EditFlow: Benchmarking and Optimizing Code Edit Recommendation Systems via Reconstruction of Developer Flows

CHENYAN LIU, Shanghai Jiao Tong University, China and National University of Singapore, Singapore
YUN LIN[*], Shanghai Jiao Tong University, China
JIAXIN CHANG, Shanghai Jiao Tong University, China
JIAWEI LIU, Shanghai Jiao Tong University, China
BINHANG QI, National University of Singapore, Singapore
BO JIANG, Bytedance Network Technology, China
ZHIYONG HUANG, National University of Singapore, Singapore
JIN SONG DONG, National University of Singapore, Singapore

Large language models (LLMs) for code editing have achieved remarkable progress, yet recent empirical studies reveal a fundamental disconnect between *technical accuracy* and *developer productivity*. Despite their strong benchmark performance, developers complete tasks 19% slower when using AI assistance, with over 68.81% of recommendations disrupting their mental flow. This misalignment stems from the use of static commit snapshots that lack temporal information, causing models to optimize for end results rather than the incremental, context-sensitive steps that align with developers' natural reasoning process.

To bridge this gap, we present **EditFlow**, which benchmarks and optimizes subsequent code edit recommendation systems through the reconstruction of developer editing flows. EditFlow addresses three key challenges. First, collecting edit-order data that reflects developers' flow is inherently difficult: manual annotation introduces prohibitive overhead, while development logs capture only single trajectories instead of all plausible editing flows. Second, benchmarking recommendation performance against developers' ongoing editing flow requires a *digital-twin-like simulation* that can faithfully simulate the editing process. Third, existing heterogeneous systems vary drastically in scale and architecture, posing challenges for developing a unified optimization strategy that endows all models with *mental-flow awareness* regardless of design or capability.

To overcome these challenges, we propose three tightly coupled components: (1) a **prompt auto-tuning mechanism** that learns an optimized prompt for inferring the relative order between two edits, (2) a **digital twin** that replays reconstructed edit sequences to simulate developers' editing process, and (3) **EditFlow**, a **unified optimization strategy** that optimizes the flow continuity of subsequent edit suggestions based on developers' ongoing flow.

Evaluations across diverse benchmarks, including manually annotated commits, real-world industrial code, and open-source repositories, show that EditFlow improves order reconstruction accuracy by 63.81%, reduces

[*]Corresponding author.

Authors' Contact Information: Chenyan Liu, chenyan@u.nus.edu, Shanghai Jiao Tong University, Shanghai, China and National University of Singapore, Singapore, Singapore; Yun Lin, lin_yun@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Jiaxin Chang, cjx001234@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Jiawei Liu, amberwabi2003@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Binhang Qi, qibh@nus.edu.sg, National University of Singapore, Singapore, Singapore; Bo Jiang, jiangbo.jacob@bytedance.com, Bytedance Network Technology, Beijing, China; Zhiyong Huang, huangzy@comp.nus.edu.sg, National University of Singapore, Singapore, Singapore; Jin Song Dong, dcsdjs@nus.edu.sg, National University of Singapore, Singapore, Singapore.

ACM 2475-1421/2026/4-ART141
https://doi.org/10.1145/3798249

flow violations by over 75%, and boosts recommendation precision by 66.99%. A user study with 32 developers further demonstrates 25.11% faster task completion and significantly higher perceived recommendation quality. To the best of our knowledge, EditFlow is the first to evaluate and optimize code edit recommendation systems from the perspective of developers' mental flow, establishing *flow-awareness* as a new dimension for advancing human-AI code collaboration.

CCS Concepts: • **Software and its engineering** → **Maintaining software**; **Software maintenance tools**; • **Computing methodologies** → **Artificial intelligence**.

