Learning Project-wise Subsequent Code Edits via Interleaving Neural-based Induction and Tool-based Deduction

Chenyan Liu[1,2], Yun Lin[1*], Yuhuan Huang[1], Jiaxin Chang[1], Binhang Qi[2],
Bo Jiang[3], Zhiyong Huang[2], and Jin Song Dong[2]
[1]Shanghai Jiao Tong University, Shanghai, China
[2]National University of Singapore, Singapore
[3]Bytedance Network Technology, Beijing, China
chenyan@u.nus.edu, {lin_yun, hyh0u0, cjx001234}@sjtu.edu.cn, qibh@nus.edu.sg,
jiangbo.jacob@bytedance.com, huangzy@comp.nus.edu.sg, dcsdjs@nus.edu.sg

***Abstract*—In industrial and open-source software engineering tasks, developers often perform project-wise code editing tasks, including feature enhancement, refactoring, and bug fixing, where the leading AI models are expected to support the productivity. Hence, researchers and practitioners have proposed and adopted many LLM-based solutions to facilitate their real-world development. However, they largely suffer from the balance among predicting scope, accuracy, and efficiency. For example, solutions like Cursor achieve high accuracy only in a local editing scope while its performance drops on cross-file edits. In contrast, solutions like CoEdPilot exhibit efficiency limitations when used to predict project-wise edits.**

In this work, we propose TRACE (Tool-integrated RecommendAtion for Code Editing), a novel subsequent code editing solution to push the boundary of scope, accuracy, and efficiency. Our rationale lies in that code edits are triggered for either semantic or syntactic reasons. Therefore, TRACE predicts subsequent edits by interleaving neural-based induction for semantic edit prediction and tool-based deduction for syntactic edit prediction. The tools can be any IDE facilities, such as refactoring tools (e.g., rename) or linting tools (e.g., use-def), providing decent performance of *deducing* edit-location and edit-generation. Technically, we address the challenge of (1) when to interleave between neural-based and tool-based prediction and (2) how to further improve the performance of neural-based prediction. As for the former, we learn a neural model to detect when to invoke IDE editing tools. As for the latter, we propose a novel and fine-grained editing representation to further boost the performance of neural editing models.

Our extensive experiments show that, in comparison to the state-of-the-arts such as CoEdPilot, GrACE, and CCT5, TRACE significantly improves the performance of edit location (by 43.76%) and edit generation (by 11.16%). Our simulation experiment on an interactive editing setting shows that TRACE achieves an acceptance rate 6.15% higher than Cursor. Moreover, our user study consists of 24 participants on Cursor, CoEdPilot, and TRACE, on three code editing tasks. The results show that the experimental group with TRACE achieves leading performance on cross-file global edits. In addition, we observe concerning user behaviours on how participants deal with false predictions by the tools, shedding light on the design of future code-editing tools.

***Index Terms*—code editing, subsequent edit prediction, neural-based induction, tool-based deduction**

I. Introduction

Recent years have witnessed a surge in applying large language models (LLM) for code generation [1], [2], [3], [4], [5]. Despite their success in translating natural language descriptions into target code, incremental code edits across the project are more common in practice. Empirical observation shows that such incremental code edits account for over 70% of changes over the code commit history [6]. It highlights the need for predicting project-wise subsequent code edits, which can significantly improve the productivity of software developers. Existing works in the community have made the following efforts to address the problem.

Local Edit Solution. A line of code edit works [7], [8], [9], [10], [11] simplifies the edit generation problem into a machine translation task. In general, these approaches take as input the pre-edit code, and generate the post-edit code as output. Since those approaches cannot predict the edit location, they are generally limited when being applied to predict project-wise subsequent edits.

Project-wise Edit Solution. In this context, several industry solutions, such as Copilot Edits and Cursor's next edit suggestion [12], [13], have emerged. However, due to latency and cost considerations, these solutions appear to be conservative when recommending cross-file edits, often degrading into merely local edit suggestions. When cross-file edits are expected,

This research is supported in part by National Natural Science Fundation of China (62572300), the Minister of Education, Singapore (MOE-T2EP20124-0017, MOET32020-0004), the 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-2023-008-1B), and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme and CyberSG R&D Cyber Research Programme Office. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, Cyber Security Agency of Singapore as well as CyberSG R&D Programme Office, Singapore.

* Corresponding author

users typically rely on features like Copilot Chat or Cursor Chat, where they must specify files to edit, requiring prior knowledge and risking overlooked files.

Meanwhile, in the research community, the most relevant work is CoEdPilot [14], which provides a project-wise solution regarding both edit location and generation, by orchestrating a set of models. A *locator* model is trained to predict the edit type of each line of code in the code window. If a line is predicted to be modified (e.g., insert or replace), then a *generator* model generates its edit solution. Both the locator and generator models also incorporate prior edits as user feedback to infer the implicit edit specification.

Despite these advances toward a practical AI pair programmer solution, they still suffer from the following challenges:

- **C1 (Location Overhead):** A software project can be large, meaning the model may process numerous code windows to locate subsequent edits. Therefore, exhaustively monitoring the whole project upon any code edits, can incur non-trivial runtime overhead, and undermine the performance.
- **C2 (Overlooked Edit Composition):** Existing neural-based solutions predict each code edit as an individual *hunk* (i.e., consecutive lines of code change, see example hunks in Table I). Despite its rapid evolution, neural-based induction inherently suffers from non-trivial computational overhead for scanning the entire codebase and remains fundamentally prone to hallucinations [15]. Moreover, [16] shows that LLMs perform poorly on static analysis tasks and that pre-training on such tasks does not improve general code intelligence. However, in practice, hunks can be highly associated or occur simultaneously, forming high-level actions like refactoring. We refer to these grouped edits as **edit compositions**, where edits exhibit **coherence**, meaning they are likely to propagate to each other, and this propagation can often be captured by static analysis tools, which guarantee both speed and correctness.
- **C3 (Coarse-grained Edit Representation):** The existing state-of-the-arts [7], [8], [9], [10], [11] adopt the git-diff-style representation [17], modeling code edits as replacements or insertions (see Table III H1 as an example). While being widely adopted, such a coarse-grained edit representation can express semantically different editing scenarios in a similar way (see Table III H2 as an example), leading to training inefficiency when learning the code-editing models.

To address the above challenges, we propose TRACE (**T**ool-integrated **R**ecommend**A**tion for **C**ode **E**diting), predicting project-wise subsequent code edits primarily based on a set of prior edits. Our rationale lies in that code edits are triggered for either semantic or syntactic reasons. Therefore, TRACE predicts subsequent edits by interleaving **neural-based induction** for semantic edit prediction and **tool-based deduction** for syntactic edit prediction. Technically, TRACE recommends subsequent edits through a pipelined workflow. At each step, an *edit-composition invoker* first monitors the ongoing session and checks whether the prior edits match pre-defined edit compositions. In such cases, TRACE applies **tool-based deduction**, proactively invoking IDE services (e.g., rename, use-def update, or remove unused imports). This mechanism leverages the syntactic coherence among edits in a composition to quickly narrow down candidate locations, especially when edits span multiple files. If no tool service is triggered, TRACE falls back to **neural-based induction**, which applies a sliding window over the project with our *edit locator* and *edit generator*. Both are equipped with the novel edit representation to distinguish diverse edit scenarios (see Table III). In this way, TRACE integrates tool invocation with neural inference into a feedback-driven pipeline that progressively recommends coherent edits.

We extensively evaluate tool on 38K code commits from 678 projects across 5 programming languages. Compared to state-of-the-arts, (1) TRACE significantly improves edit location precision by 43.76%, recall by 9.96%, and edit generation accuracy by 11.16%. (2) TRACE can well identify edit composition and invoke tools appropriately (92.45% precision and 94.63% recall), (3) The novel edit representation enhances the neural edit locator by 14.57% and generator by 7.40%. In edit simulation, tool reduces time cost by 14.40% and achieves 27.71% suggestion acceptance, comparable to Cursor. A user study with 24 participants on 3 tasks confirms TRACE's effectiveness in recommending project-wise, cross-file edits.

Overall, we summarize our contributions as follows:

- **Methodology.** To the best of our knowledge, we are the first to propose to predict project-wise subsequent edits by interleaving neural-based induction and tool-based deduction. This solution can largely mitigate the LLM hallucination and improve the runtime efficiency (especially for edit location). In addition, we discover a more expressive edit representation to further advance the performance of the neural edit locator and generator.
- **Tool.** We implement TRACE as a Visual Studio Code (VS Code) extension [18] for interactive edit localization and generation. TRACE is designed to smoothly integrate Language Service Protocol (LSP) invocation and neural edit location/generation, potentially helping programmers accomplish their tasks in practice.
- **Evaluation.** We extensively and systematically evaluate TRACE on 38K code commits, spanning over 678 source projects and 5 programming languages, via benchmarks and real-world editing simulation, establishing TRACE as the new state-of-the-art project-wise code editing solution.
- **User Study.** We further conducted a user study with 24 participants over 3 editing tools, totalling about 118 man-hours. The results confirm the effectiveness of our design in practice, also revealing a concerning *over-trust* phenomenon among participants, regardless of their experience. This highlights directions for improving future code-editing solutions.

