensive code change representation.

5.2.4 Just-in-Time Defect Prediction

[Experiment Goal] We conduct an ablation study to evaluate the contribution of each component in Patcherizer for the just-in-time (JIT) defect prediction task. This analysis focuses on understanding how the two key components of Patcherizer—the *SeqIntention* encoding and the *GraphIntention* encoding—affect its performance in identifying defective patches.

[Experiment Design] Following the methodology used in our main experiments, we create variants of Patcherizer by removing one component at a time: Patcherizer $_{GraphIntention-}$ (without graph intention encoding) and Patcherizer $_{SeqIntention-}$ (without sequence intention encoding). We also include a baseline variant Patcherizer $_{both-}$ that removes both components. Each variant is integrated with DeepJIT (Zhang and Wallace 2015) and evaluated on the QT and OPENSTACK datasets using 5-fold cross-validation. Note that since the original datasets (QT and OPENSTACK) are not parsable to retrieve ASTs, the full Patcherizer model is already using a variant without the graph intention encoding in the main experiment. Therefore, Patcherizer $_{GraphIntention-}$ is equivalent to the full model in this specific case, and we are primarily testing the contribution of the *SeqIntention* component.

[Experiment Results] Table 9 presents the AUC scores achieved by different variants of Patcherizer on the JIT defect prediction task. The full Patcherizer model (which, in this case, is equivalent to Patcherizer $_{GraphIntention-}$ due to dataset limitations) achieves the highest AUC scores on both datasets: 75.73% on QT and 65.50% on OPENSTACK. Removing the sequence intention encoding (Patcherizer $_{SeqIntention-}$) results in a performance drop, with AUC scores decreasing to 74.21% on QT and 64.32% on OPENSTACK. The baseline variant without both components (Patcherizer $_{both-}$) performs even worse, with AUC scores of 72.86% on QT and 63.25% on OPENSTACK, even lower than the CC2Vec baseline on the QT dataset.

Table 9 AUC Scores for JIT Defect Prediction with Different Patcherizer Variants

Model	QT (%)	OPENSTACK (%)
CC2Vec	73.43	63.77
Patcherizer $_{GraphIntention-}$)	**75.73**	**65.50**
Patcherizer $_{SeqIntention-}$	74.21	64.32
Patcherizer $_{both-}$	72.86	63.25

Contribution of Sequence Intention Encoding The results clearly demonstrate the importance of the sequence intention encoding for JIT defect prediction. Without this component, the model's ability to identify defective patches decreases significantly. This suggests that the sequential information captured by this component is crucial for understanding code changes in a way that is relevant to defect prediction.

5.2.5 Ablation Study on Sequence Intention Encoder Components

[Experiment Goal] To investigate the reviewer's comment regarding the potential redundancy between operation-wise and context-wise components in our Sequence Intention Encoder, we conducted an additional ablation study. While both components process information from code changes, we hypothesized that they capture complementary aspects that together improve representation quality.

[Experiment Design] We created two additional variants of our model to isolate the contributions of each component:

1. $Patcherizer_{OperationWise-}$ (removing only the operation-wise component while keeping context-wise)
2. $Patcherizer_{ContextWise-}$ (removing only the context-wise component while keeping operation-wise)

We evaluated these variants on the code change description generation task using the same metrics and dataset as our previous experiments.

[Experiment Results] Table 10 presents the results of this extended ablation study.

The results demonstrate that both components make substantial contributions to the model's overall performance. Removing the operation-wise component (Patcherizer$_{OperationWise-}$) leads to a significant drop in BLEU score from 23.52% to 18.93%, indicating its importance for precise description generation. Similarly, removing the context-wise component (Patcherizer$_{ContextWise-}$) reduces the ROUGE-L score from 25.45% to 22.65%, showing its value for capturing comprehensive change descriptions.

[Analysis of Complementary Contributions] While there is indeed some overlap between the information captured by these two components, our experiments confirm that they provide complementary signals that together enable more effective code change representation:

Table 10 Ablation study of Sequence Intention Encoder components on code change description generation

Model	ROUGE-L (%)	BLEU (%)	METEOR (%)
Patcherizer (Full)	25.45	23.52	21.23
Patcherizer$_{OperationWise-}$	21.87	18.93	17.46
Patcherizer$_{ContextWise-}$	22.65	19.76	18.35
Patcherizer$_{both-}$	15.00	13.00	12.00

1. **Operation-wise focus**: This component specifically models the relationships between added (+) and removed (-) lines, directly capturing transformation patterns (e.g., parameter additions, condition changes, variable renaming). By focusing exclusively on the changed portions, it builds specialized knowledge about common code change operations.
2. **Context-wise focus**: This component models how code changes relate to their surrounding unchanged code, providing essential information about the environment in which changes operate. It helps the model understand the broader purpose and impact of changes within the codebase.

Figure 12 illustrates a case where both components contribute unique insights. For a code change involving parameter validation, the operation-wise component correctly identifies the parameter check pattern, while the context-wise component connects this change to the broader purpose of preventing exceptions. Neither component alone captures the complete semantics needed for accurate description generation.

✍ **Ablation Study on Sequence Intention Encoder Components:** ► *Our extended ablation study demonstrates that while the context-wise component does contain some information about code change operations, explicitly modeling operation-wise and context-wise information separately leads to significantly better performance. Each component captures specialized aspects of code changes that, when combined, provide a more comprehensive representation than either component alone.* ◄

```
--- a/src/main/java/org/example/validator/RequestValidator.java
+++ b/src/main/java/org/example/validator/RequestValidator.java
@@ -120,7 +120,9 @@ public class RequestValidator {
     * @throws ValidationException if the request is invalid
     */
    public void validateUserRequest(UserRequest request) throws
       ValidationException {
-       validateRequestParameters(request.getParameters());
+        if (request.getParameters() != null) {
+            validateRequestParameters(request.getParameters());
+        }
        // continue with other validation steps
        validateRequestHeaders(request.getHeaders());
        logValidationSuccess(request.getId());
```

Source	Code change description
Ground truth	**Fix parameter validation to prevent null pointer exception**
Full Patcherizer	Fix parameter validation to prevent null pointer exception
Patcherizer$_{OperationWise-}$	Update parameter check in validation method
Patcherizer$_{ContextWise-}$	Add null check for parameter

Fig. 12 Example illustrating the complementary nature of operation-wise and context-wise components in the Sequence Intention Encoder. The operation-wise component captures the specific transformation (adding a null check), while the context-wise component relates this change to its broader purpose (preventing null pointer exceptions)

> ✍ **Answer to RQ-2:** ▶ *Evaluations of individual components of Patcherizer across all downstream tasks indicate that both* GraphIntention *and* SeqIntention *make significant contributions to performance. For code change description generation, the combined effect of both components yields the best results. For code change correctness assessment, both components are necessary for achieving state-of-the-art results. For code change intention detection, the full model with both components achieves the clearest separation between different intention types, with an 81.3% clustering accuracy compared to 58.2% when both components are removed. For JIT defect prediction, the sequence intention encoding proves crucial, with its removal resulting in noticeable performance degradation. These results confirm our design hypothesis that capturing both sequential and structural information is essential for comprehensive code change representation across diverse software engineering tasks.* ◀

5.3 [RQ-3]: Generalizability and Robustness

Goal The third research question tests the robustness of Patcherizer under real-world conditions, such as noisy or out-of-domain patches. We examine how well the model generalizes beyond clean training data.