Additional Key Words and Phrases: code edit recommendation, developer mental flow, large language models

ACM Reference Format:
Chenyan Liu, Yun Lin, Jiaxin Chang, Jiawei Liu, Binhang Qi, Bo Jiang, Zhiyong Huang, and Jin Song Dong. 2026. EditFlow: Benchmarking and Optimizing Code Edit Recommendation Systems via Reconstruction of Developer Flows. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 141 (April 2026), 28 pages. https://doi.org/10.1145/3798249

1 Introduction

Large language models (LLMs) for code have achieved remarkable progress, supporting tasks from code completion [26] to documentation synthesis [13]. As these capabilities mature[1], recent efforts focus on integrating LLMs into developers' real-time workflows through *subsequent edit recommendation*: suggesting the next plausible code edit immediately after a developer performs one. Tools such as Cursor [2], Claude Code [6], and CoEdPilot [36] exemplify this shift by offering proactive, context-sensitive suggestions. These systems support an interactive loop where model outputs are accepted, modified, or dismissed by the developer, with the ultimate goal of reducing effort during code construction and maintenance.

However, recent empirical evidence suggests that this promise is far from guaranteed. The controlled trial conducted by Becker et al. [9] shows that, despite the strong standalone benchmark performance [25] of Claude models (Claude 3.5 Sonnet achieving 92.0% accuracy on HumanEval [7] and Claude 3.7 Sonnet reaching 70.3% on SWE-Bench Verified [5]), developers using Cursor Pro integrated with these models completed tasks **19% slower** than when working unaided. This striking result reveals a fundamental disconnect: **accuracy is not equivalent to productivity**. A key factor underlying this disconnect is developers' *mental flow*. Mental flow is a well-established psychological construct defined as a mental state in which a person performing an activity is fully immersed in a feeling of energized focus, full involvement, and enjoyment [15]. It is also a core determinant of developer productivity in both academic and industrial frameworks [20, 42, 44]. Empirical studies consistently show that maintaining uninterrupted flow yields substantial productivity gains [28], while even brief interruptions incur disproportionate recovery costs [40, 54].

To examine whether this disconnect stems from disruptions to developers' mental flow, we conduct two complementary studies. First, through an analysis of 50 real-world commits using Cursor and Claude Code, we show that 68.81% of model recommendations disrupt developers' ongoing mental flow, including 8.83% of suggestions that are technically correct but ill-timed. Second, through a controlled user study, we demonstrate that mitigating such flow disruptions leads to a 25.11% improvement in task efficiency.

We argue that such disruption of mental flow by existing code editing tools arises from limitations in how current code LLMs are trained. Most training data consists of commit snapshots, which represent only the final state of code changes, with the edits' temporal information lost during the commit process. As a result, LLMs learn from outcomes rather than the editing process itself, missing the temporal order that reflects developers' natural mental flow. This causes a fundamental

[1]ChatGPT-o1-mini achieves 92.4% accuracy on HumanEval [45]

mismatch: while models optimize for end results, developers work through incremental, context-sensitive steps that follow logical dependencies and cognitive continuity.

Given this fundamental mismatch, our core objective is to improve AI code editing assistants by aligning suggestions with developers' ongoing mental flow. However, realizing this objective is not straightforward, as it requires modeling aspects of the editing process that are largely absent from existing systems. In particular, it depends on the ability to infer cognitive order relations between edits, and its effectiveness must be validated through process-level evaluation rather than outcome-based metrics. Specifically, we identify three interconnected challenges that impede the development of flow-aware systems.

- **Challenge 1. The lack of flow-grounded edit-order data.** Avoiding flow violations fundamentally requires understanding *which edits should follow which others* under developers' natural reasoning processes. However, such data is difficult to obtain: manual annotation is costly and unscalable, development logs capture only a single realized trajectory, and LLM-based annotation often fails to generalize due to brittle prompts.
- **Challenge 2. The lack of process-level flow-aware evaluation.** Existing evaluation protocols for code editing assistants are predominantly outcome-oriented. For instance, systems are commonly evaluated using edit location precision and recall, content-level similarity metrics such as BLEU, or task completion measured by test passing. These metrics fail to capture the temporal appropriateness and cognitive continuity of edit suggestions, making it difficult to assess or improve flow-awareness across systems.
- **Challenge 3. The difficulty of unifying heterogeneous systems.** Existing code recommendation systems are highly heterogeneous, ranging from academic prototypes to closed-source IDE assistants and CLI-based tools. This heterogeneity, together with diverse interfaces, inference behaviours, and latency constraints, makes it difficult to consistently evaluate flow-related behaviours and to systematically reduce flow violations across different systems.