Additional demonstration videos, source code, supplementary experimental information, and user study video recordings are available at our homepage [19].

TABLE I: Example 1: Composite edit v.s. individual edit

File: executor/window.go		
H1	1	- func renewWithCapacity(chk *Chunk,cap int) *Chunk {
	2	+ func renewWithCapacity(chk *Chunk,cap,maxChunkSize int) *Chunk {
	3	newChk := new(Chunk)
	4	...
H2	5	newChk.capacity = cap
	6	- newChk.requiredRows = cap
	7	+ newChk.requiredRows = maxChunkSize
	8	return newChk
H3	9	- return renewWithCapacity(chk, newCap)
	10	+ return renewWithCapacity(chk, newCap, maxChunkSize)
File: util/chunk/row.go		
H4	11	- newChk := renewWithCapacity(r.c, 1)
	12	+ newChk := renewWithCapacity(r.c, 1, 1)

II. Problem Statement

In this work, the problem statement is formulated as follows. Given the following inputs:

- A software project $P = \{f_1, \ldots, f_n\}$, where f_i is a source file in the project,
- A sequence of prior edits as $E_p = \langle e_1, \ldots, e_k \rangle$ in an edit session, where the subscript $i \in [0, k]$ is the chronological order. Each edit $e \in E_p$ is defined as $e = (f, line_{start}, line_{end}, code_b, code_a)$, where $f \in P$ denotes the file where the edit e happens, $line_{start}$ and $line_{end}$ denote the scope of e in f, $code_b$ and $code_a$ denote code before and after the edit in the scope $(line_{start}, line_{end})$.
- An optional edit description $prompt$.

our TRACE solution is to generate e_{k+1} in the editing session based on P, E_p, and $prompt$ (optional).

Different from the solutions [20], [21], [22] of issue-resolving tasks like SWE-bench [23] which results in a set of patches in a project aligning with an issue description, our task is more progress-driven. Specifically, we predict a subsequent edit aligning with the flow of prior edits with an optional prompt. Note that not all the code editing tasks are driven by a detailed issue description.

III. Motivating Example

A. Tool-inducing Edit Composition

Table I shows four *hunks* (denoted by H_i) over two source files, extracted from commit[1] in project `ping/tidb`. Hunks can be summarized into two actions (A1 and A2):

- **A1: Method Signature Update:** The function `renewWithCapacity()` is updated to include a new parameter `maxChunkSize`, which includes H1 (line 1-4), H3 (line 9-10), and H4 (line 11-12). The three hunks are mutually dependent, as the method signature update propagates across them.
- **A2: Attribute Initialization Update:** H2 (lines 5–8) updates the initialization of `newChk.requiredRows` from variable `cap` to `maxChunkSize`.

The state-of-the-art edit locator, CoEdPilot [14], predicts these hunks sequentially. As for the first action (A1), given a prior edit as H1, it requires an LLM to analyze the *entire* project code to infer whether and what changes will be propagated. The challenge lies in the substantial computational overhead required to scan the entire codebase, as well as the risk of false positives in similar textual patterns, such as `newWithCapacity()` and `renewColumnsWithCapacity()`, caused by LM hallucination.

In this work, we mitigate such overhead and false positives by learning to invoke the inherent static tools in IDE. Specifically, we monitor the code-editing session and infer when to invoke a static tool based on prior edits (e.g., H1), to retrieve the rest of edit composition (e.g., H3 and H4), which can (1) significantly lower the cost and (2) boost the performance of edit location and generation.

Empirical Study. To validate the wide applicability of our approach, we empirically measure the frequency of edit compositions in real-world commits. We collected a dataset of 38K commits from top-starred GitHub repositories across 5 languages (dataset detail refers to Table V in Section V Experiment Setup). Following the benchmark construction methodology described in Section V-D, we applied LSP-based analysis to detect four types of edit compositions: variable renames, function renames, definition/reference lookups, and code clones. Figure 1 shows that 53.6% of commits contain at least one type of predefined edit composition detectable by LSP services, with 16.4% involving multiple types, indicating a high frequency of edit composition in real-world editing scenarios.

Challenges. To deliver this potential, TRACE needs to address *when* and *what* edit composition to invoke. First, an IDE may support various edit compositions, which may result in misleading invocation opportunities. For example, H2 in Table I may indicate a *renaming* edit, as it replaces all instances of `cap` with `maxChunkSize`. However, this reflects a usage change rather than a true renaming edit. Applying a rename tool here may introduce incorrect changes across multiple locations and files, which can be costly to undo the changes. Second, even with accurate composition prediction, key details may remain missing. For example, knowing H1 triggers a *signature update* still leaves the parameter choices in H3 and H4 to be inferred.

Thus, we use TRACE as a hybrid solution combining an edit-composition invoker, a neural edit locator and a neural edit generator that complement each other. On the one hand, the edit-composition invoker can speed up the edit locator and edit generator with higher confidence. On the other hand, the edit locator and edit generator can complement the missing details for the edit-composition invoker. This enables TRACE to better predict project-wise code edits.

B. Edit Representation

Table III shows a commit[2] from `stable-diffusion-webui`, A web interface for Stable Diffusion models, motivating us to further design a new edit representation. Existing works [14], [9], [8] represent code edits as code lines labelled

[1]https://github.com/pingcap/tidb/commit/fcef061

[2]https://github.com/AUTOMATIC1111/stable-diffusion-webui/commit/9e27af7

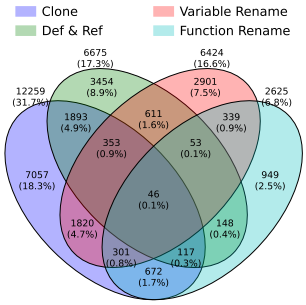

Fig. 1: Percentage of commits with edit composition.

Language	Percentage (%)
Python	17.76
Go	18.32
Java	17.03
JavaScript	20.81
TypeScript	19.20
Avg.	18.04

TABLE II: Percentage of edit hunks with multiple semantics.

with *replace*, *insert*, and *keep*. For example, for H1 in Table III, line 1 has a tag of <KEEP> and has a tag of <REPLACE>. Existing solutions follow the git-diff representation as shown in H1, H2, and H3.

However, we observe that such a representation represents different scenarios in a similar way, which can cause training confusion. For example, the representation in H1 indicates that the `if` condition (line 2) should be changed into a code block (line 3-5 in H1), which can hardly be learned to generalized to H2 and H3. Nevertheless, probing into the details of H1 indicates that the **single** hunk contains **two** edit semantics: both *insert* (line 2,3 in H1') and *replace* (line 4,5 in H1'), which are under-represented by the unified *replace* label in git diff. In other words, reformulating the edit representation as H1' enables the model to generalize the replacement at line 5 (in H1') to H2 and H3 in a far more convenient way during model training.

Empirical Study. To validate the prevalence of such multi-semantic hunks, we apply Algorithm 1 in Section IV-A to analyze the number of distinct edit semantics contained in each edit hunk from our dataset. We compute the percentage of edit hunks that encompass more than one edit semantic (e.g., code deletion and replacement within the same hunk). Table II shows that 18.04% of edit hunks involve multiple edit semantics that are not fully captured by the traditional git-diff format.

Given the observation, we propose a new edit representation with 6 editing labels in contrast to 3 traditional editing labels (i.e., *replace*, *insert*, and *keep*), further boosting the performance of neural edit locator and generator.

IV. Methodology

Figure 2 shows an overview of TRACE, which takes the code project P, the user's prior edits E_p, and optional edit description $prompt$ as input, and recommends the subsequent code edit e_{k+1} in terms of both location and content. The generated code edit serves as a new prior edit to predict the subsequent edits. TRACE orchestrates three neural network models, which function as follows:

- **Edit-composition invoker**: Given the prior edits E_p, the edit-composition invoker checks whether the last edit e_k is part of a pre-defined edit composition (e.g., variable rename and def-use update) and, if so, invokes tool services to retrieve the remaining edits within the same composition.
- **Edit locator**: Given a code window as a scope, optional prompt $prompt$, and prior edits E_p, the edit locator predicts the edit label of each line of code and each space between lines. These labels serve as indicators for (1) the editing location within the scope and (2) the edit types at those locations, facilitating the follow-up edit content generation.
- **Edit generator**: Based on the predicted edit types in the code window, the optional prompt $prompt$, and E_p, the edit generator predicts the edit content $code_a$.

