# CoEdPilot: Recommending Code Edits with Learned Prior Edit Relevance, Project-wise Awareness, and Interactive Nature

Chenyan Liu*
Shanghai Jiao Tong University
Shanghai, China
National University of Singapore
Singapore, Singapore
chenyan@u.nus.edu

Yufan Cai*
Shanghai Jiao Tong University
Shanghai, China
National University of Singapore
Singapore, Singapore
cai_yufan@u.nus.edu

Yun Lin†
Shanghai Jiao Tong University
Shanghai, China
lin_yun@sjtu.edu.cn

Yuhuan Huang
Shanghai Jiao Tong University
Shanghai, China
hyh0u0@sjtu.edu.cn

Yunrui Pei
Shanghai Jiao Tong University
Shanghai, China
yunruipei@sjtu.edu.cn

Bo Jiang
Bytedance Network Technology
Beijing, China
jiangbo.jacob@bytedance.com

Ping Yang
Bytedance Network Technology
Beijing, China
yangping.cser@bytedance.com

Jin Song Dong
National University of Singapore
Singapore, Singapore
dcsdjs@nus.edu.sg

Hong Mei
Shanghai Jiao Tong University
Shanghai, China
meih@pku.edu.cn

## ABSTRACT

Recent years have seen the development of LLM-based code generation. Compared to generating code in a software project, incremental code edits are empirically observed to be more frequent. The emerging code editing approaches usually formulate the problem as generating an edit based on *known* relevant prior edits and context. However, practical code edits can be more complicated. First, an editing session can include multiple (ir)relevant edits to the code under edit. Second, the inference of the subsequent edits is non-trivial as the scope of its ripple effect can be the whole project.

In this work, we propose CoEdPilot, an LLM-driven solution to recommend code edits by discriminating the relevant edits, exploring their interactive natures, and estimating its ripple effect in the project. Specifically, CoEdPilot orchestrates multiple neural transformers to identify *what* and *how* to edit in the project regarding both edit location and edit content. When a user accomplishes an edit with an optional editing description, an *Subsequent Edit Analysis* first reports the most relevant files in the project with what types of edits (e.g., *keep*, *insert*, and *replace*) can happen for each line of their code. Next, an *Edit-content Generator* generates concrete edit options for the lines of code, regarding its relevant prior changes reported by an *Edit-dependency Analyzer*. Last, both the *Subsequent Edit Analysis* and the *Edit-content Generator* capture relevant prior edits as feedback to readjust their recommendations. We train our models by collecting over 180K commits from 471 open-source projects in 5 programming languages. Our extensive experiments show that (1) CoEdPilot can well predict the edits (i.e., predicting edit location with accuracy of 70.8%-85.3%, and the edit content with exact match rate of 41.8% and BLEU4 score of 60.7); (2) CoEdPilot can well boost existing edit generators such as GRACE and CCT5 on exact match rate by 8.57% points and BLEU4 score by 18.08. Last, our user study on 18 participants with 3 editing tasks (1) shows that CoEdPilot can be effective in assisting users to edit code in comparison with Copilot, and (2) sheds light on the future improvement of the tool design. The video demonstration of our tool is available at https://sites.google.com/view/coedpilot/home.

## CCS CONCEPTS

• **Computing methodologies → Natural language processing**; • **Software and its engineering → Automatic programming**; **Software evolution**.

## KEYWORDS

code edit generation, edit location, interaction, language model

**ACM Reference Format:**
Chenyan Liu, Yufan Cai, Yun Lin, Yuhuan Huang, Yunrui Pei, Bo Jiang, Ping Yang, Jin Song Dong, and Hong Mei. 2024. CoEdPilot: Recommending Code Edits with Learned Prior Edit Relevance, Project-wise Awareness, and Interactive Nature. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3652142

## 1 INTRODUCTION

Recent years have seen the success of the application of LM (Language Model) in code generation tasks. LM-based approaches, such as CodeBERT [19], GraphCodeBERT [24], CodeT5 [54], Copilot

[*] Both authors contributed equally to the paper
[†] Corresponding author

*ISSTA '24, September 16–20, 2024, Vienna, Austria*
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3652142

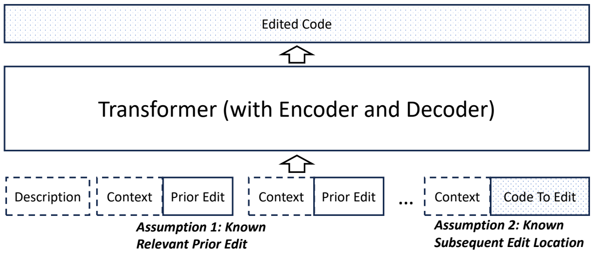

Edited Code
Transformer (with Encoder and Decoder)
Description
Context
Prior Edit
Context
Prior Edit
...
Context
Code To Edit
Assumption 1: Known Relevant Prior Edit
Assumption 2: Known Subsequent Edit Location

**Figure 1: State-of-the-art Code Editing Framework [15] [25] [33]. The dotted rectangles represent the code before and after the recommended edits.**

[23], and ChatGPT [44], dominate the code generation solutions by translating users' description and surrounding code context to new code. Nevertheless, compared to generating new code, empirical observation shows that the activities of editing existing code happen more frequently [31, 32, 40]. Empirical study on the commits in a large number of open-source projects shows that editing behaviors take about 70% in the commit history [41].

Many transformer-based approaches are proposed to generalize the code generation solutions to code editing solutions, such as GRACE [25], CCT5 [37], CoditT5 [58], and MODIT [15]. While those approaches are different in representing the edits in deep learning models, they formulate the edit generation problem as a translation problem by (1) taking the input as *known* relevant prior edits (and their context) and code region (and their context) where the change is *known* to happen and (2) generating the output as a piece of edited code. We show a model architecture as Figure 1 to capture the general idea of the state-of-the-art solutions, where optional edit description, prior edits and their optional context, and the code under the edit are fed to a language model to output a piece of edited code.

While those solutions have laid an important foundation for code editing tasks, there is still a gap between the solutions and the practical scenarios.

- **Assumption of Relevance of Prior Edits.** In an editing session, existing work usually assumes that all the prior edits of a target edit are relevant. However, it might not be true in practice (see Section 2 for more details). Feeding the model with irrelevant prior edits can introduce noisy input, compromising the accuracy of the generated edits.
- **Assumption of Availability of Subsequent Edit Location.** In addition, it is also non-trivial to know where the edits can happen because the ripple effect of a prior edit may propagate across the whole project [50].
- **Interactive Nature between Multiple Edits.** Lastly, code edits can interact with each other regarding their syntactic dependency and semantic relevance. However, existing transformers still lack of design to capture such interaction.

In this work, we propose, CoEdPilot, an LM-based solution to address the above concerns. We designed CoEdPilot to monitor the ripple effect of an edit as to where the subsequent edits can happen, infer the most relevant prior edits, and capture the edit interaction more explicitly. To this end, we design CoEdPilot by orchestrating a set of neural transformers [52] to coherently work with each other. Once an edit-triggering event happens (e.g., an edit $e$ happens with an optional edit description $prp$), the following components are activated in an order:

- *Two-staged edit location:* In the first stage, we scan the whole project with an *Edit-propagating File Locator*, which reports a set of files $\mathcal{F}$ where the changes can happen in a coarse-grained way. In the second stage, with the reported files $\mathcal{F}$, we apply a sliding window on those files with our *Edit-propagating Line Locator* to report the type of edit (e.g., *keep*, *insert*, and *replace*) for each line of code in the files. As a result, we can have a set of lines of code with labelled type of edit, denoted as $\mathcal{L}_e = \{l_e = (l,t) | l \in \mathcal{L}, t \in \{insert, replace\}\}$, where $\mathcal{L}$ is the set of lines of code in the project. $\mathcal{L}_e$ includes all the lines of code predicted to be inserted with or replaced with new content.
- *Edit content generation:* With the reported editing locations $\mathcal{L}_t$, we use our trained *Edit-content Generator* to further generate the edit content for each location with prediction $e_t = (l,t)$, regarding the editing description $prp$ and a set of selected relevant prior edits. Specifically, we select a set of relevant prior edits $\mathcal{P} = \{e = (l,t,c_a,c_b)\}$ to generate a list of edit options, where $l$ indicates the editing line of code, $t$ indicates the edit type, $c_a$ indicates the code content after the edit, and $c_b$ indicates the code content before the edit. Note that, $c_a$ and $c_b$ further incorporate user feedback on the code under the edit, allowing us to adapt the user's intention on-the-fly in the editing session.
- *Edit-dependency analyzer:* For selecting the relevant prior edits, we train an *Edit-dependency Analyzer* to parse all the prior edits and select the most syntactically and semantically relevant ones for generating the target edit.

Once a new edit $e'$ is accepted, it serves as a new edit-triggering event to activate the above procedures.