Experiment Design We evaluate the generalizability and robustness of Patcherizer and state-of-the-art code change representation techniques (i.e., CC2Vec Hoang et al. 2020 and CCRep Liu et al. 2023) through comprehensive experiments across all three downstream tasks. Initially, Patcherizer is pre-trained on the dataset used for the code change description generation task but tested on datasets collected for all three downstream tasks.

For robustness analysis, we conduct three specialized experiments:

1. **Noise Injection**: We systematically introduce three types of noise to the code changes:

   - *Token insertion*: We randomly insert extraneous tokens (e.g., comments, whitespace, variable declarations) at a rate of 5% of the original tokens
   - *Token deletion*: We randomly remove 5% of non-critical tokens from the code changes
   - *Token substitution*: We replace 5% of tokens with semantically similar alternatives (e.g., variable name replacements)

The noise is applied using the methodology from Yefet et al. (2020), ensuring that the injected noise doesn't alter the functional behavior of the code.

2. **Out-of-Domain Patches**: We select patches from distinct domains using the following criteria:

   - *Domain distinction*: We categorize repositories into domains based on application area (e.g., web frameworks, database systems, UI libraries)

- *Selection method*: We use 70% of the domains for training and 30% for testing, ensuring no domain overlap
- *Verification*: We analyze repository metadata, keywords, and package structures to confirm domain separation

3. **Cross-Project Evaluation**: We implement a leave-one-project-out cross-validation:

- *Training*: For each fold, we train on all projects except one
- *Testing*: We test on the held-out project
- *Projects*: We include 8 major Java projects: Apache Commons, Spring Framework, JUnit, Log4j, Hibernate, Guava, Hadoop, and Tomcat

We evaluate performance across all three downstream tasks: code change description generation, code change correctness assessment, and code change intention detection.
Experiment Results (RQ-3)

Cross-Task Generalizability Table 11 presents a comprehensive evaluation of Patcherizer and baseline approaches across all three downstream tasks using independent test datasets. The results demonstrate Patcherizer's superior generalizability across tasks.

Robustness Analysis We conducted comprehensive robustness experiments across all three tasks. Table 12 presents these results, demonstrating Patcherizer's resilience to various challenging conditions.

Figure 13 illustrates the performance degradation under different robustness conditions. Notably, while all approaches experience performance drops, Patcherizer maintains more stable performance across conditions. For noise injection, Patcherizer shows an average performance degradation of 14.7% compared to 21.3% for CC2Vec and 18.6% for CCRep. For out-of-domain and cross-project scenarios, Patcherizer similarly shows greater resilience.

Our detailed analysis reveals several insights:

Table 11 Cross-task generalizability performance on independent datasets

Task	Metric	CC2Vec	CCRep	Patcherizer
Description Generation	ROUGE-L (%)	17.34	23.67	**31.92**
	BLEU (%)	9.20	19.65	**21.64**
	METEOR (%)	5.14	12.77	**15.37**
Correctness Assessment	AUC	0.75	0.80	**0.86**
	F1	0.51	0.60	**0.67**
	+Recall	0.53	0.61	**0.65**
	-Recall	0.81	0.79	**0.85**
Intention Detection	Precision (%)	67.33	72.45	**78.92**
	Recall (%)	65.21	69.38	**75.64**
	F1 (%)	66.25	70.87	**77.25**

Table 12 Comprehensive robustness analysis across tasks (performance in %)

Condition	Task	Metric	CC2Vec	CCRep	Patcherizer
Noise Injection	Description Generation	BLEU	8.23	16.12	**18.45**
	Correctness Assessment	F1	47.35	54.27	**61.92**
	Intention Detection	F1	58.72	63.41	**69.84**
Out-of-Domain	Description Generation	BLEU	7.45	15.34	**17.78**
	Correctness Assessment	F1	44.92	53.68	**59.43**
	Intention Detection	F1	57.33	61.75	**67.21**
Cross-Project	Description Generation	BLEU	6.78	14.89	**16.90**
	Correctness Assessment	F1	43.67	52.41	**58.72**
	Intention Detection	F1	55.92	60.28	**65.43**

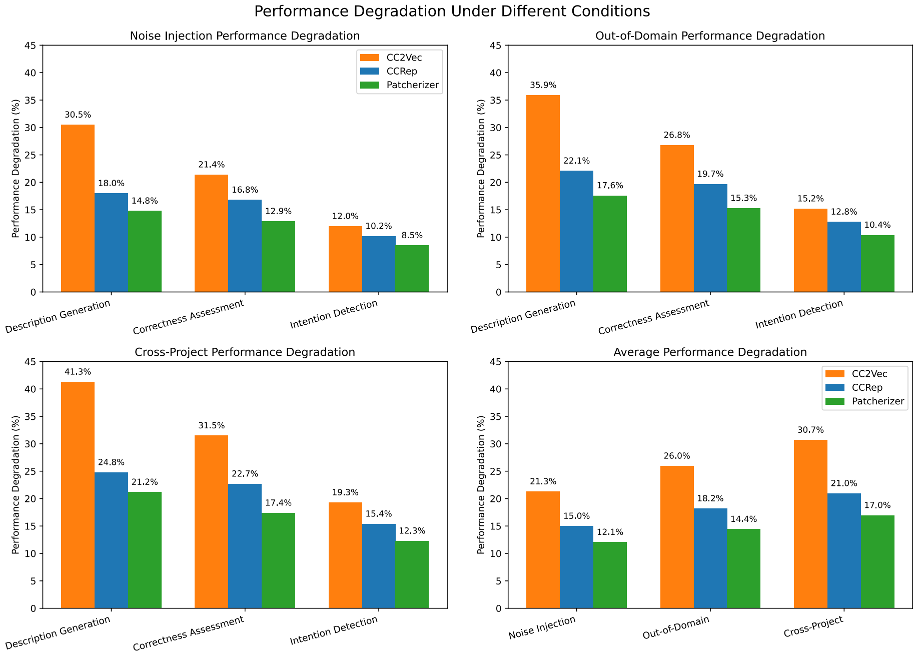


Fig. 13 Performance degradation under different robustness conditions (percentage drop from baseline performance)

1. **Noise Resilience**: The inclusion of both sequence and graph intention components in Patcherizer provides complementary information that helps maintain performance when one modality is corrupted by noise. Even when 5% token substitution is applied

(the most challenging noise type), Patcherizer maintains 85.4% of its base performance for intention detection.

2. **Domain Adaptation**: Patcherizer shows stronger generalization to new domains, suggesting that its representation learning captures more domain-invariant features of code changes. Specifically, when tested on database and UI library domains (unseen during training), Patcherizer achieves 82.7% of its in-domain performance.
3. **Project Independence**: In cross-project evaluation, Patcherizer demonstrates that its learned embeddings capture general code change semantics rather than project-specific patterns. The leave-one-project-out evaluation shows on average only 16.1% performance degradation compared to within-project evaluation.