Once the latest edit e_k is applied to the codebase, it is passed to the edit-composition invoker. If e_k is identified as part of a predefined edit composition, the corresponding tool service will be automatically triggered. Tools such as variable and function renaming from LSP provide both edit location and content, allowing us to skip the locator and generator steps. Other tools may only provide approximate code windows, which require further processing by the edit locator and

TABLE III: Example 2: Edit representation beyond replacement

File: modules/sd_samplers_kdiffusion.py

H1	1	`extra_params_kwargs = self.initialize(p)`
	2	`- if 'sigma_min' in inspect.signature(self.func).parameters:`
	3	`+ parameters = inspect.signature(self.func).parameters`
	4	`+ xi = x + noise * sigma_sched[0]`
	5	`+ if 'sigma_min' in parameters:`
	6	`  extra_params_kwargs['sigma_min'] = sigma_sched[-2]`
H1'	1	`extra_params_kwargs = self.initialize(p)`
	2	`+ parameters = inspect.signature(self.func).parameters`
	3	`+ xi = x + noise * sigma_sched[0]`
	4	`- if 'sigma_min' in inspect.signature(self.func).parameters:`
	5	`+ if 'sigma_min' in parameters:`
	6	`  extra_params_kwargs['sigma_min'] = sigma_sched[-2]`
H2	7	`  ...`
	8	`- if 'n' in inspect.signature(self.func).parameters:`
	9	`+ if 'n' in parameters:`
	10	`  extra_params_kwargs['n'] = len(sigma_sched) - 1`
H3	11	`  ...`
	12	`- if 'sigma_sched' in inspect.signature(self.func).parameters:`
	13	`+ if 'sigma_sched' in parameters:`
	14	`  extra_params_kwargs['sigma_sched'] = sigma_sched`

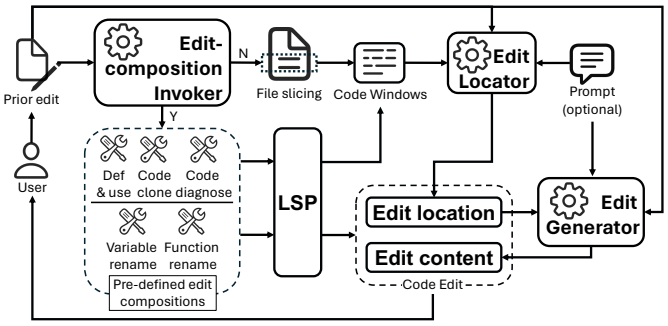

Fig. 2: Overview of TRACE: TRACE generates code edits (in terms of the edit location and the edit content) by orchestrating three models, i.e., edit composition invoker, edit locator, and edit generator. The generated code edits can serve as new prior edits to further generate new edits.

Edit Representation	R_{edit}	::=	$L_{inter}, R_{code}, L_{inter}$
Code Representation	R_{code}	::=	R_{line}
		|	$R_{line}, L_{inter}, R_{code}$
Line Representation	R_{line}	::=	L_{inline}, LoC
Line of Code	LoC	::=	one line of code
Inter-line Label	L_{inter}	::=	<NULL>
		|	<INSERT>
		|	<BLOCK-SPLIT>
Inline Label	L_{inline}	::=	<KEEP>
		|	<REPLACE>
		|	<DELETE>

Fig. 3: Edit representation of TRACE in BNF, more expressive for different code editing scenarios, with six edit labels.

generator for contextual analysis. The edit locator analyzes these windows to label each line and inter-line position based on context, determining precise editing lines and plans. Subsequently, the generator combines contextual information to produce the specific edit content for each identified location. If no composition is detected, TRACE slices the project into code windows and feeds them into the edit locator for labelling. Only code windows that receive edit-requiring labels (e.g., **<REPLACE>**, **<INSERT>**) are subsequently passed to the generator for edit content generation. In this work, we train code LLMs as the edit locator and edit generator. Edit adopted by the user will be added to prior edits for the next recommendation. Given the space limit, model training hyper-parameters and input/output examples are available at [19].

A. Edit Representation

Given an edit e, its edit representation specifies what edit operation type (e.g., insert, keep, replace, delete) is applied on each line of code in $code_b$. Generally, an edit specified in Section II can have many representations, which makes an impact on the model training effectiveness. In this work, we report a training-friendly representation for the model to distinguish different editing scenarios. We used BNF to define our designed edit representation. As shown in Figure 3, an edit representation (R_{edit}) consists of two inter-line edit labels (L_{inter}) and a code representation (R_{code}). A code representation consists of line representations (R_{line}), the edit label of a code line (L_{inline}), and a line code (LoC). The edit labels are annotated with the edit type defined as follows.

- **<KEEP>** (L_{inline}): No edit to conduct for a line;
- **<REPLACE>** (L_{inline}): A line of code is to modify;
- **<DELETE>** (L_{inline}): A line of code is to delete.
- **<NULL>** (L_{inter}): No code to insert between lines;
- **<INSERT>** (L_{inter}): New code is to insert between lines;
- **<BLOCK-SPLIT>** (L_{inter}): A block consists of continuous lines with the same edit type. This edit label is to split two blocks with different semantics.

An example. Table IV illustrates how a real edit hunk from project `localstack/localstack` [3] is translated into our representation. The first column contains inter-line labels and the second contains inline labels. Compared to the git-diff format, the representation decomposes coarse-grained differences into finer-grained edit semantics. While the git-diff format captures the entire hunk as a single replace, TRACE derives 6 edit semantics, i.e., a delete change (i.e., delete line 3); a replace change (i.e., replace line 4) and an insert change (i.e., insert between line 4, 5), etc. This granularity enables more precise generalisation, improving both edit localisation and generation (see the results in Section V-A and Section V-B).

[3]see https://github.com/localstack/localstack/commit/667c6c5.

Algorithm 1 Edit Representation Translation

1: **Input:** An edit hunk labelled in git-diff format, L
2: **Output:** An edit hunk labelled in TRACE format, L^*
3: $code\_token\_mapping = LCS(parser(L.old\_version\_code), parser(L.new\_version\_code))$
4: $line\_mapping = token2line\_mapping(code\_token\_mapping)$
5: **for** $block \in line\_mapping$ **do**
6: $\quad inter\_labels, inline\_labels = convert2label(block.old.line\_idx, block.new.line\_idx)$;
7: $\quad L^*.inter\_labels$.extend($inter\_labels$);
8: $\quad L^*.inline\_labels$.extend($inline\_labels$);
9: **end for**
10: **ASSERT** $len(L^*.inter\_label) - 1 == len(L^*.inline\_label) == len(L.old\_version\_code\_lines)$
11: **return** L^*

Algorithm 1 shows how an edit hunk from git-diff format is translated into our representation, by aligning code lines before (e.g. old line 3-6 in Table IV) and after the edit (e.g. new line 3-9). Since no identical lines exist between the two versions, direct line-level matching is infeasible. In Algorithm 1 line 3, We use Tree-sitter [24] to tokenise both versions into syntax elements with their types, code text, and line indices. Longest Common Sub-sequence (LCS) is applied to the token sequences to match elements that remain unchanged, where a match is defined as having both identical type and code text. Based on the line index of matched elements, $token2line\_mapping(\cdot)$ builds line-level alignment. Hence in Table IV, old line 3 maps to $\emptyset$, old line 4 to new line 3 and $\emptyset$ to new lines 8, 9. We then assign inline and inter-line labels according to each mapping block via $convert2label(\cdot)$: matched lines in the old version are labelled with <REPLACE>; lines in the old version that have no counterpart are labelled <DELETE>; for unmatched lines in the new version, the corresponding inter-line positions in the old version are marked with <INSERT>. Additionally, <BLOCK-SPLIT> functions as a separator between replace blocks. The rest of the inter-line spaces are labelled with <NULL>. Algorithm 1 returns hunk with enriched representation.

B. Edit-composition Invocation

1) Predefined Edit Composition: We select five predefined edit compositions[4], each is denoted as an edit set E,

- **Variable Rename:** Edits in the variable rename composition E_{var} consistently modify all occurrences of the same variable identifier across the codebase.
- **Function Rename**: Similarly, edits of this composition E_{func} replace the same function identifier;

[4]These five composition types reflect basic capabilities widely supported by mainstream LSPs and commonly observed in practice. More advanced compositions depend on specific languages and LSP implementations.