We train our neural models from the collected over 180K commits from 471 open source projects in 5 programming languages. We evaluate our models with extensive experiments. Our extensive experiment shows that (1) CoEdPilot can identify edit locations with an accuracy of 70.8-85.3%; and (2) for each identified edit location, CoEdPilot achieves the exact match rate of 41.8% and the BLEU score of 60.7 for the top-1 recommendation. Our ablation study shows that CoEdPilot, as a code-editing framework, can improve the exact match rate and BLEU score of state-of-the-art edit generators such as GRACE and CoditT5 by on average 8.57% and 18.08 respectively. Further, our user study on 18 participants with 3 editing tasks on feature enhancement, refactoring, and bug fixing shows that (1) in comparison to our baseline Copilot, CoEdPilot can be effective in assisting users to edit code by its advantage on the project-wise awareness and the capture of the interaction between relevant edits, and (2) sheds light on the future improvement of the tool design such as distribution-shifting edits from the training dataset.

Overall, we summarize our contributions as follows:

- We propose CoEdPilot, an LM-driven solution to make the state-of-the-art edit generation models more practical by predicting the relevant prior edits, subsequent edit location, and the interactive nature between the edits.
- We design CoEdPilot as a modularized framework, which allows us to plug into any edit-content generators in the community.

**Table 1: The code edits in src/testing/benchmark.go**

| Hunk | Before Edit | After Edit |
|---|---|---|
| H1 (insert) | type benchContext struct {<br>maxLen int // The largest recorded benchmark name.<br>} | type benchContext struct {<br>match *matcher<br>maxLen int // The largest recorded benchmark name.<br>} |
| H2 (insert) | func runBenchmarksInternal(...) bool {<br>// ... other code ...<br>ctx := &benchContext{<br>extLen: len(benchmarkName("", maxprocs)),<br>}<br>// ... other code ...<br>} | func runBenchmarksInternal(...) bool {<br>// ... other code ...<br>ctx := &benchContext{<br>match: newMatcher(matchString, *matchBenchmarks, "-test.bench"),<br>extLen: len(benchmarkName("", maxprocs)),<br>}<br>// ... other code ...<br>} |
| H3 (replace) | func (b *B) runBench(...) bool {<br>// ... other code ...<br>if b.level > 0 {<br>name = b.name + "/" + name<br>}<br>// ... other code ...<br>} | func (b *B) runBench(...) bool {<br>// ... other code ...<br>benchName, ok := b.name, true<br>if b.context != nil {<br>benchName, ok = b.context.match.fullName(&b.common, name)<br>}<br>if !ok {<br>return true<br>}<br>// ... other code ...<br>} |

**Table 2: The code edits in src/testing/testing.go**

| Hunk | Before Edit | After Edit |
|---|---|---|
| H4 (insert) | type testContext struct {<br>mu sync.Mutex<br>// ... other code ...<br>} | type testContext struct {<br>match *matcher<br>mu sync.Mutex<br>// ... other code ...<br>} |
| H5 (replace) | func (t *T) run(...) bool {<br>testName := name<br>if t.level > 0 {<br>testName = t.name + "/" + name<br>}<br>// ... other code ...<br>} | func (t *T) run(...) bool {<br>testName, ok := t.context.match.fullName(&t.common, name)<br>if !ok {<br>return true<br>}<br>// ... other code ...<br>} |
| H6 (replace) | func newTestContext(maxParallel int) *testContext {<br>return &testContext{<br>startParallel: make(chan bool),<br>maxParallel: maxParallel,<br>running: 1, // Set the count to 1 for the main (sequential) test.<br>}<br>} | func newTestContext(maxParallel int, m *matcher) *testContext {<br>return &testContext{<br>match: m,<br>startParallel: make(chan bool),<br>maxParallel: maxParallel,<br>running: 1, // Set the count to 1 for the main (sequential) test.<br>}<br>} |

- We implement an open-source CoEdPilot as a VS Code plugin, which adopts a cloud infrastructure and allows the programmers to try in practice with convenience.
- We conduct extensive experiments (simulation, model-wise evaluation, and user study) showing the effectiveness of individual models as independent model design, model interaction as a whole system, and UI design as a tool.

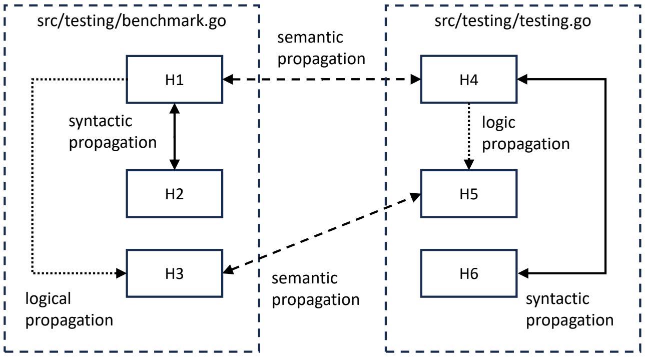


**Figure 2: The type of edit propagation in the code-editing example showed in Table 1 and Table 2.**

Given the space limit, the tool video demonstration, experimental details, and further discussion are available at [6].

## 2 MOTIVATING EXAMPLE

Table 1 and Table 2 (implemented by Go programming language) shows a simplified code-editing example from the commit `00a2` under the project `golang/go`[1]. We summarize the programmer's editing intention in such a commit as follows.

**Original Design.** The function under edit is to update the way to select test cases and benchmark in the `golang/go` project. The Go project is delivered with the `testing` package where a set of test cases are used to check the performance on a set of benchmarks of Go programs. The source file `src/testing/testing.go` automates the testing of the project by selecting a subset of required test cases, and the source file `src/testing/benchmark.go` selects a subset of the benchmark of Go programs such as runtime overhead, memory allocation, lock performance, etc. The old implementation of selecting test cases and benchmarks is by keyword-based matching the name of benchmark and test suites with a string (see the hunk of *Before Edit* of H3 in Table 1 and H5 in Table 2).

**Editing Intention.** In the editing session, the programmer intended to introduce a regular expression matcher to select the required benchmark and test cases.

**Editing Implementation.** To this end, the programmer edits the `benchmark.go` and `testing.go` files as follows:

- **H1** (see Table 1): Introduce a variable `matcher` with pointer type `*matcher` in the type `benchContext`;
- **H2** (see Table 1): Introduce a parameter of type `matcher` when initializing a reference of `benchContext`;
- **H3** (see Table 1): Replace the keyword-based matching implementation with regular-expression-based matching implementation;
- **H4** (see Table 2): Introduce a variable `matcher` with pointer type `*matcher` in the type `testContext`;
- **H5** (see Table 2): Replace the keyword-based matching implementation with regular-expression-based matcher implementation;
- **H6** (see Table 2): Introduce a parameter of type `matcher` when initializing a reference of `testContext`;

While each edit is a simple operation, they are interactive and relevant as different types of edit propagation as shown in Figure 2.

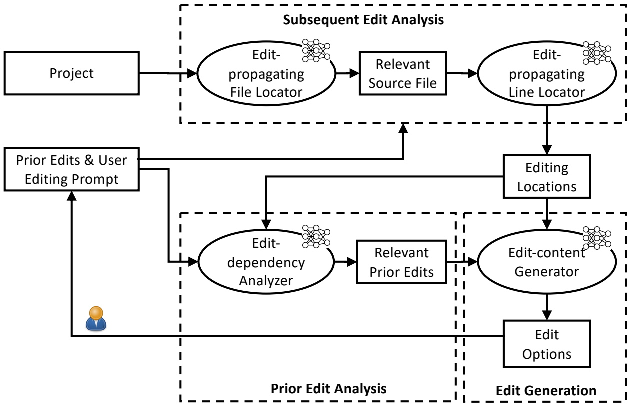


**Figure 3: Overview of CoEdPilot, consisting of subsequent edit analysis, edit generation, and prior edit analysis. The analysis is triggered once an edit-trigger event happens. CoEdPilot orchestrates a set of neural-transformer-based components to accomplish the editing task**

Following the notation in Table 1 and Table 2, we use $H_i$ ($i$ = 1, ..., 6) to indicate the hunk in the code example.

- **Syntactic Propagation:** Syntactic propagation indicates that an edit $e_i$ incurs a compilation error in the project, which further mandates another edit $e_j$ to fix the error. For example, hunk H1 happens so as to cause a compilation error on the location of hunk H2 for missing an initialized parameter. In Figure 2, the fact that H1 and H2 point to each other indicates that the edit propagation caused by program syntax is mutual.
- **Semantic Propagation:** Semantic propagation indicates that an edit $e_i$ is propagated to $e_j$ because $e_i$ and $e_j$ are applied to similar functionalities. In Figure 2, for the example of the editing pair (H1, H4) and (H3, H5), an edit can propagate to the other edit in the pair.
- **Logical Propagation:** Logical propagation indicates that an edit $e_i$ lays a foundation for another edit $e_j$ to accomplish a task. In Figure 2, H1 does not necessarily cause a compilation error at H3, however, H1 introduces a variable `matcher` so that the matching implementation is updated at H3.

Thus, we can see that (1) the edits are interactive with each other in a different way, (2) only a limited number of prior edits is relevant and informative to contribute to an edit, and (3) the edit can propagate to any possible files in the project. However, despite that the existing state-of-the-art solutions such as GRACE [25], CCT5 [37], MODIT [15] and CoditT5 [58] lay an important foundation (see the summary their model architecture in Figure 1), they are still far from accomplishing the edit recommendation tasks in the aforementioned practice.

## 3 APPROACH

Figure 3 shows an overview of our CoEdPilot design, which takes a set of prior edits and an optional edit prompt, and generates the output as a list of subsequent editing locations and their editing options. Overall, the CoEdPilot architecture consists of subsequent edit analysis, prior edit analysis, and edit generation.