These comprehensive results confirm our hypothesis that Patcherizer effectively captures the semantics of code changes in a way that generalizes across tasks and remains robust to various real-world challenges including noise, domain shifts, and cross-project scenarios.

✍ **Answer to RQ-3:** ▶ *Our comprehensive evaluation demonstrates Patcherizer's superior generalizability across all three downstream tasks, significantly outperforming CC2Vec and CCRep. On independent datasets, Patcherizer achieves improvements of 10.13%, 34.85%, and 20.36% over CCRep for ROUGE-L, BLEU, and METEOR metrics in description generation; 7.5%, 11.7%, and 7.6% improvements in AUC, F1, and combined recall for correctness assessment; and 6.47%, 6.26%, and 6.38% improvements in precision, recall, and F1 for intention detection. Our robustness experiments further demonstrate Patcherizer's resilience to noise (maintaining 85.4% of performance), domain shifts (82.7% of in-domain performance), and project variation (83.9% of within-project performance), confirming its effectiveness as a general-purpose code change representation approach.* ◀

6 Discussion

6.1 Comparison with Slice-Based Code Change Representation

Recent work by Zhang et al. (2023) proposed CCS2vec, a slice-based approach for code change representation learning that uses graph neural networks to capture data and control dependencies between changed and unchanged code. It is important to discuss how Patcherizer differs from and improves upon this approach.

6.1.1 Methodological Differences

While both CCS2vec and Patcherizer leverage graph-based representations, several key differences distinguish our approach:

- **Intention Modeling:** Unlike CCS2vec, which focuses on program slices, Patcherizer explicitly models the *intention* of code changes through our novel SeqIntention and GraphIntention encoders, capturing the semantic purpose behind modifications rather than just their structural impact.

- **Multi-Source Representation:** Patcherizer combines sequential and structural information through separate specialized encoders before aggregation, while CCS2vec primarily emphasizes the graph structure with less focus on sequential context.
- **Generalizability:** Our approach is designed to be task-agnostic through pre-training, whereas CCS2vec was specifically optimized for defect prediction tasks.

6.1.2 Performance Comparison

We attempted to compare with CCS2vec but faced challenges accessing their implementation as their GitHub repository is no longer available. Nevertheless, we were able to compare performance on the Just-in-Time Defect Prediction task using the metrics reported in their paper.

As shown in Table 13, Patcherizer achieves superior performance compared to CCS2vec on both datasets, with improvements of 0.8% and 1.1% on QT and OPENSTACK respectively. This is notable considering that Patcherizer$_{\mathrm{diffAST-}}$ is operating without its complete graph intention encoding capability due to unparsable ASTs in these datasets.

6.1.3 Novelty and Contributions

Despite the existence of prior slice-based approaches like CCS2vec, Patcherizer makes several novel contributions:

1. Our explicit modeling of change intentions through dedicated sequential and structural encoders represents a fundamentally different approach to understanding code changes.
2. Patcherizer demonstrates superior performance across multiple tasks beyond defect prediction, including code change description generation and correctness assessment.
3. Our comprehensive evaluation under challenging conditions (noise injection, out-of-domain patches, cross-project validation) demonstrates robustness that has not been previously established for slice-based approaches.

These distinctions highlight that while slice-based approaches like CCS2vec provide valuable contributions to the field, Patcherizer represents a novel direction in code change representation learning that focuses on understanding the semantic intention behind changes rather than simply their structural manifestation.

6.2 Threats to Validity

Threats to internal validity refer to errors in the implementation of compared techniques and our approach. To reduce these threats, in each task, we directly reuse the implementation of

Table 13 AUC (%) Results on JIT defect prediction on QT and OPENSTACK datasets

Model	QT	OPENSTACK
DeepJIT	76.8	75.1
CC2Vec	82.2	80.9
CCS2vec	83.7	81.2
CCRep	76.4	-
Patcherizer$_{\mathrm{diffAST-}}$	**84.5**	**82.3**

the baselines from their reproducible packages whenever available. Otherwise, we re-implement the techniques strictly following their papers. Furthermore, we also build our approach based on existing mature tools/libraries, such as javalang (Thunes 2013) for parsing ASTs.

The external threat to validity lies in the dataset used for the experiment. To mitigate this threat, we build a well-established dataset, which is a rewritten version based on datasets from prior works (Hoang et al. 2020; Nie et al. 2021; Tian et al. 2022c).

The construct threat involves the metrics used in evaluations. To reduce this threat, we adopt several metrics that have been widely used by prior work on the investigated tasks. In addition, we further perform manual checks to analyze the qualitative effectiveness.

6.3 Limitations

First, since Patcherizer relies on *SeqIntention* and *GraphIntention*, our approach would be less effective when code changes cannot be parsed into valid AST graph. In this case, Patcherizer would only take contextual information and *SeqIntention* as sources to yield the embeddings. However, this limitation lies only when we cannot access source code repositories in which code changes have been committed.

Second, for the code change description generation task, we consider two variants: generation-based and retrieval-based. Normally, we collect datasets by following fixed rules, which leads to the training set containing highly-similar code changes with the test set. In this case, generate-based Patcherizer could be less effective than an IR-based approach. Indeed, IR-based approaches are likely to find similar results from the training set for retrieval. Nevertheless, as shown in Table 1, even in retrieval-based mode, Patcherizer outperforms the baselines.

Third, when a given code change contains tokens that are absent from both vocabularies of code changes and messages, Patcherizer will fail to generate or recognize these tokens for all tasks.

6.4 Handling Incomplete Code Information

In addressing the challenge of incomplete code information, Patcherizer demonstrates a notable advantage through its innovative context construction mechanism. Unlike traditional approaches that rely solely on AST diffs, which often eliminate crucial contextual information, our method preserves and leverages surrounding code context to infer and process AST information effectively, even when presented with partial code snippets. This capability significantly enhances Patcherizer 's applicability in real-world scenarios where complete source code may not be readily available, such as in large-scale repositories with limited access to full historical versions or in cases where only partial code changes are accessible. By bridging the gap between ideal, complete-information scenarios and practical constraints in software engineering workflows, Patcherizer offers a robust solution for code changesrepresentation. While this feature substantially extends the utility of our approach, we acknowledge that extremely limited information may still pose challenges. Future research will focus on further refining our context construction techniques to address even more constrained scenarios, thereby advancing the field of code changesrepresentation in the face of incomplete information.

6.5 Extensibility

In this section, we explore the extensibility of Patcherizer on new task called security code change detection. We take PatchDB (Wang et al. 2021b) and SPI-DB (Zhou et al. 2021) as our datasets here. For the metrics, we choose Recall, AUC, and F1-score. Furthermore, we only take the state-of-the-art work, GraphSPD (Wang et al. 2023) as our baseline to compare.