TABLE IV: Enriched edit semantic labelling: The first column denotes inter-line labels, including N (NULL), I (INSERT), and B (BLOCK-SPLIT). The second column denotes inline labels, including K (KEEP), R (REPLACE) and D (DELETE).

Enriched edit representation on hunk before edit				Hunk after edit		
N						
	K	1	def extract_tags(req_data):	1	def extract_tags(req_data):	
N						
	K	2	tags = []	2	tags = []	
N						
	D	3	- req_tags = {k: v for k, v in req_data.items() if k.startswith('Tags.member.')}			
N						
	R	4	- for i in range(int(len(req_tags.keys()) / 2)):	3	+ for i in range(1, 200):	
I				4	+ k1='Tags.member.%s.Key'%i	
				5	+ k2='Tags.member.%s.Value'%i	
	R	5	- key = req_tags['Tags.member.' + str(i + 1) + '.Key']	6	+ key = req_data.get(k1)	
B						
	R	6	- value = req_tags['Tags.member.' + str(i + 1) + '.Value']	7	+ value = req_data.get(k2)	
I				8	+ if key is None or value is None:	
				9	+ break	
	K	7	tags.append({'Key': key, 'Value': value})	10	tags.append({'Key': key, 'Value': value})	
N						
	K	8	return tags	11	return tags	
N						

- **Def-Use Propagation:** Edits to function signatures (e.g., parameter addition or type change) affect both definitions and usage sites, forming a def&use composition E_{defuse}.
- **Code Clone Update:** Edits in code clone composition E_{cc} share similar pre-edit code $code_b$ and post-edit code $code_a$.
- **Code Diagnose Fix:** Edits within diagnostic compositions E_{dia} propagate to each other via lint errors.

2) Learning Invocation: We design edit composition invoker as a multi-label classifier predicting whether the latest prior edit fits any defined composition type. We exclude prompt tuning as it is non-trivial for prompts to accurately decide when and which edit composition to invoke across hundreds of potential scenarios. Instead, we adopt an encoder model, as shown in Figure 4: given prior edits E_p, we divide the edits into the latest edit and the rest of the prior edits, feeding into the encoder. The encoder outputs a confidence score for each type of edit composition in CLS position (see Figure 4), where scores above a threshold indicate composition membership. For each edit, we adopt XML tags including <BEFORE> and <AFTER> as the instructions used for model training. Here, we exclude the code diagnostics composition from the training dataset, as LSP implementations push the diagnosis proactively upon document changes, without requiring active invocation by the Invoker. To optimize this prediction, we apply a binary cross-entropy loss over the multi-label output, treating each composition type as an independent label. For a single training instance with predicted logits $\{x_i\}$ for composition i, and ground truth labels $\{y_i\}$, the loss is defined as shown in Equation 1, here $\sigma(\cdot)$ represents the sigmoid function.

$$\mathcal{L}_{\text{BCE}} = -\sum_i \left[y_i \log \sigma(x_i) + (1 - y_i) \log(1 - \sigma(x_i)) \right] \quad (1)$$

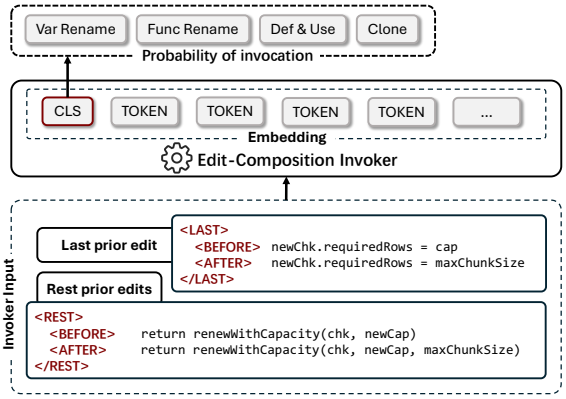

Fig. 4: Overview of edit-composition invoker

C. Edit Locator and Edit Generator

We adopt fine-tuning for both the edit locator and the generator: the encoder-only design localizes edits in a single pass without costly decoding iterations, and fine-tuning allows the enriched edit representation to be exploited more effectively than prompt-based approaches.

1) Edit Locator: Given a code window from $line_{start}$ to $line_{end}$, an optional edit description $prompt$, and selected prior edits E'_p, the edit locator predicts the edit labels op specified in Section IV-A by adopting masked language modelling (MLM) [25].

Figure 5 illustrates the details. For a code window, we place <MASK> before each line of code and <INTER-MASK> between lines. The goal of the neural locator is to recover the two types of mask tokens. We use cleaned commit messages as $prompt$ during training. We append selected edits in E_p as a way to incorporate prior edits. Edits are selected as the top-ranked result based on textual similarity, using BM25 [26].

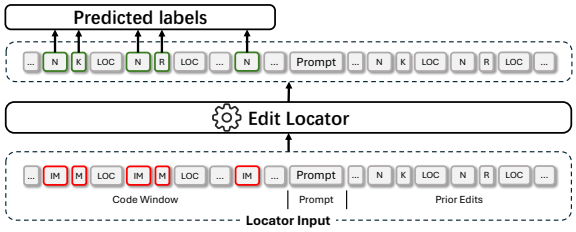

Fig. 5: Neural locator overview: encoder trained to recover the masked tokens. IM is inter-line mask, M is inline mask and LoC denotes a line of code. N denotes <NULL>, K denotes <KEEP>, R denotes <REPLACE>.

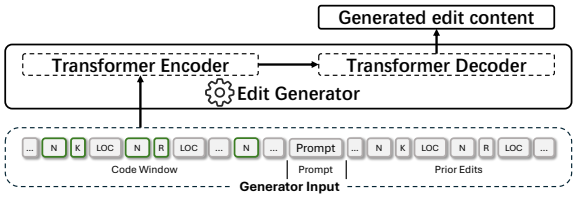

Fig. 6: Overview of neural generator. Given predicted labels like <REPLACE>, a transformer of encoder-decoder generates an edit solution. The code window, prompt, and prior edits follow the same format as shown in Figure 5.

Edits in E_p are formatted in enriched representation with the post-edit code concatenated.

TRACE locator is optimized via Cross-Entropy loss on <MASK> and <INTER-MASK> positions:

$$\mathcal{L}_{\text{CE}} = -\sum_{i=1}^{C} y_i \log p_i \tag{2}$$

As shown in Equation 2, C is the total number of classes (e.g., <KEEP>, <REPLACE>, <DELETE> for <MASK>, and <NULL>, <INSERT>, <BLOCK-SPLIT> for <INTER-MASK>), y_i denotes the one-hot gold label for class i.

2) Edit Generator: As shown in Figure 6, the generator input includes the code window $code_b$ with edit labels op, selected prior edits E'_p, and optional edit description $prompt$. Each input window only contains one hunk to edit. The generator, formulated as a sequence-to-sequence model [27] fine-tuned from an encoder-decoder architecture [7], generates k candidate edits using beam search [28].

The generator is fine-tuned using Cross-Entropy loss on the target post-edit code $code_a$, as shown in Equation 2, where C denotes the vocabulary size.

V. Experiment

We evaluate TRACE via the following research questions:

- **RQ1 (Edit Location, Section V-A)**: Can the edit locator effectively identify editable lines and their types?
- **RQ2 (Edit Generation, Section V-B)**: Can the edit generator effectively generate the edit content?
- **RQ3 (Ablation study, Section V-C)**: Can the enriched edit semantic representation effectively boost the performance of the neural locator and generator?

TABLE V: Multilingual benchmark of TRACE on 678 repositories and 5 programming languages.

Language	Dataset	#Proj	#Commit	#Locator window	#Generator Hunk
Python	Train	66	5,703	72,807	36,114
	Valid	23	815	10,066	5,096
	Test	36	1,630	20,026	10,078
Go	Train	80	4,919	66,705	32,594
	Valid	21	707	9,414	4,636
	Test	64	1,405	18,457	9,206
Java	Train	59	9,480	135,196	67,714
	Valid	8	1,354	18,813	9,611
	Test	40	2,705	38,777	19,578
Javascript	Train	77	1,826	21,724	11,022
	Valid	14	264	3,233	1,706
	Test	33	520	5,989	3,105
Typescript	Train	108	5,116	64,766	33,054
	Valid	26	734	4,743	9,416
	Test	34	1,469	18,360	9,397
Total	Train	390	27,044	361,198	180,498
	Valid	92	3,874	50,942	25,792
	Test	207	7,729	101,609	51,364

- **RQ4 (Edit-Composition Invocation, Section V-D)**: Can the edit-composition invoker be triggered accurately?
- **RQ5 (Real-world Simulation, Section V-E)**: What is the performance of TRACE in a real-life editing environment?

Given the space limit, qualitative analysis and the visualized simulation process are available at [19].