[1] The address can be referred in https://github.com/golang/go

- **Subsequent Edit Analysis** takes a set of selected prior code edits and an optional editing prompt to estimate the subsequent edits in the project. In this work, we adopt a two-stage estimation. The first stage estimates the relevant source files, with *Edit-propagating File Locator*, for where the subsequent edits can happen in a coarse-grained (and lighted) way. The second stage further applies a fine-grained detector (i.e., *Edit-propagating Line Locator*) to predict the editing type of each line of code in those files.
- **Prior Edit Analysis** takes the editing locations and selects the most relevant prior edits with *Edit-dependency Analyzer*, regarding their potential of syntactic, semantic, and logical edit propagation to a target edit location.
- **Edit Generation** generates the concrete edit options for each edit location with predicted editing type of *insert* and *replace*, regarding the selected prior edits. Note that, once the user confirms a recommended edit option by (1) directly accepting our recommendation, (2) modifying based on our recommendation, or (3) input his or her own edit, it will be included as a new prior edit. Further, the newly applied edit serves as a new edit-triggering event to launch a new round of editing recommendations.

## 3.1 Subsequent Edit Analysis

**Problem Statement.** We consider the problem of finding the subsequent edits with an edit and its optional user prompt as a problem of edit propagation. Thus, we rephrase the problem as follows. Given a project be a set of files $P$, the user's editing prompt be $prp$, the latest edit $e = (c_b, c_a)$ where $c_b$ is the code before edit, and $c_a$ is the code after edit, we aim to locate a subset of files $F \subset P$, where each $f \in F$ specifies the subsequent edits by attaching each line of code with an editing type as *keep*, *insert*, or *replace*.

**Challenge.** As mentioned above, the edits can interact with each other regarding the syntactic dependency and semantic relevance. As for analyzing syntactic dependencies, we usually need to parse the whole compilable project to build the program dependency graph [20] to track the data, control, and call dependencies. However, the graph construction for large projects could be time-consuming. Further, the implementation of syntactic graph construction [8, 51] and semantic relevance [7, 18, 30] are usually language-dependent. Therefore, we use the neural models for estimating both the syntactic dependency and semantic relevance between two pieces of source code in a more runtime-efficient and language-independent way.

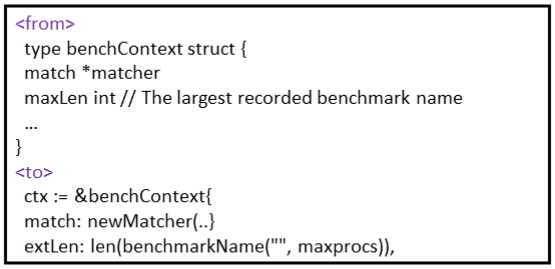
In this work, we adopt a two-stage localization solution, i.e., file localization in a coarse-grained way and line of code localization in a fine-grained way.

### 3.1.1 Propagation File Localization.

Technically, we select a subset $F' \subset F$ where $F' = \{f | sub_{edt}(f, e) > th_{sub}, f \in F\}$, where $sub_{edt}(.,.)$ is a likelihood estimation function for the file $f$ which can be co-edited given the input edit $e$. Further, $th_{sub}$ is a threshold to estimate its likelihood.

We estimate the propagation likelihood regarding two factors, i.e., (1) the estimated dependency of the file $f$ on $e$, and (2) the semantic similarity between some code in $f$ and $e$. Namely, we design Equation 1 as follows.

$$sub_{edt}(f, e) = \alpha_1 \cdot dep(e, f) + \alpha_2 \cdot sem(e, f) + \epsilon \tag{1}$$



```
<from>
 type benchContext struct {
 match *matcher
 maxLen int // The largest recorded benchmark name
 ...
}
<to>
 ctx := &benchContext{
 match: newMatcher(..}
 extLen: len(benchmarkName("", maxprocs)),
```

**Figure 4: An example of input of our transformer for learning the dependency.**

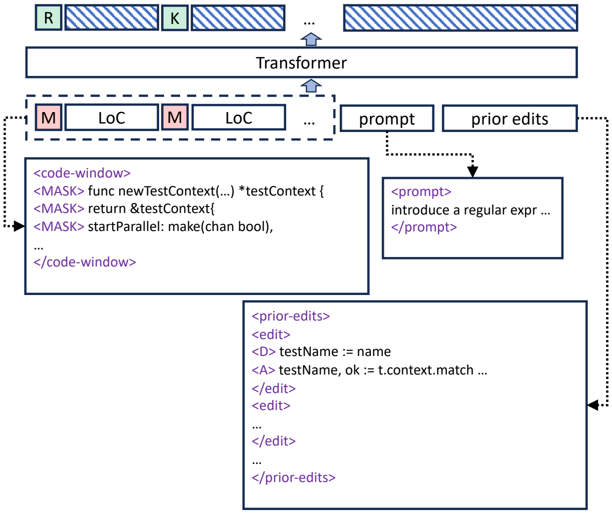
In Equation 1, we let each coefficient $\alpha_i > 0$. We quantize each factor (estimated dependency $dep(e, f)$ and semantic similarity $sem(e, f)$) as a score between 0 and 1 as follows.

*Estimated Dependency.* Given an edit $e = (c_b, c_a)$ and a source file $f$, we develop a dependency inference function $dep(e, f)$ to quantize the likelihood that $f$ depends on $e$. Technically, we use a transformer (e.g., CodeT5 and CodeBERT) as our base model to learn the dependency between the source code. We follow the design of GRACE [25] by constructing the input of a transformer-based language model as shown in Figure 4. Specifically, we use the tags `<from>` and `<to>` as the separator between two pieces of source code. Those tags play a role as instruction tuning. Then we add one dense layer to have two output neurons activated with sigmoid function, i.e., (1) the former code depends on the latter code and (2) the latter code depends on the former code. Given a pair of source code $c_1$, $c_2$, their labelled dependencies are $y_1$ and $y_2$ ($y_1 = 1$ or 0 is for whether $c_1$ depends on $c_2$ and $y_2 = 1$ or 0 is for whether $c_2$ depends $c_1$), and their estimated dependency are $\hat{y_1}$ and $\hat{y_2}$, we design the loss function as Equation 2:

$$\begin{aligned} loss(c_1, c_2) = -(y_1 \times log(\hat{y_1}) + (1 - y_1) \times log(1 - \hat{y_1}) + \\ y_2 \times log(\hat{y_2}) + (1 - y_2) \times log(1 - \hat{y_2})) \end{aligned} \tag{2}$$

In this work, we use Jin et al.'s dependency analyzer [28, 29] to construct the training dataset. Limited by the input length, we split a file $f$ into $k$ smaller segments as $seg_1, .., seg_k$. Further, we choose $c_b$ (the code before the edit) of the latest edit as the target code $c_{tar}$. Then we estimate the likelihood of the dependency between $c_{tar}$ and each code segment. For convenience, we use the symbol of the second output neuron $\hat{y_2}$ to denote the likelihood of the latter code to depend on the former code, $dep(e, f) = max(\hat{y_2}(c_{tar}, seg_i))$. That is, we adopt one-directional dependency to infer the edit propagation. Further, we select the $max(.)$ function as we favour the recall over the precision in this stage. By replacing the analyzer tool [28, 29] with a neural network, we reduce the runtime overhead of analyzing a pair of code snippets from ~70 seconds to ~0.01 second.

*Semantic Similarity and Prompt Relevance.* We capture the semantic similarity of code-to-code by neural embedding in a universal way. The rationale is that we believe a pretrained neural network such as CodeT5 and CodeBERT can capture both the syntactic and semantic similarity. Therefore, still considering the limit of input length of a transformer, we split a source file $f$ into $k$ segments as $seg_1, ..., seg_k$, $c_{tar} = c_b$ where $c_b$ is the code before the edit, and $emd(.)$ as the representation of a piece of code or a prompt extracted



```
<code-window>
<MASK> func newTestContext(...) *testContext {
<MASK> return &testContext{
<MASK> startParallel: make(chan bool),
...
</code-window>
```

```
<prompt>
introduce a regular expr ...
</prompt>
```

```
<prior-edits>
<edit>
<D> testName := name
<A> testName, ok := t.context.match ...
</edit>
<edit>
...
</edit>
...
</prior-edits>
```

**Figure 5: Overview of fine-grained edit location architecture. We formulate the edit location problem as a MLM (Mask Language Modelling) problem to predict the edit type of each LoC (Line of Code).**

from the transformer, we can have:

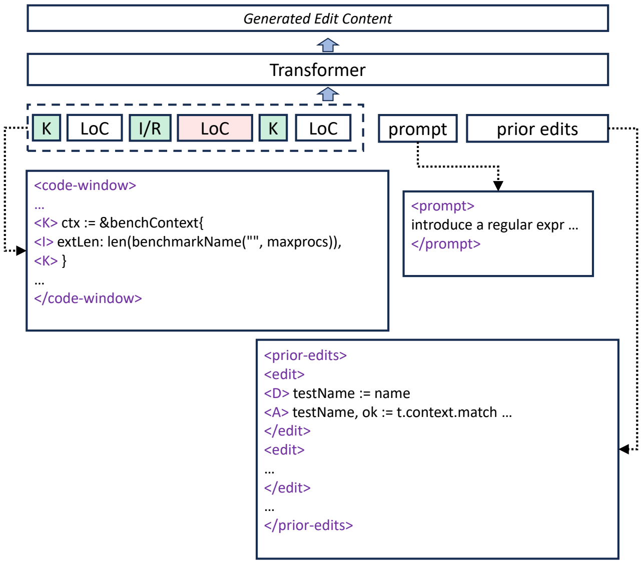
$$sem(e, f) = max(cos(emd(c_{tar}), emd(seg_i))) \quad (3)$$

By this means, with given hyperparameters $\alpha_1$, $\alpha_2$, $\epsilon$, and $th_{sub}$, we have a set of reported source files in a coarse-grained way. These coefficients, intercept and thresholds are available at [6].