To address the aforementioned challenges, we propose three tightly integrated components. (1) A **prompt auto-tuning strategy** that leverages a small human-annotated dataset of edit orderings to learn an optimized prompt, capable of reconstructing the correct edit order between any two edits. (2) A **digital twin framework** that simulates developers' editing trajectories and their interactions with code recommendation systems, guided by the recovered edit order, enabling faithful evaluation of whether system-generated recommendations align with developers' ongoing mental flow. (3) Building on the above components, we propose our flow-alignment optimization solution, **EditFlow**, a unified wrapper for existing recommendation systems, which leverages the learned prompt to assess flow coherence, filters out flow-breaking recommendations. These components are interdependent: the prompt auto-tuning strategy (1) provides the core capability for both the digital twin (2) and EditFlow (3), while the digital twin (2) enables offline validation of the optimization (3).

We comprehensively evaluate EditFlow through four research questions. On the annotated dataset (**RQ1**), the auto-tuned prompt achieves 87.26% accuracy, substantially outperforming zero/few-shot and hand-crafted baselines by 63.81%. On a real-world industrial dataset (**RQ2**), it demonstrates strong robustness with only 30 flow violations, representing a reduction of over 75% compared to the best-performing baseline, and exhibiting close alignment with practical editing behavior. For edit recommendation (**RQ3**), the flow-aware optimization boosts precision across state-of-the-art systems (*Cursor*, *Claude Code*, and *CoEdPilot*), yielding an average precision improvement of 66.99%. Finally, a user study (**RQ4**) with 32 participants over 3 editing tasks shows 25.11% faster task completion and higher perceived recommendation quality.

We summarize our main contributions are as follows:

- To the best of our knowledge, we are the first to apply the concept of **mental flow in code editing**, and propose **EditFlow**, a post-processing solution that enables **flow-aligned edit recommendation**.
- To operationalize mental flow, we propose a **prompt auto-tuning strategy** that learns an optimized prompt to infer edit orders, enabling accurate recovery of editing sequences as a structured representation of developers' mental flow.
- To evaluate flow alignment at the process level, we design a **process-based digital twin evaluation framework** that simulates realistic editing trajectories guided by the recovered mental flow graph, enabling flow-aware assessment of heterogeneous recommendation systems.
- We demonstrate that EditFlow substantially improves **flow-aligned recommendation precision** and **developer productivity** through extensive experiments and a controlled user study.
- We implement EditFlow as a **VS Code extension** that provides visualized flow simulation and flow-aware recommendation filtering, with supplementary materials (e.g., demonstration videos, datasets and the learned prompt) available at our anonymous website [3].