Experiment Setup: We construct a multilingual benchmark of 5 programming languages (Python, Go, Java, JavaScript, and TypeScript). We use an open-source LLM (e.g., Llama-3-8B-Instruct [29], see our anonymous site [19] for prompt design) to filter out commits with vague or multi-intent messages (e.g., adding new features while fixing unrelated bugs), as such content hinders learning the semantic link between language and edits. Commit messages are typically abbreviated and often contain irrelevant noise such as PR numbers, repeated messages or committer information, leading to a distribution shift from real-life edit descriptions. To address this, we refined retained messages for clarity (e.g., removing pull request IDs or committer emails). Our manual evaluation shows that the refined messages are more natural and concise, and LLaMA3 performs well in this task by effectively removing noise and rewriting messages into fluent descriptions.

The resulting benchmark comprises 678 top-starred repositories and a final set of 38k commits, statistics are shown in Table V. Based on the benchmark, we translate the data format for training different models to predict specific training tasks such as composition invoking, edit location, and edit generation. We adopt cross-project splits, with training and test sets from disjoint projects to prevent data leakage.

A. RQ1 (Edit location)

Benchmark: To train and evaluate the neural locator, we collected 101,609 sliding code windows from the general benchmark, each is paired with up to three relevant hunks. To avoid data leakage, hunks that overlap with the target code window are excluded. To better align with real-world

TABLE VI: RQ1: Edit location performance: TRACE uses enriched semantics; baselines use plain representations.

	Acc. (%)	Prec. (%)	Rec. (%)	F1 (%)
TRACE Locator	**93.48**	**69.61**	**67.89**	**68.71**
CoEdPilot Locator	83.97	48.42	61.75	51.91
Code Clone Detector	82.15	40.47	44.14	41.38

deployment scenarios, a single code window may contain multiple edit hunks. All models, including baselines and TRACE, take the same input: a code window, an optional prompt, and selected prior edits. The output is a label sequence.

Metric: We evaluate predicted labels using accuracy, macro-averaged precision, recall, and F1-score. Macro averaging equally weights all classes, ignoring the class imbalance.

Baseline: TRACE locator is compared with the state-of-the-art CoEdPilot locator and naive code clone detector[5] [30].

All neural models adopt the `Salesforce/codet5-large` [31], [3] encoder, fine-tuned on our dataset. This experiment precludes LSP, as it requires the simulation of the entire project. The detailed evaluation for TRACE with external tools equipped is in Section V-E.

Result: Table VI shows the locator's static performance with enriched semantics, achieving 69.61% precision and 68.71% F1 in locating edit lines. The strength of our enriched semantic locator lies in high-precision edit identification, achieving a 43.76% improvement over the best baseline. Meanwhile, the code clone detector performs well mainly for copy-paste edits, but its overall effectiveness is limited.

B. RQ2 (Edit generation)

Benchmark: Similar to the neural locator, we collected 51,364 test hunks from the general benchmark, each paired with 3 other hunks from the same commit. All generators share the same input, i.e., a code window with edit labels, an optional prompt, and selected prior edits. The target output is the post-edit code for the hunk within the window.

Metric: We generate 10 candidates per sample, ranked by confidence, and evaluate performance at Top-1, 3, 5, and 10 using exact match rate (EMR) and BLEU-4 [32]. EMR is the percentage of samples with an exact match in the Top-k, while BLEU-4 takes the highest score among them.

Baseline: We compare our enriched edit semantic generator with 3 state-of-the-art edit generators: CoEdPilot [14], GrACE [8] and CCT5 [9]. All models adopt `Salesforce/codet5-base` as the base model and are fine-tuned on our dataset.

Result: Table VII shows the static generation performance of edit generation with Top-10 candidates. Our generator with enriched edit semantics achieves the best performance in generating edit options: the Top-1 candidate has an exact match rate of 48.02%, yielding a notable boost of 11.16% compared with the state-of-the-art generator models. In the Top-10 options, an exact match can be found in more than 57% of the cases, suggesting potential gains in user efficiency.

[5]The code clone detector is included based on the observation that developers often locate relevant code by searching for similar code snippets.

TABLE VII: RQ2: static generation performance: TRACE uses enriched semantics; baselines use plain representations.

Model	**Metric**	**@1**	**@3**	**@5**	**@10**
TRACE generator	**EMR**	**48.02**	**53.44**	**55.04**	**57.05**
	BLEU	**69.68**	**73.55**	**74.73**	**76.10**
CoEdPilot generator	**EMR**	43.20	48.92	50.61	52.70
	BLEU	64.90	69.64	70.94	72.48
GrACE	**EMR**	39.54	46.30	48.45	50.88
	BLEU	61.89	68.47	70.33	72.36
CCT5	**EMR**	39.76	45.76	47.77	51.45
	BLEU	62.91	67.88	69.43	72.43

TABLE VIII: RQ3: Ablation study.

Model	Locator				Generator				
	Acc. (%)	Prec. (%)	Rec. (%)	F1 (%)		@1	@3	@5	@10
Enriched	**93.48**	**69.61**	67.89	**68.71**	EMR	**48.02**	**53.44**	**55.04**	**57.05**
					BLEU	**69.68**	**73.55**	**74.73**	**76.10**
Plain	90.78	60.76	**72.68**	65.41	EMR	44.71	50.08	51.70	53.61
					BLEU	65.82	70.14	71.41	72.81

C. RQ3 (Ablation study)

Benchmark: We evaluate the edit locator and generator using benchmarks from Sections V-A and V-B.

Metric: Same metrics as in Section V-A and Section V-B.

Baseline: We compare our enriched edit semantic representation with the 3-label one by fine-tuning models of the same size on the same dataset, differing only in representation.

Result: As shown in Table VIII, our enriched edit semantics achieve 69.61% precision, a 14.57% improvement over the plain representation. We observed that the plain semantics locator achieves slightly higher recall, likely due to having fewer labels, which simplifies classification with coarser decision boundaries and fewer misclassifications. However, for edit location recommendation, we prioritize precision over recall, as developers benefit more from accurate suggestions than from broader but less precise coverage. It is noteworthy that enriched semantics improve generator performance by 7.40% over plain semantics, mainly by offering clearer edit instructions for target code windows.

D. RQ4 (Edit-composition invocation)

In this experiment, we leverage the facilities of LSPs [33], [34], [35], [36] as the IDE edit composition.

Benchmark: Given a commit of N edit hunks, we randomly select one target hunk H_t and up to 2 background hunks H_{b1}, H_{b2}. We configure the project state such that H_t, H_{b1}, H_{b2} are applied (post-edited) while the remaining $N-3$ hunks remain unapplied (pre-edited), simulating a partial commit completion scenario. We invoke LSP services (rename, find reference, find clone) at H_t's location. If the service returns edit locations matching any of the remaining $N-3$ unedited hunks (determined by line index overlap), we label this as a positive sample with H_t, H_{b1}, H_{b2} as input and the corresponding LSP service type as output. The dataset contains 8,534 training, 1,294 validation, and 2,499 test samples.

Metrics: We use macro-averaged precision, recall, and F1-score to assess the edit composition invocation performance.

TABLE IX: RQ4: The performance of Invoker.

		Precision (%)	Reccall (%)	F1 (%)
Edit-Composition Invoker	Variable Rename	91.15	98.59	94.72
	Function Rename	98.71	97.44	98.07
	Def & use	84.06	87.90	85.94
	Clone	95.88	94.59	95.23
	Average	**92.45**	94.63	**93.49**
Blindly invoking		22.01	**100.00**	35.50
Randomly invoking		21.85	50.26	29.71

Baseline: We compare our method against two baselines: blindly invoking all LSP services, and randomly invoking one.
Result: Table IX shows the performance of the edit-composition invoker, which achieves an F1 score of 93.49%, substantially outperforming both blindly and randomly invoking LSP functions. The results indicate that the edit composition can be effectively invoked.