*3.1.2 Propagation Line Localization.* Given the located source files with the propagation potential, we apply a sliding window across each file to identify the editing type of each line of source code. As shown in Figure 5, we fine-tune a base transformer model as a MLM (Mask Language Modeling) [17] problem by instruction tuning [47]. Overall, the input of the transformer consists of the target code inside the window, the user prompt, and the relevant prior edits (see more details in Section 3.2). For each input component, we introduce instructions (or tags) such as `code-window`, `prompt`, `prior-edits` and `edit` as the separators for the model to learn the input structure. For each line of the code, we additionally introduce an operator as follows:

- *keep*: the operator type indicates that a line should not be changed, symbolled as `<K>`.
- *insert*: the operator type indicates that there shall be some code inserted after the line, symbolled as `<I>`.
- *replace*: the operator type indicates that the line should be replaced by either an empty line (i.e., delete) or a few different lines (update), symbolled as `<R>`.

These edit operators are masked with a special token `<MASK>` in input. Therefore, we apply MLM task on the operators to train the model to recover them. The prompt is collected from the commit message from the code commit histories. Further, we introduce the details of selecting the prior edits in Section 3.2 and Section 3.4.



```
<code-window>
...
<K> ctx := &benchContext{
<I> extLen: len(benchmarkName("", maxprocs)),
<K> }
...
</code-window>
```

```
<prompt>
introduce a regular expr ...
</prompt>
```

```
<prior-edits>
<edit>
<D> testName := name
<A> testName, ok := t.context.match ...
</edit>
<edit>
...
</edit>
...
</prior-edits>
```

**Figure 6: Overview of edit generator, which generates the edit content for a fine-grained edit location.**

## 3.2 Prior Edit Analysis

**Problem Statement.** Given a set of prior edits $E_p = \{e_{p_1}, ..., e_{p_k}\}$ where $e_i = (c_{b_i}, c_{a_i})$, and the target code $c_{b_{tar}}$, we quantize the likelihood of the influence of $e_{p_i}$ to $c_{b_{tar}}$ between 0 and 1. Specifically, we denote the estimation function as $rel(.,.) : E_p \times C \longrightarrow (0,1)$, where $C$ is the set of pieces of code, i.e., $rel(e_i, c_{b_{tar}}) \in (0,1)$.

We quantize the relevance of prior edits by their syntactic dependency and semantic similarity by Equation 4:

$$\begin{aligned} rel(e_{p_i}, c_{b_{tar}}) = FCN(&dep(e_{p_i}, c_{b_{tar}}), sem(e_{p_i}, c_{b_{tar}}), \\ &loc_{sim}(e_{p_i}, c_{b_{tar}})) \end{aligned} \quad (4)$$

Further, in Equation 4, $FCN$ is a multi-layer fully connected network, the dependency estimation function $dep(.,.)$ for estimating the dependency from $c_{c_{tar}}$ to the code before the edit of $e_{p_i}$ and the semantic relevance function $sem(.,.)$ is defined in Section 3.1.1. Function $loc_{sim}$ evaluates the proximity between $e_{p_i}$ and $c_{b_{tar}}$ as:

$$loc_{sim}(e_{p_i}, c_{b_{tar}}) = \begin{cases} 1 - \frac{|loc(e_{p_i}) - loc(c_{b_{tar}})|}{k} & \text{if } ld(e_{p_i}, c_{b_{tar}}) < k \\ 0 & otherwise \end{cases} \quad (5)$$

In Equation 5, we use a sliding window of size $k$ to define whether the location difference of $e_{p_i}$ and $c_{b_{tar}}$ is small (i.e., $ld(e_{p_i}, c_{b_{tar}}) < k$). If it is, we estimate the proximity as Equation 5. Otherwise, the function $loc_{sim}(.,.)$ is 0. Finally, we define a threshold $th_{pri}$ to identify the set of relevant prior edits $E_{rel} = \{e_p | rel(e_p, c_{b_{tar}}) > th_{pri}\}$.

## 3.3 Edit Generation

Figure 6 shows the overall model architecture to generate edit content on *one* edit location and selected prior edits. Similar to the design of the location of edit lines, the edit generation model takes as input a code-window under the edit, the user's prompt, and relevant prior edits. The prompt and the prior edits share similar tags for the model to capture the structure.

In contrast, the code window describes a *hunk* consists of consecutive lines of the same edit type (*replace* and *insert*) with a few lines of type *keep* as its context. Specifically, each line is attached with a tag `<K>` for the edit type of *keep*; and with a tag `<I>` or `<R>` for the edit type of *insert* and *replace* respectively. Further, the output predicts the edit content of the edit location. We train the transformer with classical cross-entropy loss [1]. On the runtime, we use Beam Search [21] to generate $k$ edit options ranked with their confidence. Last but not least, the user can either accept or modify upon our recommended edits, the new edit will be kept as a new prior edit as user feedback, to further facilitate the whole editing session.

## 3.4 Model Training

Overall, we have the following neural models to train, i.e., an *Edit-dependency Analyzer* (see Section 3.1.1), an *Edit-propagating Line Locator* (see Section 3.1.2), and an *Edit-content Generator* (see Section 3.3).

We first train the *Edit-dependency Analyzer* by collecting the dependency of source code by running Jin et al.'s dependency analysis tool [28, 29] on the open-source projects. Note that, our neural dependency analyzer is expected to predict the dependency between arbitrary two pieces of code without the awareness of their programming language. Then, we train the *Edit-propagating Line Locator* and an *Edit-content Generator* in an interactive manner. We collect the commits from the open source projects as the training dataset (see Section 5). For each commit, we take one hunk as an individual edit, then we train our models by estimating the random order of both intra-file edits and inter-file edits. The rationale is that we do not know the sequence of files being edited and that of the edited locations within a file. Therefore, we do not make any editing partial order assumption on those edits.

Further, given a set of prior edits, we normalize their relevance into a probability distribution $X$. For example, assume that we have three prior edits with the relevance to an edit and a prompt (i.e., the editing description) as 0.7, 0.3, 0.6, then we normalize their relevance to $X = \{\frac{0.7}{0.7+0.3+0.6}, \frac{0.3}{0.7+0.3+0.6}, \frac{0.6}{0.7+0.3+0.6}\} = \{0.437, 0.187, 0.375\}$ to sample the prior edits during the training.

# 4 TOOL DESIGN

Figure 7 shows a screenshot of our CoEdPilot tool as a Visual Studio Code extension [2], which consists of functions designed according to our approach (see Figure 3). We introduce the basic functions and GUI (see Figure 7) as follows. A detailed video is available at [6].

- **Triggering the Edit Recommendation**: When the users edit the code, they can trigger the edit recommendation with a shortcut (or right-click the editor) to request edit locations. Then, an *Edit Description Input* ① will be shown for them to input their optional description of the edit.
- **Subsequent Edit Recommendation**: Then, CoEdPilot shows an *Edit Location View* ② where the edit locations are organized in terms of edit files as their parent nodes and edit lines as their child nodes. The users can click a child node to highlight the corresponding location in the code editor, where a line with edit type of *insert* is in green and a line with that of *replace* is in red.
- **Edit Option Recommendation**: Next, the users can further request the edit option in each edit location, as shown in *Editable Difference View* ③ in Figure 7 where how the code before and after the edit is simulated. Users can use the *Edit Operation Button* ④ to *browse*, *accept* and *ignore* the edit options. The accepted edit (and their follow-up modification) will be recorded as prior edits for next recommendation.
- **Cloud Service**: Last, we follow the design of Copilot to deploy CoEdPilot on the cloud so that the user request (e.g., for edit location and edit generation) and their response are communicated between the server and the client. Users can check the network connection by *Query State* ⑤ as in Figure 7.

# 5 EXPERIMENT

We evaluate CoEdPilot with the following research questions:

- **RQ1 (Locating Propagating Files, see Section 3.1.1)**: Can CoEdPilot locate the edit-propagating source files?
- **RQ2 (Locating Propagating Lines, see Section 3.1.2)**: Given the located source files, can CoEdPilot locate the edit-propagating lines of code?
- **RQ3 (Edit Generation, see Section 3.3)**: Given edit location, what is the performance of generating edit options?
- **RQ4 (Prior Edit Relevance, see Section 3.1.2)**: Can CoEdPilot select the relevant prior edits accurately?
- **RQ5 (Performance Boost for State-of-the-arts)**: Whether the framework of CoEdPilot further boost the performance of the state-of-the-art solutions?