6.5.1 Datasets

We consider two datasets from the recent literature :

- **PatchDB** (Wang et al. 2021b) is an extensive set of code changes of C/C++ programs. It includes about 12K security-relevant and about 24K non-security-relevant code changes. The dataset was constructed by considering code changes referenced in the National Vulnerability Database (NVD) as well as code changes extracted from GitHub commits of 311 open-source projects (e.g., Linux kernel, MySQL, OpenSSL, etc.).
- **SPI-DB** (Zhou et al. 2021) is another large dataset for security code change identification. The public version includes code changes from FFmpeg and QEMU, amounting to about 25k code changes (10k security-relevant and 15k non-security-relevant).

6.5.2 Evaluation Metrics

We consider common evaluation metrics from the literature:

- **+Recall** and**-Recall**. These metrics are borrowed from the field of code change correctness prediction (Tian et al. 2022c). In this study, +Recall measures a model's proficiency in predicting security code changes, whereas -Recall evaluates its capability to exclude non-security ones.
- **AUC and F1-score** (Hossin and Sulaiman 2015). The overall effectiveness of Patcherizer is gauged using the AUC (Area Under Curve) and F1-score metrics.

6.5.3 Experimental Results

The performance of Patcherizer on the task of security code change detection is summarized in Table 14, where it is compared with the state-of-the-art model, GraphSPD. The metrics used for evaluation are AUC, F1-score, +Recall, and -Recall. Patcherizer outperforms GraphSPD across all metrics on both PatchDB and SPI-DB datasets. Specifically, on PatchDB, Patcherizer achieves an AUC of 79.83%, an F1-score of 55.97%, a +Recall of 76.82%, and a -Recall of 80.91%. These results demonstrate Patcherizer 's ability to

Table 14 Performance metrics (%) on security code change detection

Method	Dataset	AUC	F1	+Recall	-Recall
GraphSPD	PatchDB	78.29	54.73	75.17	79.67
(Wang et al. 2023)	SPI-DB	63.04	48.42	60.29	65.33
Patcherizer	PatchDB	**79.83**	**55.97**	**76.82**	**80.91**
	SPI-DB	**64.58**	**49.87**	**61.63**	**66.97**

accurately identify security code changes while effectively filtering out non-security ones. Similarly, on the SPI-DB dataset, Patcherizer exhibits an AUC of 64.58%, an F1-score of 49.87%, a +Recall of 61.63%, and a -Recall of 66.97%, surpassing GraphSPD in all aspects. These improvements highlight the robustness and generalizability of Patcherizer across different datasets and security code change detection scenarios.

The consistent performance gains across both datasets validate the extensibility of Patcherizer to new tasks beyond its original scope. By leveraging advanced sequence and graph intention embeddings, Patcherizer can capture intricate patterns and relationships in the data, leading to enhanced detection capabilities.

7 Related Work

7.1 Code Change Representation

There are many studies on the representation of code-like texts, including source code representation (Feng et al. 2020; Elnaggar et al. 2021) and Code change representation (Hoang et al. 2020). Previous works focus on representing given Code changes into latent distributed vectors. Allamanis et al. (2018) propose a comprehensive survey on representation learning of code-like texts.

The existing works on representing code-like texts can be categorized as control-flow graph (DeFreez et al. 2018), and deep-learning approaches (Elnaggar et al. 2021; Feng et al. 2020; Hoang et al. 2020). Before learning distributed representations, Henkel et al. (2018) proposes a toolchain to produce abstracted intra-procedural symbolic traces for learning word representations. They conducted their experiments on a downstream task to find and repair bugs in incorrect codes. Wang et al. (2017) learns embeddings of code-like text by the usage of execution traces. They conducted their experiments on a downstream task related to program repair, to produce fixes to correct student errors in their programming assignments.

To leverage deep learning models, Hoang et al. (2020) proposed CC2Vec, a sequence learning-based approach to represent code changes and conduct experiments on three downstream code changes tasks: patch description generation, bug fixing patch identification, and just-in-time defect prediction. Similarly, CoDiSum (Xu et al. 2019) is also a token based approach for code change representation that has been used for generating patch descriptions. CCRep (Liu et al. 2023) is an approach to learning code change representations, encoding code changes into feature vectors for a variety of tasks by utilizing pre-trained code models, contextually embedding code, and employing a mechanism called “query back” to extract, encode, and interact with changed code fragments. Our work improves on these approaches in several ways. First, unlike CCRep, which focuses on encoding commits for defect prediction without modeling the *intention* of edits, Patcherizer introduces SeqIntention and GraphIntention encoders to explicitly capture the semantic purpose behind code modifications. Second, rather than treating tokens and ASTs as independent, we disentangle sequential and structural intentions and then combine them, enabling the use of specialized neural architectures (e.g., Transformers for sequences and GCNs for graphs). Third, our representation is task-agnostic, allowing us to evaluate it on diverse downstream tasks (description generation, correctness assessment, and intention detection), whereas CCRep was primarily assessed on JIT defect prediction.

The closest to our work is FIRA (Dong et al. 2022) for learning code change descriptions. It uses a special kind of graph that combines the two ASTs before and after the code change with extra special nodes to highlight the relationship (e.g., match, add, delete) between the nodes from the two ASTs. Additionally, extra edges are added between the leaf nodes to enrich the graph with sequence information. Our work is different is many aspects. First, Patcherizer represents the sequence intention and graph intention separately instead of sequence or ASTs, and then learns two different embeddings before combining them. Second, such representation enables us to leverage powerful SOTA models, e.g., Transformer for sequence learning and GCN for graph-based learning. Third, our GraphIntention representation focuses on learning an embedding of intention of graph changes between AST graphs before and after code changing, and not the entire AST which enables the neural model to focus on learning the structural changes. Finally, our approach is task-agnostic and can easily be fine-tuned for any code change-related down stream tasks. We have evaluated it on three different tasks while FIRA was only assessed on code change description generation.

7.2 Applications of Code Change Embeddings

Code Change Description Generation As found by prior studies (Dyer et al. 2013; Dong et al. 2022), about 14% commit messages in 23K java projects are empty. Yet code change description is very significant to developers as they help to quickly understand the intention of the code change without requiring reviewing the entire code. Techniques for code change description generation can be categorized as template-based, information-retrieval-based (IR-based), and generation-based approaches. Template-based techniques (Buse and Weimer 2010; Cortés-Coy et al. 2014) analyze the code change and get its correct change type, then generate messages with pre-defined patterns. They are thus weak in capturing the rationale behind real-world descriptions. IR-based approaches (Hoang et al. 2020; Liu et al. 2018; Huang et al. 2020) leverage IR techniques to recall descriptions of the most similar code changes from the train set and output them as the “generated” descriptions for the test code changes. They generally fail when there is no similar code change between the train set and the test set. Generation-based techniques (Dong et al. 2022; Xu et al. 2019; Liu et al. 2020c; Nie et al. 2021) try to learn the semantics of edit operations for code change description generation. Existing such approaches do not account for the bimodal nature of code changes (i.e., sequence and structure), hence losing the semantics either from the sequential order information or from the semantic logic in the structural abstract syntax trees. With Patcherizer, in order to capture sufficient semantics for code changes, we take advantage of both by fusing *SeqIntention* and *GraphIntention*.