E. RQ5 (Real-world simulation)

Benchmark: 500 commits (100 for each language) are randomly selected from the test set in the general benchmark, comprising a total of 3,211 edit hunks.
Simulation process: Compared with the previous two research questions (Section V-A and Section V-B), real-world editing scenarios are much more challenging, because: (1) the majority of the code remains unchanged (2) with fewer prior edits available; and (3) the impact of a single edit is limited—it rarely propagates to all edits within a commit and, in some cases, does not propagate at all. Hence, to comprehensively evaluate the actual performance in a real-world editing scenario, we simulate the process of a programmer making a commit by editing each hunk until the old version is transformed into the new version. We define the simulation process as 4 stages. 1) **Initialization**: For a given commit, we check out the project to its pre-commit version and use git diff to identify all edit hunks. The first hunk listed by git diff is designated as the initial edit, which is applied to obtain the starting project state. 2) **Location prediction and selection**: The process then iterates over the remaining edits. At each step, the locator model predicts candidate locations for the next edit based on the commit message and prior edits. Candidates are ranked by confidence scores, either produced by the model or set to a default value of 1.0 for locations obtained via LSP services. A predicted location is considered a match to a ground-truth edit if it has more than 50% line overlap with the ground-truth hunk. If a match is found, it is passed to the generator; otherwise, the virtual programmer randomly selects one remaining ground-truth location. 3) **Edit generation and application**: Given the selected location, the generator produces 10 candidate edits, which are compared with the ground truth. The virtual programmer applies the ground-truth edit content at the selected location to the project before proceeding. This quality control mechanism prevents the simulation from continuing from erroneous states, emulating human oversight in interactive editing. 4) **Iteration**: Steps 2 and 3 repeat until all edits in the commit are simulated and applied. A video of this simulation process is available at [19].

TABLE X: RQ5: Real-world edit simulation performance.

Model	Locating				Generation		
	Match rate (%)			Time (s)	BLEU-4 distribution (%)		
	@1	@3	@5		100	50~100	< 50
TRACE	**35.18**	**42.07**	**44.24**	3.27	**49.23**	24.32	26.45
TRACE w/o Invoker	32.52	39.80	42.23	3.47	**49.23**	**24.52**	**26.25**
Enriched semantic	32.56	40.71	43.39	3.82	**49.23**	**24.52**	**26.25**
Plain semantic	31.05	36.35	37.57	3.81	45.41	23.67	30.92
CoEdPilot	11.47	24.41	29.25	3.80	43.17	23.81	33.02
CCD	15.99	15.99	15.99	**2e-4**	45.35	23.62	31.03

Metric: For localization, we report the **Top-K match rate (MR@K)** and the **time cost**. MR@K denotes the percentage of predictions with at least one correct location in the Top-K. Time cost reflects the average latency of location prediction. Both metrics are affected by Invoker and Locator performance; component failures lead to lower MR@K and higher time costs. For generation, we analyze the **BLEU-4 score distribution** in three bands: 100 (directly usable for user), 50–100 (minor edits needed), and <50 (likely rejected). BLEU-4 scores are affected by both Invoker and Generator; component failures lead to score drops. **Acceptance rate @K** denotes the percentage of Top-K suggestions containing at least one match with the gold location and a BLEU-4 of 100, which is affected by all three components.
Baselines: TRACE composes a locator and a generator with enriched edit semantics, integrated with LSP [33], [34], [35], [36] and Invoker. We compare TRACE to baselines:

1) **TRACE w/o Invoker**: TRACE without Invoker, blindly invoking LSP services;
2) **Enriched semantic**: Neural locator and generator, both with the enriched edit semantics (6-label representation);
3) **Plain semantic**: Neural locator and generator, both of plain edit semantics (3-label representation);
4) **CoEdPilot**: Locator and generator from CoEdPilot, models re-trained on the same dataset and architecture as TRACE;
5) **CCD**: Code clone detector as locator and neural generator of plain edit semantics.
6) **Cursor**: Since Cursor of version 0.46 lacks APIs for large-scale simulation, to simulate a commit, each time we manually apply an edit to the project. If this edit triggers Cursor's Tab recommendation, we follow the single suggested edit and mark it as Top-1. Otherwise, we input the prior edit and edit description into the Cursor Chat, which provides multiple edit suggestions. We rank all suggestions by their proximity to the last applied edit and evaluate whether the ground truth appears in the Top-K ranked recommendations.

Result: Table X shows the performance in the simulation. Our TRACE outperforms all baselines in locating performance, with over 35% of cases yielding a useful recommendation at the Top-1 location. Compared to the state-of-the-art CoEdPilot, TRACE achieves a notable 206.71% improvement in MR@1. Integrating LSP and Invoker improves MK@1 performance by over 8.05% over pure neural solutions, and cuts locating time by 14.40%. Meanwhile, blindly invoking the LSP service

TABLE XI: RQ5: Acceptance rate in real-world simulation.

Model	Acceptance rate (%)		
	@1	@3	@5
TRACE	**25.71**	**28.54**	**29.55**
TRACE w/o Invoker	24.70	27.73	28.74
Enriched	24.49	27.73	28.95
Plain	21.46	25.91	26.52
CoEdPilot	8.30	17.81	20.65
CCD	12.96	12.96	12.96
Cursor	24.22	26.19	27.20

is prone to introducing false positive edits, reducing performance to that of neural-only solutions. The enriched semantic approach shows a 15.49% improvement in MR@5 over plain semantic and a 47.83% improvement over CoEdPilot, confirming the effectiveness of our enriched edit representation as an additional component alongside Edit-composition Invoker and LSP. Note that the code clone detector has the same MR@K, as it cannot rank results and marks all as Top-1. Although code clone detection excels in time efficiency, its recommendation quality remains insufficient for practical use. We also observed a time discrepancy between CoEdPilot and its original work. The higher time cost arises as the locator scans more sliding windows (n) per query and selects from m prior edits per window via a neural network, adding $O(mn)$ time complexity.

For the generator, our enriched semantic model generates helpful edit solutions (BLEU-4>50) in over 73% of the cases, with around 50% of them achieving BLEU-4=100. Compared to the state-of-the-art CoEdPilot, TRACE achieves a performance boost of 14.04% in terms of high-quality suggestion (BLEU-4=100) and a 19.90% drop in low-quality suggestion (BLEU-4<50). Compared to plain semantic and the state-of-the-art CoEdPilot, TRACE achieves 8.41% and 14.04% improvements respectively in high-quality suggestions (BLEU-4=100), while reducing low-quality suggestions (BLEU-4<50) by 16.89% and 19.90% respectively, confirming the effectiveness of the TRACE generator and enriched edit representation.

Table XI shows the acceptance rate of baselines, where TRACE is on par with Cursor while achieving a 6% improvement. Cursor's `Tab` feature offers fast and accurate suggestions but triggers conservatively, while its Chat interface struggles to locate cross-file edits proactively. As a result, only 8.82% of accepted edits are cross-file. In contrast, TRACE leverages LSP integration to support efficient cross-file localization, with 38.46% of accepted edits being cross-file edits.

VI. User Study

To validate TRACE in real-world use, we implement TRACE as a VS Code extension and design this user study. Video recordings and user interaction data are provided at our homepage [19].

Baseline: We compare TRACE with: (1) CoEdPilot as the state-of-the-art subsequent edit suggestion system, and (2) Cursor [12] (version 0.46), a popular AI-powered IDE. During evaluation, we impose no restrictions on Cursor's functionality, and all tools can access IDE LSP services to ensure fairness.

Participant: We recruit 24 Computer Science students from three universities in both China and Singapore. All participants completed a pre-study questionnaire on their experience with programming and AI-assisted tools. See our website for more demographic details [19]. Participants are stratified into three balanced groups based on their self-reported programming proficiency and AI tool experience, to ensure comparable skill levels across groups. The experimental group (EG) uses TRACE, while control group 1 (CG1) uses CoEdPilot and control group 2 (CG2) uses Cursor.

TABLE XII: Performance of EG (TRACE), CG1 (CoEdPilot) and CG2 (Cursor), the time cost in minutes.

EG	T1	T2	T3	CG1	T1	T2	T3	CG2	T1	T2	T3
P1	3.32	7.18	8.78	**P9**	3.55	8.42	15.60	**P17**	15.68	11.25	4.87
P2	5.22	5.78	5.15	**P10**	3.58	13.03	10.03	**P18**	15.87	19.28	3.42
P3	3.83	13.37	3.50	**P11**	6.62	13.12	9.60	**P19**	17.10	18.20	4.83
P4	3.13	8.77	4.47	**P12**	7.90	13.07	6.70	**P20**	17.82	14.55	3.05
P5	3.52	11.18	4.68	**P13**	4.25	12.55	7.83	**P21**	17.08	20.12	2.98
P6	2.92	11.00	5.62	**P14**	3.92	11.72	5.52	**P22**	13.68	30.00	2.82
P7	4.73	8.52	3.67	**P15**	5.18	16.38	4.57	**P23**	16.98	12.03	2.93
P8	2.97	11.37	3.20	**P16**	8.65	18.08	18.32	**P24**	14.63	7.30	3.10
Avg.	3.70	9.65	4.88	**Avg.**	5.46	13.30	9.77	**Avg.**	16.11	16.59	3.50

Editing task: Participants are asked to reproduce edits from 3 real-world GitHub commits, each under a 30-minute time budget. Selected tasks represent common editing scenarios, with all necessary domain knowledge provided, ensuring completion by participants with general programming experience. Performance is measured by task completion and time.