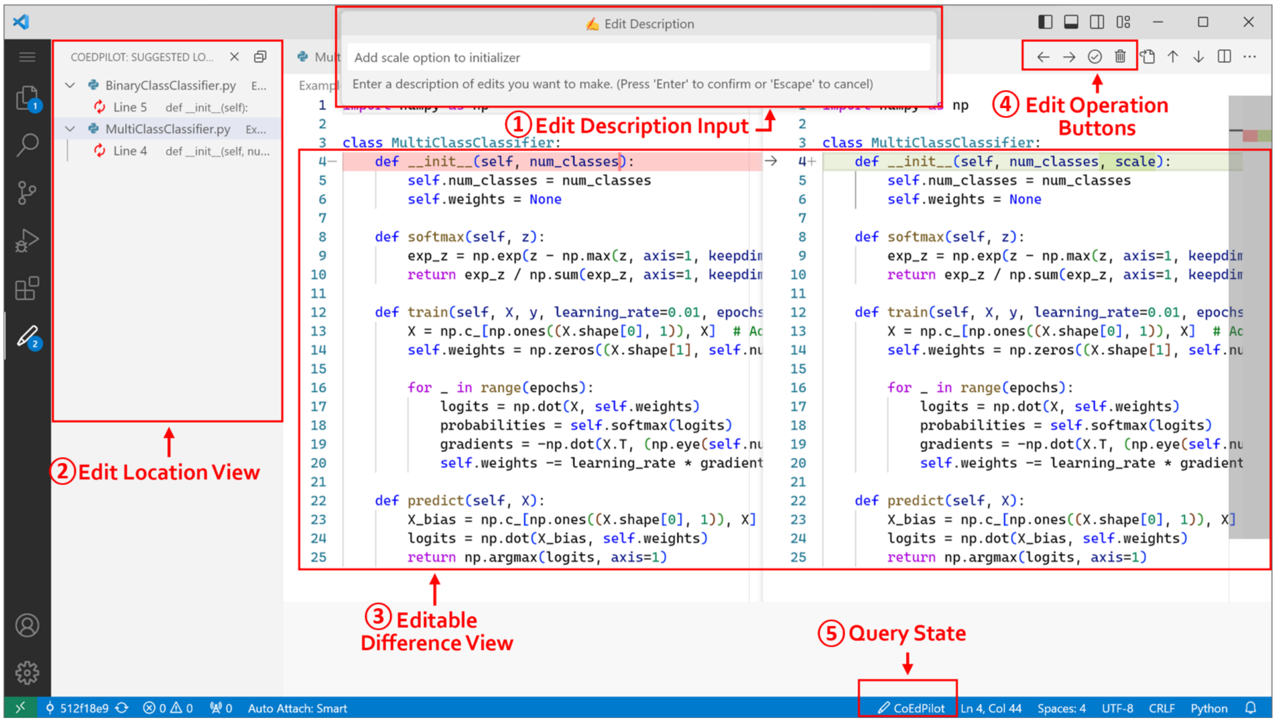
Note that, CoEdPilot serves more as a complementary framework to enhance the state-of-the-art edit generator by locating subsequent edits and capturing relevant prior edits. Thus, in RQ5, we briefly compare our edit generation model with the state-of-the-arts, followed by how we can boost their performance.

## 5.1 Benchmark Construction

To evaluate the performance of CoEdPilot, we construct a benchmark of 5 programming languages (i.e., JavaScript, Java, Go, Python, and TypeScript) from 471 open-source projects. Upon construction, we select the top 100 projects from GitHub according to the number of their stars. For each programming language, we remove the projects with educational purposes (e.g., tutorial) or non-English commit messages. For each project, we select commits with the following criteria[2] in our dataset:

- A commit shall include at least three hunks;
- A commit shall include hunks with the number of changed lines of code less than 15 (considering the length limit of our model);
- The commit message shall be an English message with a token length over 5;
- The commit shall not contain the automatically generated source files (e.g., the Java files with `@auto` keywords) or non-source files (e.g., `.bak`, `.log`, and `.pyc` files)

[2] In this work, we provide our definition of good quality, but we encourage the practitioners to adjust the definition according to their practical scenarios.



Add scale option to initializer

Enter a description of edits you want to make. (Press 'Enter' to confirm or 'Escape' to cancel)

```
class MultiClassClassifier:
    def __init__(self, num_classes):
        self.num_classes = num_classes
        self.weights = None

    def softmax(self, z):
        exp_z = np.exp(z - np.max(z, axis=1, keepdim
        return exp_z / np.sum(exp_z, axis=1, keepdim

    def train(self, X, y, learning_rate=0.01, epochs
        X = np.c_[np.ones((X.shape[0], 1)), X]  # Ac
        self.weights = np.zeros((X.shape[1], self.nu

        for _ in range(epochs):
            logits = np.dot(X, self.weights)
            probabilities = self.softmax(logits)
            gradients = -np.dot(X.T, (np.eye(self.nu
            self.weights -= learning_rate * gradient

    def predict(self, X):
        X_bias = np.c_[np.ones((X.shape[0], 1)), X]
        logits = np.dot(X_bias, self.weights)
        return np.argmax(logits, axis=1)
```

```
np

class MultiClassClassifier:
    def __init__(self, num_classes, scale):
        self.num_classes = num_classes
        self.weights = None

    def softmax(self, z):
        exp_z = np.exp(z - np.max(z, axis=1, keepdim
        return exp_z / np.sum(exp_z, axis=1, keepdim

    def train(self, X, y, learning_rate=0.01, epochs
        X = np.c_[np.ones((X.shape[0], 1)), X]  # Ac
        self.weights = np.zeros((X.shape[1], self.nu

        for _ in range(epochs):
            logits = np.dot(X, self.weights)
            probabilities = self.softmax(logits)
            gradients = -np.dot(X.T, (np.eye(self.nu
            self.weights -= learning_rate * gradient

    def predict(self, X):
        X_bias = np.c_[np.ones((X.shape[0], 1)), X]
        logits = np.dot(X_bias, self.weights)
        return np.argmax(logits, axis=1)
```

**Figure 7: The screenshot of CoEdPilot tool, implemented as a Visual Studio Code extension.**

**Table 3: Benchmark of CoEdPilot on 471 open source projects on 5 programming languages. For the columns of 'Train', 'Valid', 'Test' dataset, we show the number of their training samples.**

| Language | Model | Train | Valid | Test | #Proj | #Com | #File | #Hunk |
|---|---|---|---|---|---|---|---|---|
| JavaScript | File location | 22K | 3K | 6K | 93 | 34K | 34K | 658K |
| | Line location | 382K | 54K | 109K | | | | |
| | Edit generation | 460K | 65K | 130K | | | | |
| Java | File location | 68K | 10K | 20K | 89 | 24K | 72K | 556K |
| | Line location | 335K | 47K | 95K | | | | |
| | Edit generation | 389K | 55K | 111K | | | | |
| Go | File location | 46K | 7K | 14K | 98 | 50K | 88K | 1174K |
| | Line location | 695K | 99K | 198K | | | | |
| | Edit generation | 822K | 117K | 234K | | | | |
| Python | File location | 60K | 9K | 17K | 91 | 33K | 42K | 555K |
| | Line location | 327K | 46K | 93K | | | | |
| | edit generation | 389K | 55K | 111K | | | | |
| TypeScript | File location | 65K | 9K | 17K | 100 | 39K | 76K | 817K |
| | Line location | 480K | 68K | 137K | | | | |
| | Edit generation | 572K | 81K | 163K | | | | |

As a result, we have the dataset as shown in Table 3, with an average commit filter rate of 6.89%. Further, we train our dependency analyzer on 49 sampled projects of different programming languages with 77K positive pairs and 24K randomly sampled negative pairs.

## 5.2 Experiment Setup

*5.2.1 RQ1 (Propagating-file Location).* We extract a commit with $k$ hunks as a set, denoted as $H$, located in $m$ source files, to construct $k$ training samples. In each sample, we select one hunk $h \in H$ as the target hunk. Assume that the $m'$ files with other hunks as the ground-truth positive files, and we randomly select $n$ ($n > m$) files not in the commit as negative files. If CoEdPilot reports $g$ files as positive and $h$ out of $g$ files are true positive, we measure the precision of file location as $\frac{h}{g}$ and the recall of the file location as $\frac{h}{m}$.

*5.2.2 RQ2 (Propagating-line Location).* We parse a commit with $k$ hunks as a set, denoted as $H$, located in $m$ files to construct $k$ training samples as follows. Each time, we select one out of $k$ hunks, i.e., $h \in H$, as the target edit to be predicted. We select relevant prior edits from $H \setminus \{h\}$ with CoEdPilot. Then we apply a sliding window of size $s$ across the $m$ files for CoEdPilot to report the hunk $h$. We apply the above procedure for $k$ times, each of which we select a different target edit.

We measure the average accuracy, precision, and recall in the $k$ times as follows. In each time, for the $m$ lines of code not in the prior edits, we measure the accuracy as $\frac{n}{m}$, where $n$ is the number of lines with edit type predicted accurately. Further, we compute the precision and recall for each of the three labels individually. Assume for each edit type, there are $l$ positive lines of code and CoEdPilot reports $t$ lines of code as positive and $d$ out of $t$ lines are the true positive, thus we have the precision as $\frac{d}{t}$ and the recall as $\frac{d}{l}$. Given the imbalance in sample sizes across these labels, we employ the macro-averaging method to calculate the final precision and recall.