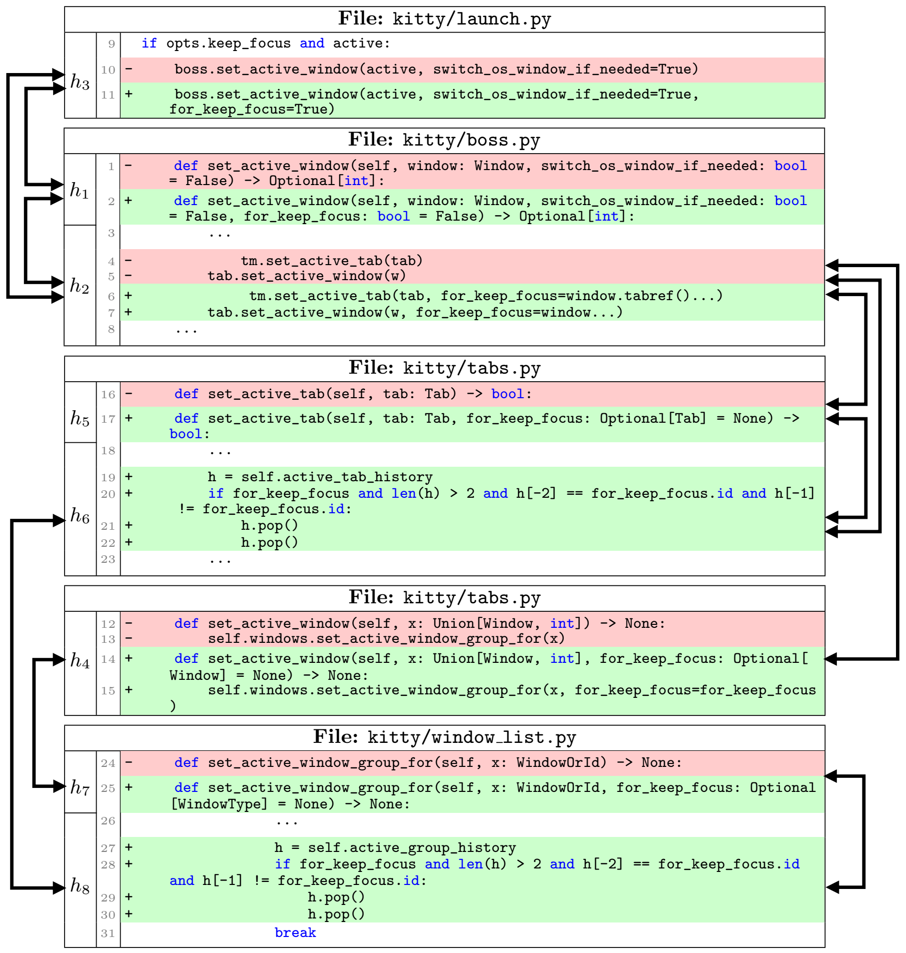
Building on these contributions, the remainder of the paper proceeds as follows. We begin with a motivating commit example that illustrates how existing code editing tools violate developers' mental flow during the editing process (Section 2). We then formalize mental flow as pairwise edit order relations and define the problem of flow-aligned edit recommendation (Section 3). To support process-level assessment, we introduce a digital-twin-based evaluation framework with corresponding metrics (Section 4). Using this framework, we conduct an empirical validation on real-world commits to quantify the prevalence and types of flow-violating recommendations in existing systems (Section 5). Finally, we present the design and implementation of EditFlow, including prompt auto-tuning for edit order recovery and flow-aware recommendation filtering, and evaluate its effectiveness through extensive experiments and a controlled user study (Section 6 & Section 7).

2 Motivating Example

Figure 1 shows a real-world commit from project `kovidgoyal/kitty`[2], which contains 8 edit hunks (denoted by h_i) across 4 files. Hunks such as $h_1\&h_2$, $h_5\&h_6$, and $h_7\&h_8$ are grouped as paired edits, representing the function signature and corresponding implementation body. An edge from h_i to h_j represents a partial order that users may proceed to h_j after completing h_i. All edges in the graph are bidirectional. This commit demonstrates a typical cascading change: adding a new parameter `for_keep_focus` to a function call at h_3, which creates a ripple effect throughout the call chain. The change first propagates to the target function's signature (h_1) and implementation (h_2), then branches into two parallel paths updating dependent functions: one path handles `set_active_window()` modifications ($h_2 \rightarrow h_4 \rightarrow h_7, h_8$), while the other addresses `set_active_tab()` changes ($h_2 \rightarrow h_5, h_6$).

For a set of edit hunks $\mathbb{H} = \{h_1, h_2, ..., h_8\}$ that together achieve an editing goal (adding the `for_keep_focus` parameter), the theoretical number of possible recommendation sequences is $|\mathbb{H}|! = 40,320$. However, the cognitively feasible sequences represent only a tiny fraction of this vast space. For example, after completing h_3, an AI assistant faces 7 possible immediate recommendations for the next edit, yet only h_1 and h_2 represent cognitively coherent choices that preserve the developer's mental flow continuity. Beyond treating all remaining edits as equally viable, current systems compound this challenge by introducing false positives that further dilute the precision of flow-coherent suggestions. Users must bear additional cognitive overhead to parse

[2]https://github.com/kovidgoyal/kitty/commit/c4c62c15



File: `kitty/launch.py`

```
 if opts.keep_focus and active:
-    boss.set_active_window(active, switch_os_window_if_needed=True)
+    boss.set_active_window(active, switch_os_window_if_needed=True,
    for_keep_focus=True)
```