- **Task 1**: Refactor the `if` condition from `if "http" in XX` to `if XX.startswith("http")` to improve the robustness of string matching [37], requiring 8 edits across 5 files;
- **Task 2**: Add a `train_data_size` flag to the BERT data pipeline, allowing users to limit training samples and triggering changes along the call chain [38], requiring 9 edits across 2 files along the call chain;
- **Task 3**: Add the `noise_shape` and `seed` arguments to the Dropout layer API in Keras, enhancing control over the dropout mask shape and randomness seed [39], requiring 5 edits in a single file.

Setup: Participants first complete a warm-up tutorial to familiarize themselves with the assigned tool. For each task, they receive 1) the project code, 2) background knowledge like code functionality and API usage, 3) a detailed edit description, 4) the first edit as a hint, and 5) test cases for validating edits via execution. The initial edit for each task is selected to provide context and guide subsequent edits: for Task 1, any edit can be randomly chosen due to the uniform editing pattern; for Task 2, the call chain entry point; and for Task 3, the documentation for new arguments.. Screens are recorded for analysis.

Metric: We evaluate user study via three metrics: **average time cost** for user efficiency, **Wilcoxon Signed Rank Test p-values** for statistical significance between groups ($p < 0.05$ denotes significance), and **Effect Size** (r) to quantify the difference magnitude ($r \geq 0.5$ denotes a large effect).

Result: Participants' performance in completing 3 editing tasks is shown in Table XII, with the following observations:

1) **Task 1**: EG outperforms both CG1 ($p = 0.0781, r = 0.62$) and CG2 ($p = 0.0078, r = 0.94$, statistically significant);

2) **Task 2**: EG significantly outperforms both CG1 ($p = 0.0156, r = 0.85$) and CG2 ($p = 0.0234, r = 0.80$);
3) **Task 3**: CG2 significantly outperforms both EG ($p = 0.0390, r = 0.72$) and CG1 ($p = 0.0078, r = 0.94$).

To analyze user performance, we first instrumented the extension to monitor user actions in real-time, tracking LSP trigger events from user edits and recording user responses to recommendations (accept, reject, or modify). Second, we manually analyzed video recordings with three researchers, focusing on sessions with notably fast/slow task completion times, plus randomly selected sessions with typical completion times. Analysis examined user responses to edit recommendations and behaviors during test failures, with cross-validation among researchers. Based on this analysis, we provide the following explanations:

Why do TRACE and CoEdPilot outperform Cursor in Task 1? Both TRACE (EG) and CoEdPilot (CG1) support project-wide edit localization across files. TRACE further leverages LSP-based clone detection for efficient cross-file propagation. In contrast, Cursor (CG2) relies on users to specify files, which is time-consuming when relevant files are unknown to users. Additionally, Multi-file rewriting in Cursor is slow and disruptive to mental flow.

Why do TRACE outperforms both CoEdPilot and Cursor in Task 2? Task 2 involves 9 edits across 2 files, which are distant but syntactically coherent. Still taking advantage of tool deduction, TRACE can identify the subsequent edit location via the LSP service, avoiding the exhaustive file scanning, as in CoEdPilot. In contrast, Cursor users struggle due to its reliance on full-file rewriting and limited cross-file localization ability. For example, user P22 manually searched across files but ended up in the wrong one and exhausted the time budget.

Why does Cursor outperform TRACE and CoEdPilot in Task 3? Given the shared context among 5 edits, Cursor allows users to quickly trigger the next edit recommendation via `Tab`, or generate all correct edits in a single rewrite via Chat, which significantly improves editing efficiency. Despite lacking edit compositions in this task, TRACE still outperforms CoEdPilot by fewer false positive suggestions with its improved predicting performance.

How do users respond to false positives (over-trust phenomenon)? Video analysis reveals an *over-trust* phenomenon in nearly all users, regardless of their AI tool or programming experience. During warm-up, over 92% of participants quickly gained confidence after a few correct predictions, leading to less caution in accepting recommendations. When test cases failed, users took considerable time to revisit prior decisions, damaging performance. We believe that improving human-computer interaction (e.g., intuitive undo/review) and explainable AI (e.g., rationale behind suggestions) is more crucial than further boosting model accuracy, and we plan to explore this direction.

VII. Related Work

Edit Localization identifies editable regions based on prior edits, yet most works focus on fault localization. Spectrum-based (SBFL) [40], [41], [42] and mutation-based (MBFL) [43], [44] methods localize faults via test outcomes or code mutations. Neural models like Toggle [45] and LLMAO [46] apply attention or transformers to rank suspicious code. General edit localization is more challenging and underexplored, often relying on static tools. LASE [47] uses heuristics to extract edit patterns; CCDemon [48] leverages clone detection. However, these methods capture limited propagation types with low precision and provide no actionable instructions.

Code Edit Generation: LLMs are widely applied in software engineering for code [49], [50], comment [51] and test generation [52], [53], [54], [55], [56], [57]. Among these, code edit generation is especially demanded [58], aiming to suggest edits for a given snippet. Codit [59] proposed a tree-based model; Recoder [60] fused code, AST, and paths. CURE [61], CoditT5 [10], and CCT5 [9] introduced edit-oriented pre-training. Overwatch [62] modeled temporal edit sequences, while GrACE [8] encodes both target and prior edits in prompts. Our model further leverages enriched edit semantics for finer control and efficient learning from prior edits.

Tool Invocation: Recent advances have integrated static analysis tools for coding tasks. Pei *et al.* [63] adopted LSP to retrieve function signatures for function call infilling. ContextModule [64] and Blinn *et al.* [65] extended this approach for more general code completion task. Recent studies [66], [67], [68] also combine LLMs with static analysis to improve code quality, including fixing vulnerabilities and readability issues. However, these approaches do not directly support interactive code editing or predict subsequent edit locations. For editing tasks, CodePlan [69] employs dependency graphs and LLM-based planning for repository-level tasks, while MarsCode Agent [70] introduces multi-agent collaboration with LSP services and code knowledge graphs for bug fixing. These works differ from TRACE in several key aspects: they are all agentic solutions focused on solving SWE-bench-like tasks [23], where, given a project, editing requirements, and tests, they automatically complete edits to make the modified project pass the tests. Compared to TRACE's task, this type of task has lower latency requirements. Additionally, although they all utilize static tools, they serve merely as retrieval mechanisms to provide additional context rather than for edit localization. Moreover, MarsCode Agent's use of LSP functionality is relatively limited.

VIII. Conclusions

This paper introduces TRACE, a subsequent code editing solution that effectively captures the coherence of project-wide code edits. TRACE proposes Invoker, which integrates LSP to capture edit composition and introduces enriched edit semantics for more accurate representation. In our experiment, TRACE significantly improves edit localization and generation while demonstrating high performance in interactive editing settings, establishing itself as a new state-of-the-art solution for the end-to-end code editing task.

REFERENCES

[1] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, *et al.*, "Codebert: A pre-trained model for programming and natural languages," *EMNLP*, 2020.

[2] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, *et al.*, "Graphcodebert: Pre-training code representations with data flow," *The International Conference on Learning Representations*, 2020.

[3] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[4] GitHub, "GitHub Copilot," 2023.

[5] OpenAI, "Chatgpt." https://openai.com/chatgpt, 2021. Accessed on March 29, 2023.

[6] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 180–190, IEEE, 2013.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[8] P. Gupta, A. Khare, Y. Bajpai, S. Chakraborty, S. Gulwani, A. Kanade, A. Radhakrishna, G. Soares, and A. Tiwari, "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, (New York, NY, USA), p. 1483–1495, Association for Computing Machinery, 2023.

[9] B. Lin, S. Wang, Z. Liu, Y. Liu, X. Xia, and X. Mao, "Cct5: A code-change-oriented pre-trained model," *arXiv preprint arXiv:2305.10785*, 2023.

[10] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "CoditT5: Pretraining for source code and natural language editing," in *International Conference on Automated Software Engineering*, 2022.

[11] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (Los Alamitos, CA, USA), pp. 443–455, IEEE Computer Society, nov 2021.

[12] "Cursor - The AI Code Editor." https://www.cursor.com/, 2025. [Accessed 25-02-2025].

[13] "Introducing Copilot Edits (preview)." https://code.visualstudio.com/blogs/2024/11/12/introducing-copilot-edits/, 2024. [Accessed 03-08-2024].

[14] C. Liu, Y. Cai, Y. Lin, Y. Huang, Y. Pei, B. Jiang, P. Yang, J. S. Dong, and H. Mei, "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 2024, (New York, NY, USA), p. 466–478, Association for Computing Machinery, 2024.