*5.2.3 RQ3 (Edit Generation).* Given a commit with the set of hunks as $H$, we then choose one hunk $h$ as the target edit, and have $H' = H \setminus \{h\}$, as the prior edits. We use Beam Search to generate the top-1, top-3, top-5, and top-10 edit options for each edit location. For each configuration, we measure its performance with (1) the exact match rate (EMR) for a commit (i.e., an edit session) and (2) the BLEU4 score [45] of the generated edit content. Specifically, assume that we generate the edit content exactly the same as the ground truth edit for $m$ out of $k$ times, the exact match rate is $\frac{m}{k}$. Further, we calculate the highest BLEU4 score from all $k$ times' predictions.

#### 5.2.4 RQ4 (Prior Edit Prediction).

We compare training the edit locating models and the edit generation models with selective prior edits (by our *Edit-dependency Analyzer*) and random prior edits. We compare their performance as mentioned in Section 5.2.2 and Section 5.2.3.

#### 5.2.5 RQ5 (Performance Boost).

We design the experiment as follows. We select the state-of-the-art solutions, i.e., GRACE [25], CCT5 [37], and CoditT5 [58], as the baselines, observe the boosting effect of CoEdPilot. CoPilot [23] is neglected for its programming API is yet published at the time of this work.

- **Rough Edit Location** We provide the baselines with rough location as a general hunk area to observe their performance in generating edits.
- **Precise Edit Location** We equip baselines with our edit location model so that they are fed with specific lines to further observe their performance.

We measure the performance of edit generation as in Section 5.2.5. Given the space limit, we provide more experimental details (e.g., hyperparameters, hardware configuration, etc.) in our websites [6].

### 5.3 Results

#### 5.3.1 RQ1 and RQ2 (Propagating-file Location & Line).

Table 4 shows the overall performance of CoEdPilot to detect the edit locations regarding different granularity (i.e., file-level and line-level). We achieve a average precision of 79.52% and a recall of 72.93% to locate the edit file, and the precision of on average 86.97% and the recall of 84.82% to locate the edit lines. We observe that the performance of CoEdPilot lies in identifying the edit pattern (e.g., the commit `4bf1c` in Golang/Go project (see an example at [10]), which demonstrates the concrete example). Further, the average runtime overhead to infer a file takes 1.6s. We probe into the commits and summarize the reasons for false positives and false negatives as follows:

**Reason 1: Noisy Samples in the Training Dataset.** As for inferring the location of subsequent edits, we find that the quality of the dataset is of vital importance. Despite that we have set a number of criteria to filter out some commits, we still observe that noisy training samples might introduce negative effects. One of the observations is that some programmers can submit some *irrelevant* changes files (as well as the edits) in a single commit, which makes CoEdPilot challenging to report some edit locations. Further, we find that quite a number of edits are about code comments and documentation (e.g., the commit `3f442` in `golang/go` project [9]), which may not be well captured by CoEdPilot.

Nevertheless, cleaning the whole dataset regarding the edit relevance is a non-trivial work, which is iterative and interactive between human observation/interpretation and automatic inference. Thus, we leave the solution in our future work.

**Table 4: The accuracy of propagating-file & line location**

| Programming Language | File Location | | Line Location | | |
|---|---|---|---|---|---|
| | Precision (%) | Recall (%) | Accuracy (%) | Precision (%) | Recall (%) |
| JavaScript | 81.52 | 71.21 | 94.89 | 86.62 | 83.88 |
| Python | 70.84 | 73.40 | 94.48 | 85.03 | 82.64 |
| Java | 85.28 | 75.67 | 95.37 | 87.99 | 85.99 |
| Go | 80.10 | 72.12 | 95.79 | 88.99 | 87.32 |
| TypeScript | 79.84 | 72.25 | 95.23 | 86.21 | 84.25 |
| Average | 79.52 | 72.93 | 95.15 | 86.97 | 84.82 |

**Table 5: The performance of edit generation**

| Programming Language | Metric | Top-1 | Top-3 | Top-5 | Top-10 |
|---|---|---|---|---|---|
| Javascript | **BLEU4** | 60.70 | 69.71 | 71.37 | 73.02 |
| | **EMR(%)** | 41.83 | 47.50 | 49.31 | 50.99 |
| Python | **BLEU4** | 57.59 | 65.65 | 67.47 | 69.11 |
| | **EMR(%)** | 33.48 | 38.52 | 40.41 | 42.09% |
| Java | **BLEU4** | 60.54 | 68.35 | 70.11 | 71.73 |
| | **EMR(%)** | 40.69 | 46.87 | 48.78 | 50.51 |
| Go | **BLEU4** | 65.37 | 71.96 | 73.47 | 74.98 |
| | **EMR(%)** | 48.94 | 55.09 | 57.18 | 59.16% |
| Typescript | **BLEU4** | 61.75 | 70.31 | 71.99 | 73.68 |
| | **EMR(%)** | 41.58 | 46.86 | 48.57 | 50.65 |

**Table 6: Relevance of prior edits on edit location & generation**

| Prior Edit Relevance | Edit-propagating line locator | | | Edit-content generator | |
|---|---|---|---|---|---|
| | Accuracy (%) | Precision (%) | Recall (%) | EMR (%) | BLEU4 |
| Selective Prior Edits | 94.89 | 86.62 | 83.88 | 41.83 | 60.70 |
| Random Prior Edits | 91.86 | 81.73 | 72.37 | 18.87 | 46.56 |

**Reason 2: Informativeness of Edit Inference.** Further, we observe that some false negatives are caused by single-directed interaction. For example, an addition of method call implies an addition of importing a relevant library declaring the method, however, the implication does not hold in the other way (see example at [11]). When some interactions between the edits are not causal, the inference becomes more challenging.

#### 5.3.2 RQ3 and RQ4 (Edit Generation & Prior Edit Prediction).

Table 5 shows the overall performance of edit generation with Top-k candidates. Further, Table 6 shows the relevance of prior edits in locating the subsequent edits and edit content generation. We can see that (1) CoEdPilot achieves good performance in generating the edit options, and (2) the selective prior edits play a vital role in enhancing the performance. An example can be referred to [11], where CoEdPilot is good at capturing the edit pattern (via syntactic dependency or semantic relevance). The random prior edits can break the pattern, which introduces additional edit chaos during the recommendation. Further, we find that the mis-prediction of the edit options shares similar reasons introduced in Section 5.3.1.

#### 5.3.3 RQ5 (Performance Boost).

In Table 7, we compare CoEdPilot with three baselines, i.e., GRACE, CCT5, and CoditT5, in

**Table 7: Performance Boost with CoEdPilot**

| Approach | EMR(%) | BLEU4 |
|---|---|---|
| CoEdPilot (Line Locator + Edit Generator) | **29.96** | **78.58** |
| CoditT5 | 7.42 | 69.01 |
| GRACE | 2.73 | 38.36 |
| CCT5 | 14.19 | 75.37 |
| GRACE + Line Locator | 18.61 | 71.61 |
| CCT5 + Line Locator | 15.45 | 78.27 |

**Table 8: Runtime Estimation of CoEdPilot**

| Step | File locator<br>(s / file) | Line locator<br>(s / file) | Edit-content generator<br>(s / location) |
|---|---|---|---|
| Prepare Input | 0.0064 | 0.3976 | 0.0683 |
| Model Inference | 0.1008 | 0.0878 | 0.3972 |
| Total | 0.1072 | 0.4854 | 0.4655 |

generating top-1 edit option. As described in Section 5.2.5, the fine-tuned baselines are fed with the hunk-level location (i.e., the lines included in a hunk) to predict the edited code. We can see that the performance gap between CoEdPilot and the baselines are large. The reason lies in that the edit locator can largely help the edit generator to generate edits in a far more precise position.

Given that CoEdPilot is an extensible and integrable framework, we replace our edit generation model with the fine-tuned baselines, observing that the performance of both GRACE and CCT5 is boosted significantly. Note that, CoditT5 can predict location as CoEdPilot, we do not equip it with our locator. In comparison to CoditT5, we observe that our two-stage model has the advantage of utilizing more input length given the limit of existing base models such as CodeT5.

## 6 USER STUDY

To further evaluate how the programmers can use CoEdPilot as a tool in practice, we design a user study to evaluate its functionalities as described in Section 4.

**Baseline.** To evaluate whether the design of CoEdPilot can well support practical code edits, we choose Copilot [23] as a baseline for its wide popularity for generating code. We omit the full manual editing mode in this study because (1) Copilot is supported by powerful GPT-3.5 Turbo, which is shown to improve the programming productivity by 27% - 57% [42], and (2) the limitation of budget and overhead.

**Participant.** We recruit 18 participants from three universities in China and Singapore, including both undergraduate and graduate students. We conduct a pre-study (including a test) based on their programming experience. Their demographic analysis is available at [6]. We divide them into two equivalent groups based on their experience. The experimental group uses CoEdPilot while the control group uses Copilot in the study.

**Code Edit Tasks.** To ensure that the participants can focus on editing with a light-weighted overhead of comprehension, we extract a simplified version from three real-world commits. The tasks are selected as follows:

- **Bug Fix (Task 1)**: We show a bug as mistakenly used `range(arr)` for `range(len(arr))` in the project. We ask the participants to find and fix multiple such mistaken uses across the project.
- **Refactoring (Task 2)**: We ask the participants to extract three pieces of duplicated code into a new function.
- **Feature Enhancement (Task 3)**: We ask the participants to introduce a *scale* capability to normalize the input vectors for existing class classifiers, which requires multiple edit propagation.