File: `kitty/boss.py`

```
-    def set_active_window(self, window: Window, switch_os_window_if_needed: bool
    = False) -> Optional[int]:
+    def set_active_window(self, window: Window, switch_os_window_if_needed: bool
    = False, for_keep_focus: bool = False) -> Optional[int]:
         ...
-            tm.set_active_tab(tab)
-        tab.set_active_window(w)
+             tm.set_active_tab(tab, for_keep_focus=window.tabref()...)
+        tab.set_active_window(w, for_keep_focus=window...)
     ...
```

File: `kitty/tabs.py`

```
-    def set_active_tab(self, tab: Tab) -> bool:
+    def set_active_tab(self, tab: Tab, for_keep_focus: Optional[Tab] = None) ->
    bool:
         ...
+        h = self.active_tab_history
+        if for_keep_focus and len(h) > 2 and h[-2] == for_keep_focus.id and h[-1]
     != for_keep_focus.id:
+            h.pop()
+            h.pop()
         ...
```

File: `kitty/tabs.py`

```
-    def set_active_window(self, x: Union[Window, int]) -> None:
-        self.windows.set_active_window_group_for(x)
+    def set_active_window(self, x: Union[Window, int], for_keep_focus: Optional[
    Window] = None) -> None:
+        self.windows.set_active_window_group_for(x, for_keep_focus=for_keep_focus
    )
```

File: `kitty/window_list.py`

```
-    def set_active_window_group_for(self, x: WindowOrId) -> None:
+    def set_active_window_group_for(self, x: WindowOrId, for_keep_focus: Optional
    [WindowType] = None) -> None:
                 ...
+                h = self.active_group_history
+                if for_keep_focus and len(h) > 2 and h[-2] == for_keep_focus.id
    and h[-1] != for_keep_focus.id:
+                    h.pop()
+                    h.pop()
                 break
```

Fig. 1. Motivating example: Edit partial order graph

and validate irrelevant recommendations, ultimately causing the assistance to reduce rather than enhance productivity.

To validate this analysis, we replicated the editing process in leading AI coding tools. After completing h_3, Claude Code immediately jumped to recommending h_5, skipping the natural intermediate steps h_1 and h_2 entirely. When we then completed the cognitively natural flow $h_3 \rightarrow h_1 \rightarrow h_2$ and queried Cursor for subsequent recommendations, it correctly suggested h_5 but failed in other aspects of sequencing. Cursor recommended reverting the user-completed edit h_2, creating cognitive dissonance by contradicting a previously applied change.

Taken together, these failures highlight the urgent need for *flow-aware optimization* (Problem 3.3) to filter flow-violating suggestions. Achieving this requires (1) inferring pairwise cognitive order relations (Problem 3.1), and (2) a process-level evaluation framework to assess flow alignment throughout the editing process (Problem 3.2). We address these needs through a unified pipeline. We first develop a prompt auto-tuning strategy for edit order recovery, then leverage the recovered flow graph to drive a digital twin framework for process-level evaluation, and finally build EditFlow as a post-processing wrapper that filters and re-ranks recommendations across heterogeneous coding assistants.

3 Problem Formulation

As established in earlier sections, existing coding assistants often disrupt developers' mental flow, motivating us to formulate developers' mental flow as perceived order relations among edit hunks. This formulation enables reasoning about which edits constitute cognitively coherent subsequent steps under a given editing context, forming the basis for flow-aware recommendation.

3.1 Preliminaries

Definition 1 (Edit Hunk)**.** An *edit hunk h* is a tuple $(f, \ell_{\text{start}}, \ell_{\text{end}}, c_{\text{pre}}, c_{\text{post}})$ where:

- f is the file path;
- $\ell_{\text{start}}, \ell_{\text{end}}$ are line numbers defining the edit location;
- $c_{\text{pre}}, c_{\text{post}} \in \Sigma^*$ are the code content before and after the edit, where Σ^* denotes the set of all strings (including the empty string ε).

Let $\mathbb{H} = \{h_1, h_2, \ldots, h_n\}$ denote all edit hunks in a commit C.