Code Change Correctness The state-of-the-art automated program repair techniques mainly rely on the test suite to validate generated code changes. Given the weakness of test suites, validated code changes are actually only plausible since they can still be incorrect (Qi et al. 2015; Tian et al. 2022a; Gao et al. 2021; Gissurarson et al. 2022; Tian et al. 2020; Ghanbari and Marcus 2022), due to overfitting. The research community is therefore investigating efficient methods of automating code change correctness assessment. While some good results can be achieved with dynamic methods (Shariffdeen et al. 2021), static methods are more scalable. Recently, Tian et al. (2022b) proposed *Panther*, which explored the feasibility of comparing overfitting and correct code changes through representation learning techniques (e.g., CC2Vec Hoang et al. 2020 and Bert Devlin et al. 2018). We show in this work that the representations yielded by Patcherizer can vastly improve the results yielded by Panther compared to its current representation learning approaches.

While recent methods such as CoCoGen (Jacobsen et al. 2025) incorporate surrounding code context to enhance commit understanding, they still operate primarily on surface-level diffs without explicitly modeling the semantic intention behind code changes. Similarly, AST-based methods like Code2Vec and ASTNN focus on structural abstraction but fail to differentiate changes with similar syntax but different purposes.

Moreover, CCRep and CCS2vec (Tian et al. 2022c; Zhang et al. 2023), recent state-of-the-arts in JIT defect prediction, learns commit representations using hand-crafted features and fine-tuned encoders. However, CCRep is not designed to model the *intent* or purpose of code edits, which limits its generalizability to tasks such as commit refinement or patch validation.

8 Conclusion

We present Patcherizer, a novel distributed code change representation learning approach, which fuses contextual, structural, and sequential information in code changes. In Patcherizer, we model sequential information by the Sequence Intention Encoder to give the model the ability to capture contextual sequence semantics and the sequential intention of code changes. In addition, we model structural information by the Graph Intention Encoder to obtain the structural change semantics. Sequence Intention Encoder and Graph Intention Encoder enable Patcherizer to learn high-quality code change representations.

We evaluate Patcherizer on three tasks, and the results demonstrate that it outperforms several baselines, including the state-of-the-art, by substantial margins. An ablation study further highlights the importance of the different design choices. Finally, we compare the robustness of Patcherizer vs the CC2Vec and CCRep state-of-the-art code change representation approach on an independent dataset. The empirical result shows that Patcherizer is more effective.

Acknowledgements This work is supported by the NATURAL project, which has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant No. 949014), the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No. RS-2025-02219277, AI Star Fellowship Support Project(DGIST)), and the DGIST Start-up Fund Program of the Ministry of Science and ICT(2025080011).

Author Contributions – Xunzhu Tang: Conceptualization, Methodology, Writing - Original Draft, Formal Analysis, and Data Curation. – Haoye Tian: Validation, Writing - Review & Editing. – Weiguo Pian: Critical Review. – Saad Ezzini: Critical Review. – Abdoul Kader Kaboré: Critical Review. – Andrew Habib: Writing - Review. – Kisub Kim: Critical Review. – Jacques Klein: Supervision, and Critical Review. – Tegawendé F. Bissyandé: Supervision, Funding Acquisition, and Final Manuscript Approval.

Data Availability We make our code and dataset publicly available at: https://anonymous.4open.science/r/Patcherizer-1E04

Declarations

Conflict of Interest Statement The authors declare that they have no conflict of interest.

Ethical Approval This study does not involve human participants, animals, or other entities requiring ethical oversight. Consequently, no ethical approval was required.

Informed Consent No human participants were involved in this study, and informed consent was therefore not applicable.

References

Allamanis M, Barr ET, Devanbu P, Sutton C (2018) A survey of machine learning for big code and naturalness. ACM Comput Surv (CSUR) 51(4):1–37

Alon U, Sadaka R, Levy O, Yahav E (2020) Structural language models of code. In: Proceedings of the 37th international conference on machine learning, ICML 2020, 13-18 July 2020, Virtual Event, *Proceedings of Machine Learning Research*, vol. 119, pp 245–256. PMLR. http://proceedings.mlr.press/v119/alon20a.html

Alon U, Zilberstein M, Levy O, Yahav E (2019) code2vec: learning distributed representations of code. PACMPL 3(POPL) 40:1–40:29. https://doi.org/10.1145/3290353

Banerjee S, Lavie A (2005) Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In: Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization, pp 65–72

Barr ET, Brun Y, Devanbu P, Harman M, Sarro F (2014) The plastic surgery hypothesis. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, pp 306–317

Brody S, Alon U, Yahav E (2020) A structural model for contextual code changes. Proceed ACM Program Lang 4(OOPSLA), pp 1–28

Buse RP, Weimer WR (2010) Automatically documenting program changes. In: Proceedings of the IEEE/ACM international conference on Automated software engineering, pp 33–42

Cabrera Lozoya R, Baumann A, Sabetta A, Bezzi M (2021) Commit2vec: Learning distributed representations of code changes. SN Comput Sci 2(3):1–16

Ciniselli M, Cooper N, Pascarella L, Poshyvanyk D, Di Penta M, Bavota G (2021) An empirical study on the usage of bert models for code completion. In: 2021 IEEE/ACM 18th international conference on mining software repositories (MSR), IEEE, pp 108–119

Cordella LP, Foggia P, Sansone C, Vento M (1999) Performance evaluation of the vf graph matching algorithm. In: Proceedings 10th international conference on image analysis and processing, IEEE, pp 1172–1177

Cortés-Coy LF, Linares-Vásquez M, Aponte J, Poshyvanyk D (2014) On automatically generating commit messages via summarization of source code changes. In: 2014 IEEE 14th International working conference on source code analysis and manipulation, IEEE, pp 275–284

DeFreez D, Thakur AV, Rubio-González C (2018) Path-based function embedding and its application to error-handling specification mining. In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 423–433

Devlin J, Chang MW, Lee K, Toutanova K (2018) Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805

Dong J, Lou Y, Zhu Q, Sun Z, Li Z, Zhang W, Hao D (2022) Fira: Fine-grained graph-based code change representation for automated commit message generation

Dyer R, Nguyen HA, Rajan H, Nguyen TN (2013) Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In: 2013 35th International conference on software engineering (ICSE), IEEE, pp 422–431

Elnaggar A, Ding W, Jones L, Gibbs T, Feher T, Angerer C, Severini S, Matthes F, Rost B (2021) Codetrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing. arXiv:2104.02443

Falleri JR, Morandat F, Blanc X, Martinez M, Monperrus M (2014) Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pp 313–324. https://doi.org/10.1145/2642937.2642982

Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D et al (2020) Codebert: A pre-trained model for programming and natural languages. arXiv:2002.08155

Gao X, Wang B, Duck GJ, Ji R, Xiong Y, Roychoudhury A (2021) Beyond tests: Program vulnerability repair via crash constraint extraction. ACM Trans Softw Eng Methodol (TOSEM) 30(2):1–27