**Table 9: Overall performance of EG (Experimental Group) and CG (Control Group):The completion time is in seconds.**

| EG | Task1 | Task2 | Task3 | CG | Task1 | Task2 | Task3 |
|---|---|---|---|---|---|---|---|
| **P1** | 221 | 515 | 1196 | **P10** | 339 | 696 | 1287 |
| **P2** | 897 | 389 | 279 | **P11** | 360 | 776 | 1563 |
| **P3** | 366 | 487 | 216 | **P12** | 480 | 483 | 545 |
| **P4** | 160 | 529 | 963 | **P13** | 522 | 724 | 1770 |
| **P5** | 230 | 301 | 756 | **P14** | 277 | 395 | 838 |
| **P6** | 364 | 473 | 617 | **P15** | 181 | 446 | 930 |
| **P7** | 329 | 688 | 588 | **P16** | 337 | 720 | 825 |
| **P8** | 840 | 780 | 1020 | **P17** | 151 | 666 | 1515 |
| **P9** | 290 | 638 | 1050 | **P18** | 266 | 722 | 1563 |
| **Average** | 410.78 | 533.33 | 742.78 | **Average** | 323.67 | 625.33 | 1070.33 |

**Study Setup.** We conducted a warm-up session with a tutorial for both CoEdPilot and Copilot, followed by a practice task, to familiarize them with the tools. For each task, we allocate each participant with 30 minutes to accomplish. We prepare the test cases for each edit task for them to validate their edits. The test cases are designed to guarantee that all the participants can confirm their accomplishing edits. During the study, we ask the participants to run a video-recorder so that we can conduct the post-mortem analysis. Finally, we measure their performance regarding (1) whether they can successfully accomplish the tasks (i.e., all the test cases passed) and (2) their efficiency in accomplishing the tasks.

**Results.** Table 9 shows the participants' performance to accomplish the code-editing tasks, with the following observation:

- **Task 1**: EG underperforms CG in Task 1 on average completion time without statistical significance (the $p$-value in Wilcoxon Signed Rank test is 0.33 and the effect size is -0.08).
- **Task 2**: EG outperforms CG in Task 2 on average completion time without statistical significance (the $p$-value in Wilcoxon Signed Rank test is 0.07 > 0.05 and the effect size is 0.60).
- **Task 3**: In contrast, EG outperforms CG in Task 3 on the completion time with statistical significance (the $p$-value is $0.003 < 0.05$ and the effect size is 0.96).

**Why CG outperforms EG in Task 1?** In Task 1 (i.e., fixing a duplicated bug), we observe that CoEdPilot users (EG group) still suffer from the learning curve of the new tool for running the function of predicting edit location and edit content. Further, some participants (P2 and P8) were still building their trust in our recommendations such as edit location and edit content, despite that they are accurate. As a result, they spend more time confirming our results. We deem this a common problem for any new tool deployed on either user study or production line. In contrast, given that the edit pattern in the task is simple, some Copilot user (e.g., P17) tries to search the expression across the project. In the other words, they address their need of edit location with keyword-based search in Task 1.

**Why EG outperforms CG in Task 2 but without statistical significance?** In Task 2 (i.e., refactoring by method extraction), the CoEdPilot users become more experienced in adapting our tool by switching between different functions such as location prediction, edit generation, edit option selection, etc. The accurate edit location can largely mitigate the efforts in finding the cross-file code duplication for the new function. Compared to the CG participants with manually summarized edit patterns, the EG group is gradually outperforming the CG group (the $p$-value 0.07 is closer to 0.05).

**How EG outperforms CG in Task 3?** Task 3 (i.e., enhance the model training with *scale* function) is the most difficult task, where the editing pattern cannot be captured by keyword search. For example, one edit to insert a `scale` parameter is associated with another edit to insert a follow-up decision logics with the `scale` variable. In such a scenario, EG outperforms CG in general.

Nevertheless, we observe that the performance of the participants varies, some accomplish the task in less than 5 minutes (e.g., P2) while some take a longer time (e.g., P1, P8, and P9). We investigate their tool logs and videos, finding that some participants modify the edit content with their own interpretation, which leads to the buggy code. Taking the buggy edit as the prior edits, CoEdPilot can generate confusing edits afterwards. Only by running the test cases to validate the results, the participants can realize they produce a bug during the editing. Human mistakes in such an interaction-based tool are a long-standing problem, we will address the issue in our future work. Further, the CoEdPilot group accepts recommended 69.3% edit options, among which they modify 31.6% generated edits. For the space limit, more statistics of user behaviors in the study are available at [6].

## 7 THREAT TO VALIDITY

Several aspects of the user study may impair its validity:

**Internal Validity:** In this study, the experiment group may face a steeper learning curve, while the control group is already acquainted with CoPilot. This learning disparity could lead to observed differences in the test that are attributed to learning effects rather than the actual performance of the extension.

**External validity:** The edit tasks are simplified versions of code derived from actual commits and equipped with comprehensive instructions. This modification might deviate from the real editing scenario. Moreover, as edit tasks exclusively focus on Python in this study, such specificity choice could confound the interpretation of the plugin's effectiveness.

**Statistical Validity:** Due to the limitation of time and resources, we recruited 18 participants in the study. The relatively small size may not provide sufficient statistical power to detect genuine differences in the effectiveness of the extension. Consequently, the generalizability and robustness of the study findings might be compromised.

## 8 RELATED WORK

**Code Generation.** Code generation is long standing software engineering task [13, 35, 48, 55, 56], which starts from sequence-based and tree-based approaches [39, 49], and gravitates towards pre-trained language models, such as BERT [17], GPT [12], T5 [26, 46], CodeBERT [19], GraphCodeBERT [24], DietCodeBERT [60], CodeT5 [54], CodeT5+ [53], CodeT [16], and Incoder [22]. Recently, StarCoder [34] is trained with over 8 programming languages, Git commits, GitHub issues, and Jupyter notebooks. It outperforms existing open Code LLMs on popular programming benchmarks and matches or surpasses closed models such as code-cushman-001 from OpenAI (the original Codex [43] model). Meanwhile, our approach uniquely generates incremental edits, rather than new code.

**Code Edit Generation.** Among the work to edit code [14, 15, 27, 36, 38, 58, 59, 61], Codit [14] is the first to introduce tree-based neural networks for predicting the edits. Following their tree model structure, Recoder [61] introduces another abstract syntax tree (AST) reader along with the code reader to outperform the Codit model. Further, CURE [27] introduces the pre-training models for automatic program repair. CoditT5 [58] pre-train a CodeT5 [54] base model with the input including natural language comments and edit code hunk and the output including an edit plan. The current state-of-the-art transformer-based model is GRACE [25], which trains a prompting large language model [57] with a designed prompt to include the associated code update. Overwatch [59] symbolically analyzes edit sequence patterns by formulating them into rules based on prior program transformations. Our solution, CoEdPilot, is complementary to the majority of the transformer-based code-edit generation model. In addition to exploring the interactive nature of code edits, we further learn to capture the relevant prior edits and subsequent edit location.

## 9 CONCLUSION

In this work, we introduce CoEdPilot, an end-to-end framework to interactively generate code edits by orchestrating a set of neural transformers as components, regarding prior edit analysis, subsequent edit analysis, and edit generation. Our extensive experiments show that CoEdPilot is able to predict the edit location and generate edit options in an effective way. Further, the framework is complementary to a set of state-of-the-art edit generators to boost their performance. Our user study shows that CoEdPilot as a VS Code plugin is effective in assisting programmers in practice. In the future, we will improve the quality of the training dataset for a more effective model and address the potential mistaken human feedback in tool design.

## 10 DATA AVAILABILITY

Our models and datasets are published in our HuggingFace website [5], both source code and VS Code extension are available in our GitHub [3, 4].

## ACKNOWLEDGMENTS

This research is supported in part by National Key Research and Development Program of China (Grant No.2023YFB4503802), the Bytedance Network Technology, the Minister of Education, Singapore (T2EP20120-0019, MOET32020-0004), the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity Research and Development Programme (Award No. NRF-NCR_TAU_2021-0002), National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-GC-2023-008).

# REFERENCES

[1] 2023. CrossEntropyLoss — PyTorch 2.1 documentation. https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html.

[2] 2023. Visual Studio Code. https://code.visualstudio.com/.

[3] 2024. code-philia/CoEdPilot-extension: Extension for CoEdPilot. https://github.com/code-philia/CoEdPilot-extension.

[4] 2024. code-philia/CoEdPilot: Source code for CoEdPilot. https://github.com/code-philia/CoEdPilot.

[5] 2024. CoEdPilot - a code-philia Collection. https://huggingface.co/collections/code-philia/coedpilot-65ee9df1b5e3b11755547205.

[6] 2024. CoEdPilot website. https://sites.google.com/view/coedpilot/home.