[15] Z. Xu, S. Jain, and M. Kankanhalli, "Hallucination is inevitable: An innate limitation of large language models," *arXiv preprint arXiv:2401.11817*, 2024.

[16] C.-Y. Su and C. McMillan, "Do code llms do static analysis?," *arXiv preprint arXiv:2505.12118*, 2025.

[17] "Git - git-diff Documentation." https://git-scm.com/docs/git-diff, 2024. [Accessed 12-09-2024].

[18] "Visual Studio Code Extension API." https://code.visualstudio.com/api, 2024.

[19] "TRACE — sites.google.com." https://sites.google.com/view/code-trace, 2024. [Accessed 02-08-2024].

[20] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," *Advances in Neural Information Processing Systems*, vol. 37, pp. 50528–50652, 2024.

[21] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604, 2024.

[22] K. Zhang, J. Li, G. Li, X. Shi, and Z. Jin, "Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges," *arXiv preprint arXiv:2401.07339*, 2024.

[23] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?," *arXiv preprint arXiv:2310.06770*, 2023.

[24] "Tree-sitter Introduction — tree-sitter.github.io." https://tree-sitter.github.io/tree-sitter/, 2024. [Accessed 01-08-2024].

[25] J. Salazar, D. Liang, T. Q. Nguyen, and K. Kirchhoff, "Masked language model scoring," *arXiv preprint arXiv:1910.14659*, 2019.

[26] Z. Shi, J. Keung, and Q. Song, "An empirical study of bm25 and bm25f based feature location techniques," in *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices*, pp. 106–114, 2014.

[27] I. Sutskever, "Sequence to sequence learning with neural networks," *arXiv preprint arXiv:1409.3215*, 2014.

[28] M. Freitag and Y. Al-Onaizan, "Beam search strategies for neural machine translation," in *Proceedings of the First Workshop on Neural Machine Translation*, (Vancouver), pp. 56–60, Association for Computational Linguistics, Aug. 2017.

[29] AI@Meta, "Llama 3 model card," 2024.

[30] "GitHub - rapidfuzz/RapidFuzz." https://rapidfuzz.github.io/RapidFuzz/, 2024. [Accessed 12-09-2024].

[31] "Salesforce/codet5-large · Hugging Face — huggingface.co." https://huggingface.co/Salesforce/codet5-large, 2024. [Accessed 01-08-2024].

[32] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "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, 2002.

[33] "microsoft/pyright: Static Type Checker for Python." https://github.com/microsoft/pyright, 2024. [Accessed 22-03-2025].

[34] "tools/gopls at master." https://github.com/golang/tools/tree/master/gopls, 2024. [Accessed 22-03-2025].

[35] "eclipse-jdtls/eclipse.jdt.ls: Java language server." https://github.com/eclipse-jdtls/eclipse.jdt.ls, 2024. [Accessed 22-03-2025].

[36] "typescript-language-server/typescript-language-server: TypeScript & JavaScript Language Server." https://github.com/typescript-language-server/typescript-language-server, 2024. [Accessed 22-03-2025].

[37] "Fix up if http in : to be more sensible startswiths, AUTOMATIC1111/stable-diffusion-webui." https://github.com/AUTOMATIC1111/stable-diffusion-webui/commit/0afbc0c2355ead3a0ce7149a6d678f1f2e2fbfee, 2024. [Accessed 12-09-2024].

[38] "Add a flag to control the number of train examples. tensorflow/models." https://github.com/tensorflow/models/commit/1c89b792ccdb53dd0cc2504f3bce502e5f0aa4e5, 2024. [Accessed 12-09-2024].

[39] "Add noise_shape and seed to Dropout layer API. keras-team/keras." https://github.com/keras-team/keras/commit/8c0c3774e6cf88704f685784f8baba9694220d4d, 2024. [Accessed 12-09-2024].

[40] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pp. 89–98, IEEE, 2007.

[41] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 23–32, IEEE, 2011.

[42] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 1, pp. 449–456, IEEE, 2007.

[43] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 153–162, IEEE, 2014.

[44] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[45] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, "A deep dive into large language models for automated bug localization and repair," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1471–1493, 2024.

[46] A. Z. Yang, C. Le Goues, R. Martins, and V. Hellendoorn, "Large language models for test-free fault localization," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pp. 1–12, 2024.

[47] J. Jacobellis, N. Meng, and M. Kim, "Lase: An example-based program transformation tool for locating and applying systematic edits," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 1319–1322, IEEE, 2013.

[48] Y. Lin, X. Peng, Z. Xing, D. Zheng, and W. Zhao, "Clone-based and interactive recommendation for modifying pasted code," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 520–531, 2015.

[49] A. Ni, S. Iyer, D. Radev, V. Stoyanov, W.-t. Yih, S. Wang, and X. V. Lin, "Lever: Learning to verify language-to-code generation with execution," in *International Conference on Machine Learning*, pp. 26106–26128, PMLR, 2023.

[50] Y. Cai, Z. Hou, D. Sanán, X. Luan, Y. Lin, J. Sun, and J. S. Dong, "Automated program refinement: Guide and verify code large language model with refinement calculus," *Proceedings of the ACM on Programming Languages*, vol. 9, no. POPL, pp. 2057–2089, 2025.

[51] Y. Cai, Y. Lin, C. Liu, J. Wu, Y. Zhang, Y. Liu, Y. Gong, and J. S. Dong, "On-the-fly adapting code summarization on trainable cost-effective language models," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[52] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 919–931, IEEE, 2023.

[53] B. Qi, Y. Lin, X. Weng, Y. Huang, C. Liu, H. Sun, and J. S. Dong, "Intention-driven generation of project-specific test cases," *arXiv preprint arXiv:2507.20619*, 2025.

[54] X. Ren, X. Ye, Y. Lin, Z. Xing, S. Li, and M. R. Lyu, "Api-knowledge aware search-based software testing: where, what, and how," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1320–1332, 2023.

[55] Y. Lin, Y. S. Ong, J. Sun, G. Fraser, and J. S. Dong, "Graph-based seed object synthesis for search-based unit testing," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1068–1080, 2021.

[56] Y. Lin, J. Sun, G. Fraser, Z. Xiu, T. Liu, and J. S. Dong, "Recovering fitness gradients for interprocedural boolean flags in search-based testing," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 440–451, 2020.

[57] R. Liu, X. Teoh, Y. Lin, G. Chen, R. Ren, D. Poshyvanyk, and J. S. Dong, "Guipilot: A consistency-based mobile gui testing approach for detecting application-specific bugs," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 753–776, 2025.

[58] A. Alaboudi and T. D. LaToza, "Edit-run behavior in programming and debugging," in *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–10, IEEE, 2021.

[59] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1385–1399, 2022.

[60] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "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*, ESEC/FSE 2021, (New York, NY, USA), p. 341–353, Association for Computing Machinery, 2021.

[61] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1161–1173, 2021.

[62] Y. Zhang, Y. Bajpai, P. Gupta, A. Ketkar, M. Allamanis, T. Barik, S. Gulwani, A. Radhakrishna, M. Raza, G. Soares, and A. Tiwari, "Overwatch: Learning patterns in code edit sequences," *Proc. ACM Program. Lang.*, vol. 6, oct 2022.

[63] H. Pei, J. Zhao, L. Lausen, S. Zha, and G. Karypis, "Better context makes better code language models: A case study on function call argument completion," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, pp. 5230–5238, 2023.

[64] Z. Guan, J. Liu, J. Liu, C. Peng, D. Liu, N. Sun, B. Jiang, W. Li, J. Liu, and H. Zhu, "Contextmodule: Improving code completion via repository-level contextual information," *arXiv preprint arXiv:2412.08063*, 2024.

[65] A. Blinn, X. Li, J. H. Kim, and C. Omar, "Statically contextualizing large language models with typed holes," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 468–498, 2024.

[66] S. Blyth, S. A. Licorish, C. Treude, and M. Wagner, "Static analysis as a feedback loop: Enhancing llm-generated code beyond correctness," *arXiv preprint arXiv:2508.14419*, 2025.

[67] Z. Li, S. Dutta, and M. Naik, "Iris: Llm-assisted static analysis for detecting security vulnerabilities," *arXiv preprint arXiv:2405.17238*, 2024.

[68] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.

[69] R. Bairi, A. Sonwane, A. Kanade, A. Iyer, S. Parthasarathy, S. Rajamani, B. Ashok, and S. Shet, "Codeplan: Repository-level coding using llms and planning," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 675–698, 2024.

[70] Y. Liu, P. Gao, X. Wang, J. Liu, Y. Shi, Z. Zhang, and C. Peng, "Marscode agent: Ai-native automated bug fixing," *arXiv preprint arXiv:2409.00899*, 2024.