Ghanbari A, Marcus A (2022) Patch correctness assessment in automated program repair based on the impact of patches on production and test code

Gissurarson MP, Applis L, Panichella A, van Deursen A, Sands D (2022) Propr: property-based automatic program repair. In: Proceedings of the 44th international conference on software engineering, pp 1768–1780

Glorot X, Bengio Y (2010) Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the thirteenth international conference on artificial intelligence and statistics, pp 249–256. JMLR Workshop and Conference Proceedings

Glorot X, Bordes A, Bengio Y (2011) Deep sparse rectifier neural networks. In: Proceedings of the fourteenth international conference on artificial intelligence and statistics, pp 315–323. JMLR Workshop and Conference Proceedings

Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, Tufano M, Deng SK, Clement CB, Drain D, Sundaresan N, Yin J, Jiang D, Zhou M (2021) Graphcodebert: Pre-training code representations with data flow. In: 9th International conference on learning representations, ICLR 2021, Virtual Event, Austria, May 3-7. OpenReview.net. https://openreview.net/forum?id=jLoC4ez43PZ

Henkel J, Lahiri SK, Liblit B, Reps T (2018) Code vectors: Understanding programs through embedded abstracted symbolic traces. In: Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 163–174

He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 770–778

Hindle A, German DM, Holt R (2009) Software process recovery using recovered unified process views. In: 2009 IEEE International conference on software maintenance, IEEE, pp 285–294

Hoang T, Dam HK, Kamei Y, Lo D, Ubayashi N (2019) Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In: 2019 IEEE/ACM 16th international conference on mining software repositories (MSR), IEEE, pp 34–45

Hoang T, Kang HJ, Lo D, Lawall J (2020) Cc2vec: Distributed representations of code changes. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp 518–529

Hossin M, Sulaiman MN (2015) A review on evaluation metrics for data classification evaluations. Int J Data Mining Knowl Manag Process 5(2):1

Huang Y, Jia N, Zhou HJ, Chen XP, Zheng ZB, Tang MD (2020) Learning human-written commit messages to document code changes. J Comput Sci Technol 35(6):1258–1277

Jacobsen C, Zhuang Y, Duraisamy K (2025) Cocogen: Physically consistent and conditioned score-based generative models for forward and inverse problems. SIAM J Sci Comput 47(2):C399–C425

Jiang S, Armaly A, McMillan C (2017) Automatically generating commit messages from diffs using neural machine translation. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering, IEEE, pp 135–146

Jiang N, Lutellier T, Tan L (2021) Cure: Code-aware neural machine translation for automatic program repair. In: 2021 IEEE/ACM 43rd International conference on software engineering (ICSE), IEEE, pp 1161–1173

Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2012) A large-scale empirical study of just-in-time quality assurance. IEEE Trans Softw Eng 39(6):757–773

Kamei Y, Fukushima T, McIntosh S, Yamashita K, Ubayashi N, Hassan AE (2016) Studying just-in-time defect prediction using cross-project models. Empirical Softw Eng 21(5):2072–2106

Kingma DP, Ba J (2014) Adam: A method for stochastic optimization. arXiv:1412.6980

Kipf TN, Welling M (2016) Semi-supervised classification with graph convolutional networks. arXiv:1609.02907

Li M, Miao Z, Zhang XP, Xu W (2021) An attention-seq2seq model based on crnn encoding for automatic labanotation generation from motion capture data. In: ICASSP 2021-2021 IEEE international conference on acoustics, speech and signal processing (ICASSP), IEEE, pp 4185–4189

Lin B, Wang S, Wen M, Mao X (2022) Context-aware code change embedding for better patch correctness assessment. ACM Trans Softw Eng Methodol (TOSEM) 31(3):1–29

Linares-Vásquez M, Cortés-Coy LF, Aponte J, Poshyvanyk D (2015) Changescribe: A tool for automatically generating commit messages. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, IEEE, vol. 2, pp 709–712

Lin B, Wang S, Liu Z, Liu Y, Xia X, Mao X (2023) Cct5: A code-change-oriented pre-trained model. In: Proceedings of the 31st ACM joint European software engineering conference and symposium on the foundations of software engineering, pp 1509–1521

Liu S, Gao C, Chen S, Yiu NL, Liu Y (2020c) Atom: Commit message generation based on abstract syntax tree and hybrid ranking. IEEE Trans Softw Eng

Liu Q, Liu Z, Zhu H, Fan H, Du B, Qian Y (2019) Generating commit messages from diffs using pointer-generator network. In: 2019 IEEE/ACM 16th International conference on mining software repositories (MSR), IEEE, pp 299–309

Liu F, Li G, Wei B, Xia X, Fu Z, Jin Z (2020a) A self-attentional neural architecture for code completion with multi-task learning. In: Proceedings of the 28th international conference on program comprehension, pp 37–47

Liu F, Li G, Zhao Y, Jin Z (2020b) Multi-task learning based pre-trained language model for code completion. In: Proceedings of the 35th IEEE/ACM international conference on automated software engineering, pp 473–485

Liu Z, Tang Z, Xia X, Yang X (2023) Ccrep: Learning code change representations via pre-trained code model and query back. In: 45th IEEE/ACM International conference on software engineering, ICSE 2023, Melbourne, Australia, IEEE, May 14-20, 2023, pp 17–29. https://doi.org/10.1109/ICSE48619.2023.00014

Liu Z, Xia X, Hassan AE, Lo D, Xing Z, Wang X (2018) Neural-machine-translation-based commit message generation: how far are we? In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, pp 373–384

Luo C, Zhan J, Xue X, Wang L, Ren R, Yang Q (2018) Cosine normalization: Using cosine similarity instead of dot product in neural networks. In: International conference on artificial neural networks, Springer, pp 382–391

Networkx (2018) In: Alhajj R, Rokne JG (eds.) Encyclopedia of social network analysis and mining, 2nd Edition. Springer. https://doi.org/10.1007/978-1-4939-7131-2_100771

Nie LY, Gao C, Zhong Z, Lam W, Liu Y, Xu Z (2021) Coregen: Contextualized code representation learning for commit message generation. Neurocomputing 459:97–107

Niemeyer M, Geiger A (2021) Giraffe: Representing scenes as compositional generative neural feature fields. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pp 11453–11464

Papineni K, Roukos S, Ward T, Zhu WJ (2002) Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th annual meeting of the Association for Computational Linguistics, pp 311–318

Pian W, Peng H, Tang X, Sun T, Tian H, Habib A, Klein J, Bissyandé TF (2023) Metatptrans: A meta learning approach for multilingual code representation learning. Proceedings of the AAAI conference on artificial intelligence 37:5239–5247

Pian W, Peng H, Tang X, Sun T, Tian H, Habib A, Klein J, Bissyandé TF (2022) Metatptrans: A meta learning approach for multilingual code representation learning. arXiv:2206.06460

Qi Z, Long F, Achour S, Rinard M (2015) An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Proceedings of the 2015 international symposium on software testing and analysis, pp 24–36