Definition 2 (Pairwise Edit Order)**.** We use the term *partial order* to describe developers' perceived edit precedence from a cognitive perspective. Unlike the mathematical definition, our notion of partial order does *not* require transitivity: mental flow captures *local* continuity between adjacent steps, and skipping intermediate edits may incur a cognitive gap that disrupts rather than preserves flow.

For any pair of edit hunks $(h_i, h_j) \in \mathbb{H} \times \mathbb{H}$, we define their ground-truth partial order relation as $\lambda(h_i, h_j) \in \mathcal{L}$, where: $\mathcal{L} = \{\prec, \succ, \sim, \perp\}$.

Interpretation:

- $\lambda(h_i, h_j) =\prec$: After completing h_i, a developer can *naturally infer* the need for h_j based on cognitive continuity. *Example:* If h_i removes a code block (`cut`) from one location, and h_j subsequently inserts the same block elsewhere (`paste`), then $h_i \prec h_j$.
- $\lambda(h_i, h_j) =\succ$: Conversely, completing h_j naturally suggests h_i.
- $\lambda(h_i, h_j) =\sim$: Either direction preserves cognitive flow (symmetric). *Example:* If h_i deletes a parameter in a function's definition or signature, and h_j removes the corresponding argument at all call sites, then either order can be cognitively coherent, i.e., $h_i \sim h_j$.
- $\lambda(h_i, h_j) =\perp$: No apparent cognitive connection.

For conciseness, we limit the number of partial-order edit pair examples in the paper and provide additional illustrative cases on our anonymous website [3].

Remark (Cognitive vs. Technical Order). The relation λ characterizes *mental flow* rather than strict syntactic or semantic dependencies. Specifically, $\lambda(h_i, h_j) =\prec$ does not imply that h_i must precede h_j to maintain program correctness or prevent syntax errors. Instead, it indicates that completing h_i naturally renders h_j a cognitively coherent next step within the developer's reasoning process. Likewise, $\lambda(h_i, h_j) =\sim$ denotes a bidirectional cognitive relation, in which either edit may conceptually give rise to the other. For example, if h_i introduces a new function definition and h_j adds a corresponding function call, both orderings are cognitively plausible: a developer might first define the function to be invoked later, or alternatively, begin by writing the call and subsequently implement the function to fulfill the intent. We do not dismiss the latter direction merely because performing h_j first may lead to transient reference or import errors, as such temporary inconsistencies do not disrupt the continuity of the developer's mental flow.

Remark (Approximating Ground-Truth Order). The ground-truth relation $\lambda(h_i, h_j)$ reflects an intrinsic cognitive property. Although collecting edit logs may appear to be an intuitive way to obtain edit order, such logs can only reveal the immediate sequence between consecutive edits and, at best, capture one possible ordering among many. They cannot recover the full set of pairwise relations within the edit set. Therefore, in practice, we approximate λ via human annotation or inference by large language models, as described in Section 6.1. Regardless of the approach, all annotations aim to capture the underlying cognitive order between edits.

Definition 3 (Mental Flow Graph). Given edit hunks $\mathbb{H}$ and pairwise order labels λ, the *mental flow graph* is a directed graph $G = (\mathbb{H}, \mathcal{E})$, where the edge set $\mathcal{E}$ is defined as:

$$\mathcal{E} = \{(h_i, h_j) \mid \lambda(h_i, h_j) = \prec\} \cup \{(h_j, h_i) \mid \lambda(h_i, h_j) = \succ\} \cup \{(h_i, h_j), (h_j, h_i) \mid \lambda(h_i, h_j) = \sim\}.$$

Definition 4 (One-Hop Successor). Given $G = (\mathbb{H}, \mathcal{E})$ and a set of completed edits $H_{\text{prior}} \subseteq \mathbb{H}$, an edit $h' \in \mathbb{H} \setminus H_{\text{prior}}$ is a *one-hop successor* of H_{prior} if:

$$\exists h \in H_{\text{prior}} : (h, h') \in \mathcal{E}.$$

We define the set of one-hop successors of H_{prior} as:

$$\text{Succ}(H_{\text{prior}}, G) = \{\, h' \in \mathbb{H} \setminus H_{\text{prior}} \mid \exists h \in H_{\text{prior}} : (h, h') \in \mathcal{E} \,\}.$$