[7] Danyah Alfageh, Hosam Alhakami, Abdullah Baz, Eisa Alanazi, and Tahani Alsubait. 2020. Clone Detection Techniques for JavaScript and Language Independence. *International Journal of Advanced Computer Science and Applications* 11, 4 (2020). https://doi.org/10.14569/IJACSA.2020.01104102

[8] Karim Ali and Ondřej Lhoták. 2013. Averroes: Whole-program analysis without the whole program. In *European Conference on Object-Oriented Programming*. Springer, 378–400. https://doi.org/10.1007/978-3-642-39038-8_16

[9] Anonymous. 2023. An commit example to modify a number of comments. https://github.com/golang/go/commit/e914671f5d5e72b2f897a9f2dfc6bf2203d3254a.

[10] Anonymous. 2023. An commit example with edit pattern. https://github.com/golang/go/commit/4bf1ca4b0ce9a08f4c45d68fe49857914f668f69.

[11] Anonymous. 2023. An commit example with single-directed edit inference. https://github.com/golang/go/commit/400e24a8be852e7b20eb4af1999b28c20bb4ea21.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901. https://doi.org/10.48550/arXiv.2005.14165

[13] Yufan Cai, Yun Lin, Chenyan Liu, Jinglian Wu, Yifan Zhang, Yiming Liu, Yeyun Gong, and Jin Song Dong. 2024. On-the-Fly Adapting Code Summarization on Trainable Cost-Effective Language Models. *Advances in Neural Information Processing Systems* 36 (2024).

[14] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2022. CODIT: Code Editing With Tree-Based Neural Models. *IEEE Transactions on Software Engineering* 48, 4 (2022), 1385–1399. https://doi.org/10.1109/TSE.2020.3020502

[15] S. Chakraborty and B. Ray. 2021. On Multi-Modal Learning of Editing Source Code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 443–455. https://doi.org/10.1109/ASE51524.2021.9678559

[16] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022). https://doi.org/10.48550/arXiv.2207.10397

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. (June 2019), 4171–4186. https://doi.org/10.18653/v1/N19-1423

[18] Schaeffer Duncan, Andrew Walker, Caleb DeHaan, Stephanie Alvord, Tomas Cerny, and Pavel Tisnovsky. 2021. Pyclone: A Python Code Clone Test Bank Generator. In *Information Science and Applications: Proceedings of ICISA 2020*. Springer, 235–243. https://doi.org/10.1007/978-981-33-6385-4_22

[19] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. *EMNLP* (2020). https://doi.org/10.18653/v1/2020.findings-emnlp.139

[20] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041

[21] Markus Freitag and Yaser Al-Onaizan. 2017. Beam Search Strategies for Neural Machine Translation. In *Proceedings of the First Workshop on Neural Machine Translation*. Association for Computational Linguistics, Vancouver, 56–60. https://doi.org/10.18653/v1/W17-3207

[22] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations*. https://doi.org/10.48550/arXiv.2204.05999

[23] GitHub. 2023. GitHub Copilot. https://github.com/features/copilot

[24] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training code representations with data flow. *The International Conference on Learning Representations* (2020). https://doi.org/10.48550/arXiv.2009.08366

[25] Priyanshu Gupta, Avishree Khare, Yasharth Bajpai, Saikat Chakraborty, Sumit Gulwani, Aditya Kanade, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2023. Grace: Language Models Meet Code Edits. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1483–1495. https://doi.org/10.1145/3611643.3616253

[26] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Yuan Yao, Ao Zhang, Liang Zhang, et al. 2021. Pre-trained models: Past, present and future. *AI Open* 2 (2021), 225–250. https://doi.org/10.48550/arXiv.2106.07139

[27] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[28] Wuxia Jin, Yuanfang Cai, Rick Kazman, Qinghua Zheng, Di Cui, and Ting Liu. 2019. ENRE: A Tool Framework for Extensible eNtity Relation Extraction. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 67–70. https://doi.org/10.1109/ICSE-Companion.2019.00040

[29] Wuxia Jin, Dinghong Zhong, Yuanfang Cai, Rick Kazman, and Ting Liu. 2022. Evaluating the Impact of Possible Dependencies on Architecture-level Maintainability. *IEEE Transactions on Software Engineering* (2022), 1–1. https://doi.org/10.1109/TSE.2022.3171288

[30] Iman Keivanloo, Feng Zhang, and Ying Zou. 2015. Threshold-free code clone detection for a large-scale heterogeneous java repository. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 201–210. https://doi.org/10.1109/SANER.2015.7081830

[31] Rob Kitchin and Martin Dodge. 2014. *Code/space: Software and everyday life*. Mit Press. https://doi.org/10.7551/mitpress/9780262042482.001.0001

[32] Thomas D LaToza and Brad A Myers. 2010. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 185–194. https://doi.org/10.1145/1806799.1806829

[33] Jia Li, Ge Li, Zhuo Li, Zhi Jin, Xing Hu, Kechi Zhang, and Zhiyi Fu. 2023. CodeEditor: Learning to Edit Source Code with Pre-Trained Models. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 143 (sep 2023), 22 pages. https://doi.org/10.1145/3597207

[34] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023). https://doi.org/10.48550/arXiv.2305.06161

[35] Xiaonan Li, Daya Guo, Yeyun Gong, Yun Lin, Yelong Shen, Xipeng Qiu, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022. Soft-labeled contrastive pre-training for function-level code representation. *arXiv preprint arXiv:2210.09597* (2022).

[36] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating Code Review Activities by Large-Scale Pre-Training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1035–1047. https://doi.org/10.1145/3540250.3549081

[37] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. 2023. CCT5: A Code-Change-Oriented Pre-Trained Model. *arXiv preprint arXiv:2305.10785* (2023). https://doi.org/10.1145/3611643.3616339

[38] Yun Lin, Xin Peng, Zhenchang Xing, Diwen Zheng, and Wenyun Zhao. 2015. Clone-based and interactive recommendation for modifying pasted code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 520–531.

[39] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. 2016. Latent Predictor Networks for Code Generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 599–609. https://doi.org/10.18653/v1/P16-1057

[40] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. *arXiv preprint arXiv:2210.14306* (2022). https://doi.org/10.48550/arXiv.2210.14306

[41] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, and Hridesh Rajan. 2013. A study of repetitiveness of code changes in software evolution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 180–190. https://doi.org/10.1109/ASE.2013.6693078

[42] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 1–5. https://doi.org/10.1145/3524842.3528470

[43] OpenAI. 2020. CodeX. https://openai.com/blog/openai-codex.

[44] OpenAI. 2021. ChatGPT. https://openai.com/chatgpt. Accessed on March 29, 2023.

[45] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, Pierre

Isabelle, Eugene Charniak, and Dekang Lin (Eds.). Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, 311–318. https://doi.org/10.3115/1073083.1073135

[46] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. 21, 1, Article 140 (jan 2020), 67 pages. https://doi.org/10.48550/arXiv.1910.10683

[47] Julian Salazar, Davis Liang, Toan Q Nguyen, and Katrin Kirchhoff. 2019. Masked language model scoring. *arXiv preprint arXiv:1910.14659* (2019). https://doi.org/10.18653/v1/2020.acl-main.240

[48] Jiho Shin and Jaechang Nam. 2021. A survey of automatic code generation from natural language. *Journal of Information Processing Systems* 17, 3 (2021), 537–555. https://doi.org/10.3745/JIPS.04.0216

[49] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991. https://doi.org/10.48550/arXiv.1911.09983

[50] Ekincan Ufuktepe, Tugkan Tuglular, and Kannappan Palaniappan. 2022. Tracking code bug fix ripple effects based on change patterns using markov chain models. *IEEE Transactions on Reliability* 71, 2 (2022), 1141–1156. https://doi.org/10.1109/TR.2022.3167943

[51] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224. https://doi.org/10.1145/1925805.1925818

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017). https://doi.org/10.48550/arXiv.1706.03762

[53] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023). https://doi.org/10.48550/arXiv.2305.07922

[54] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021). https://doi.org/10.48550/arXiv.2109.00859

[55] Chen Yang, Yan Liu, and Changqing Yin. 2021. Recent Advances in Intelligent Source Code Generation: A Survey on Natural Language Based Studies. *Entropy* 23, 9 (2021), 1174. https://doi.org/10.3390/e23091174

[56] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. *arXiv preprint arXiv:2305.04207* (2023). https://doi.org/10.48550/arXiv.2305.04207

[57] Biao Zhang, Barry Haddow, and Alexandra Birch. 2023. Prompting large language model for machine translation: A case study. *arXiv preprint arXiv:2301.07069* (2023). https://doi.org/10.48550/arXiv.2301.07069

[58] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022. CoditT5: Pretraining for Source Code and Natural Language Editing. In *International Conference on Automated Software Engineering*. https://doi.org/10.48550/arXiv.2208.05446

[59] Yuhao Zhang, Yasharth Bajpai, Priyanshu Gupta, Ameya Ketkar, Miltiadis Allamanis, Titus Barik, Sumit Gulwani, Arjun Radhakrishna, Mohammad Raza, Gustavo Soares, and Ashish Tiwari. 2022. Overwatch: Learning Patterns in Code Edit Sequences. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 139 (oct 2022), 29 pages. https://doi.org/10.1145/3563302

[60] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: Simplifying programs for pre-trained models of code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1073–1084. https://doi.org/10.48550/arXiv.2206.14390

[61] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 341–353. https://doi.org/10.1145/3468264.3468544

Received 16-DEC-2023; accepted 2024-03-02