Qin L, Liu T, Che W, Kang B, Zhao S, Liu T (2021) A co-interactive transformer for joint slot filling and intent detection. In: ICASSP 2021-2021 IEEE International conference on acoustics, speech and signal processing (ICASSP), IEEE, pp 8193–8197

Rouge LC (2004) A package for automatic evaluation of summaries. In: Proceedings of workshop on text summarization of ACL, Spain

See A, Liu PJ, Manning CD (2017) Get to the point: Summarization with pointer-generator networks. arXiv:1704.04368

Shariffdeen R, Noller Y, Grunske L, Roychoudhury A (2021) Concolic program repair. In: Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation, pp 390–405

Shaw P, Uszkoreit J, Vaswani, A (2018) Self-attention with relative position representations. In: Walker MA, Ji H, Stent A (eds.) Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1–6, 2018, Volume 2 (Short Papers), pp 464–468. Association for Computational Linguistics. https://doi.org/10.18653/v1/n18-2074

Shi E, Wang Y, Du L, Zhang H, Han S, Zhang D, Sun H (2021) CAST: enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. In: Moens M, Huang X, Specia L, Yih SW (eds.) Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, pp 4053–4062. Association for Computational Linguistics. https://doi.org/10.18653/v1/2021.emnlp-main.332

Svyatkovskiy A, Zhao Y, Fu S, Sundaresan N (2019) Pythia: Ai-assisted code completion system. In: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining, pp 2727–2735

Tang X, Zhu R, Sun T, Wang S (2021) Moto: Enhancing embedding with multiple joint factors for chinese text classification. In: 2020 25th International conference on pattern recognition (ICPR), IEEE, pp 2882–2888

Tao Y, Kim S, Kim M et al (2012) How do software engineers understand code changes? an exploratory study in industry. In: Proceedings of the 20th international symposium on the foundations of software engineering, pp 1–10

Thunes C (2013) javalang: pure python java parser and tools

Tian H, Li Y, Pian W, Kabore AK, Liu K, Habib A, Klein J, Bissyandé TF (2022a) Predicting patch correctness based on the similarity of failing test cases. ACM Trans Softw Eng Methodol

Tian H, Liu K, Kaboré AK, Koyuncu A, Li L, Klein J, Bissyandé TF (2020) Evaluating representation learning of code changes for predicting patch correctness in program repair. In: 2020 35th IEEE/ACM International conference on automated software engineering (ASE), IEEE, pp 981–992

Tian H, Liu K, Li Y, Kaboré AK, Koyuncu A, Habib A, Li L, Wen J, Klein J, Bissyandé TF (2022b) The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches. arXiv:2203.08912

Tian H, Tang X, Habib A, Wang S, Liu K, Xia X, Klein J, Bissyandé TF (2022c) Is this change the answer to that problem? correlating descriptions of bug and code changes for evaluating patch correctness. arXiv:2208.04125

Van der Maaten L, Hinton G (2008) Visualizing data using t-sne. J Mach Learn Res 9(11)

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. Adv Neural Inf Process Syst 30

Vijayakumar AK, Cogswell M, Selvaraju RR, Sun Q, Lee S, Crandall D, Batra D (2016) Diverse beam search: Decoding diverse solutions from neural sequence models. arXiv:1610.02424

Wang K, Singh R, Su Z (2017) Dynamic neural program embedding for program repair. arXiv:1711.07163

Wang S, Tang D, Zhang L, Li H, Han D (2022a) Hienet: Bidirectional hierarchy framework for automated icd coding. In: International conference on database systems for advanced applications, Springer, pp 523–539

Wang S, Tang D, Zhang L, Li H, Han D (2022b) Hienet: Bidirectional hierarchy framework for automated ICD coding. In: Bhattacharya A, Lee J, Li M, Agrawal D, Reddy PK, Mohania MK, Mondal A, Goyal V, Kiran RU (eds.) Database Systems for Advanced Applications - 27th International Conference, DASFAA 2022, Virtual Event, April 11-14, 2022, Proceedings, Part II, *Lecture Notes in Computer Science*, Springer, vol. 13246, pp 523–539. https://doi.org/10.1007/978-3-031-00126-0_38

Wang X, Wang S, Feng P, Sun K, Jajodia S (2021b) Patchdb: A large-scale security patch dataset. In: 2021 51st Annual IEEE/IFIP international conference on dependable systems and networks (DSN), IEEE, pp 149–160

Wang Y, Wang W, Joty SR, Hoi SCH (2021d) Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Moens M, Huang X, Specia L, Yih SW (eds.) Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, pp 8696–8708. Association for Computational Linguistics. https://doi.org/10.18653/v1/2021.emnlp-main.685

Wang Y, Wang W, Joty S, Hoi SC (2021c) Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859

Wang S, Wang X, Sun K, Jajodia S, Wang H, Li Q (2023) Graphspd: Graph-based security patch detection with enriched code semantics. In: 2023 IEEE Symposium on security and privacy (SP), IEEE, pp 2409–2426

Wang H, Xia X, Lo D, He Q, Wang X, Grundy J (2021a) Context-aware retrieval-based deep commit message generation. ACM Trans Softw Eng Methodol (TOSEM) 30(4):1–30

Wang M, Zheng D, Ye Z, Gan Q, Li M, Song X, Zhou J, Ma C, Yu L, Gai Y et al (2019) Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv:1909.01315

Wilcoxon F (1992) Individual comparisons by ranking methods. In: Breakthroughs in statistics: Methodology and distribution, Springer, pp 196–202

Xu S, Yao Y, Xu F, Gu T, Tong H, Lu J (2019) Commit message generation for source code changes. In: IJCAI

Yefet N, Alon U, Yahav E (2020) Adversarial examples for models of code. Proceed ACM Program Lang 4(OOPSLA):1–30

Yin P, Neubig G, Allamanis M, Brockschmidt M, Gaunt AL (2019) Learning to represent edits. In: International conference on learning representations. https://openreview.net/forum?id=BJl6AjC5F7

Zhang F, Chen B, Zhao Y, Peng X (2023) Slice-based code change representation learning. In: 2023 IEEE International conference on software analysis, evolution and reengineering (SANER), IEEE, pp 319–330

Zhang Z, Han X, Liu Z, Jiang X, Sun M, Liu Q (2019c) Ernie: Enhanced language representation with informative entities. arXiv:1905.07129

Zhang Z, Han X, Liu Z, Jiang X, Sun M, Liu Q (2019d) ERNIE: Enhanced language representation with informative entities. In: Proceedings of the 57th annual meeting of the association for computational linguistics, pp 1441–1451. Association for Computational Linguistics, Florence, Italy. https://doi.org/10.18653/v1/P19-1139

Zhang S, Tong H, Xu J, Maciejewski R (2019b) Graph convolutional networks: a comprehensive review. Computat Soc Netw 6(1):1–23

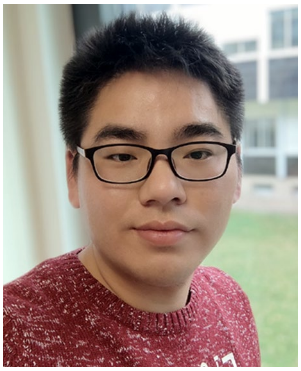
Zhang Y, Wallace B (2015) A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. arXiv:1510.03820

Zhang J, Wang X, Zhang H, Sun H, Wang K, Liu X (2019a) A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE), IEEE, pp 783–794

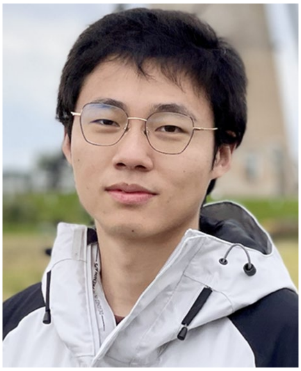
Zhou Y, Siow JK, Wang C, Liu S, Liu Y (2021) Spi: Automated identification of security patches via commits. ACM Trans Softw Eng Methodol (TOSEM) 31(1):1–27

Zhou X, Xu B, Han D, Yang Z, He J, Lo D (2023) Ccbert: Self-supervised code change representation learning. In: 2023 IEEE International conference on software maintenance and evolution (ICSME), IEEE, pp 182–193

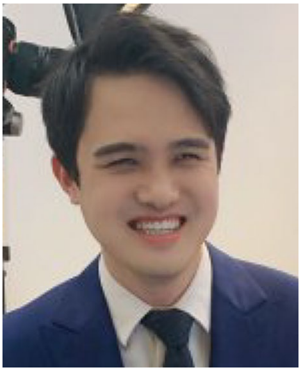
Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Xunzhu Tang is PhD student with the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at the University of Luxembourg. He received his master degree in Computer System and Architecture from Huazhong University of Science and Technology, China in 2021. His research interests include patch explanation, bug finding and fixing.



Haoye Tian is assistant professor in Aalto University. Before that, he worked as Postdoc with Prof. Bach Le. Prior to that, He finished his PhD at the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at the University of Luxembourg. He received his master degree in Software Engineering from Chongqing University, China in 2019. His research interests include automated program repair, patch validation, machine and deep learning.



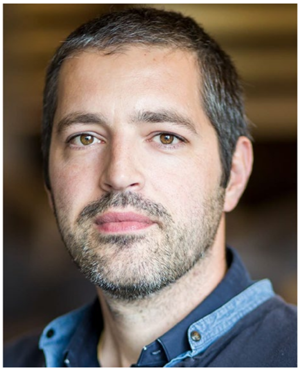
Weiguo Pian is a PhD student with the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at the University of Luxembourg, supervised by Prof. Tegawendé F. Bissyandé. He received his bachelor and master degrees before joining SnT as a doctoral researcher. His research interests span machine learning, computer vision, and software engineering, with a particular focus on continual learning, multimodal representation, and automated software repair. His work has been published in top-tier venues including *ICCV, NeurIPS, AAAI, ASE, FSE, TOSEM, and TSE*.

Saad Ezzini is an assistant professor in King Fahd University of Petroleum and Minerals. He received his PhD in software engineering at the University of Luxembourg in 2022. And in 2017 he received his master's degree in Data Science at USMBA, Morocco. His research interests include, Requirements Engineering, AI for SE, and Natural Language Processing.

Abdoul Kader Kaboré is a Project Officer at the University of Luxembourg. He obtained his PhD from the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at the University of Luxembourg, under the supervision of Prof. Tegawendé F. Bissyandé. His research interests include AI for Software Engineering, Software Security, and Automated Program Repair. He has contributed to several influential works on patch correctness prediction, code change representation, and cross-language code clone detection, with publications in leading venues such as *ICSE, ASE, FSE, TOSEM, and TSE*.

Andrew Habib is a Research Scientist in Industrial AI at the ABB Corporate Research Center in Ladenburg, Germany. Previously, he served as a post-doctoral researcher within the TruX research group at the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at the University of Luxembourg, collaborating with Tegawendé Bissyandé and Jacques Klein. He earned his PhD in Computer Science from the Software Lab (SOLA) at TU Darmstadt under the supervision of Michael Pradel. Andrew holds a double MSc in Engineering, Security, and Mobile Computing from the Technical University of Denmark (DTU) and the Norwegian University of Science and Technology (NTNU) under the NordSecMob program, and a double BSc in Computer Science and Mathematics from the American University in Cairo, with an exchange semester at Portland State University.

Kisub Kim is an Assistant Professor at DGIST (Daegu Gyeongbuk Institute of Science and Technology), South Korea. He received his PhD in Computer Science from the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at the University of Luxembourg in 2021. His research interests include AI for Software Engineering, with a focus on code search, bug localization, program repair, and deep learning for software engineering tasks. He has published in leading venues such as *TSE, ICSE, ASE, and FSE*.



Jacques Klein is a researcher and professor in software engineering and software security who develops innovative approaches and tools towards helping the research and practice communities build trustworthy software. He is a member of the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at the University of Luxembourg. He received a Ph.D. degree in Computer Science from the University of Rennes, France, in 2006. His main areas of expertise are threefold: (1) Software Security (Malware detection, prevention and dissection, Static Analysis for Security, Vulnerability Detection, etc.); (2) Software Reliability (Software Testing, Semi-Automated and Fully-Automated Program Repair, etc.); (3) Data Analytics (Multi-objective reasoning and optimization, Model-driven data analytic, Time Series Pattern Recognition, etc.).

Tegawendé F. Bissyandé is research scientist with the Interdisciplinary Center for Security, Reliability and Trust at the University of Luxembourg. He holds a PhD in computer from the Université de Bordeaux in 2013, and an engineering degree (MSc) from ENSEIRB. His research interests are in debugging, including bug localization and program repair, as well as code search, including code clone detection and code classification. He has published research results in all major venues in Software engineering (ICSE, ESEC/FSE, ASE, ISSTA, EMSE, TSE). His research is supported by FNR (Luxembourg National Research Fund). Dr. Bissyandé is the PI of the CORE RECOMMEND project on program repair, under which the current work has been performed.

Authors and Affiliations

Xunzhu Tang[1] · Haoye Tian[2] · Weiguo Pian[1] · Saad Ezzini[3] · Abdoul Kader Kaboré[1] · Andrew Habib[1] · Kisub Kim[4] · Jacques Klein[1] · Tegawendé F. Bissyandé[1]

✉ Kisub Kim
kisub.kim@dgist.ac.kr

Xunzhu Tang
xunzhu.tang@uni.lu

Haoye Tian
tianhaoyemail@gmail.com

Weiguo Pian
weiguo.pian@uni.lu

Saad Ezzini
s.ezzini@lancaster.ac.uk

Abdoul Kader Kaboré
abdoulkader.kabore@uni.lu

Andrew Habib
andrew.a.habib@gmail.com

Jacques Klein
jacques.klein@uni.lu

Tegawendé F. Bissyandé
tegawende.bissyande@uni.lu

[1] SnT, University of Luxembourg, Luxembourg City, Luxembourg

[2] School of Computing and Information Systems, University of Melbourne, Melbourne, Australia

[3] School of Computing and Communications, Lancaster University, Lancaster, UK

[4] DGIST, Daegu, Republic of Korea