3.2 Problem Statements

Problem 3.1 (Mental Flow Order Recovery). Given a commit with edit hunks $\mathbb{H} = \{h_1, \ldots, h_n\}$, the problem is to design a recovery method M that infers the partial order relation $\lambda : \mathbb{H} \times \mathbb{H} \longrightarrow \mathcal{L}$ (as in Definition 2) between any two edits, which captures developers' perceived precedence between edits. By applying the recovered order relation to all edit pairs, the mental flow graph $G = (\mathbb{H}, \mathcal{E})$ (as in Definition 3) can be induced accordingly, and serve as the ground-truth input for Problem 3.2. The recovery method M is also reused in Problem 3.3 to guide flow-aware optimization.

Problem 3.2 (Process-driven Flow Alignment Evaluation). Given a code edit recommendation system $\mathcal{S}$ and a commit $(\mathbb{H}, G)$ with mental flow graph $G = (\mathbb{H}, \mathcal{E})$ recovered from solving Problem 3.1, the problem is to evaluate the extent to which $\mathcal{S}$ produces flow-aligned edit recommendations during an editing process. Specifically, the editing process is viewed as a sequence of intermediate states, each represented by the set of already-applied edits $H^{(t)}_{\text{prior}}$, with $H^{(0)}_{\text{prior}} = \emptyset$ and $H^{(T)}_{\text{prior}} = \mathbb{H}$. At each state t, the system $\mathcal{S}$ generates a set of candidate edits $H^{(t)}_{\text{pred}}$ conditioned on $H^{(t)}_{\text{prior}}$. The goal is to quantify, across intermediate states, whether and to what extent edits in $H^{(t)}_{\text{pred}}$ are flow-continuous with some edit in $H^{(t)}_{\text{prior}}$ according to the mental flow graph G.

Problem 3.3 (Flow-aware Optimization). Given an editing context characterized by a set of completed edits H_{prior}, and a set of currently suggested edit H_{pred} predicted by code edit recommendation system $\mathcal{S}$, we seek a post-processing mechanism $\phi : 2^H \longrightarrow 2^H$. The objective of ϕ is to reduce flow-violating recommendations and prioritize flow-coherent subsequent edits, such that:

$$\text{Precision}(\phi(H_{\text{pred}})) > \text{Precision}(H_{\text{pred}}).$$

We address this problem in Section 6.2, by designing a flow-aware post-processing strategy, utilizing edit order recovery method M from Problem 3.1.

Problem Relationships. The three problems directly address the three challenges identified in Section 1: Challenge 1 (the lack of flow-grounded edit-order data) motivates Problem 3.1; Challenge 2 (the lack of process-level flow-aware evaluation) motivates Problem 3.2; and Challenge 3 (the difficulty of unifying heterogeneous systems) motivates both Problem 3.2 and Problem 3.3, as both handle diverse systems.

Beyond this correspondence, the three problems form a layered structure, as shown in Figure 2: Problem 3.3 is the ultimate application goal, and the other two problems provide the necessary infrastructure. Specifically, Problem 3.1 provides the foundation that supports both Problem 3.2 and Problem 3.3: the recovered mental flow graph serves as the ground-truth edit order in Problem 3.2, guiding the simulation of the editing process. Meanwhile, the same recovery method is leveraged as a core component of the flow-aware filtering mechanism in Problem 3.3. Problem 3.2 serves as a digital twin for validating the optimization in Problem 3.3, enabling offline assessment without real user studies.

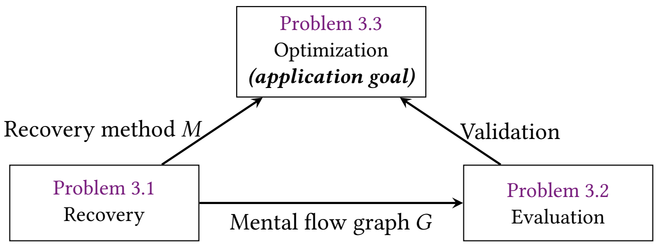


Fig. 2. Relationships among the three problems.

4 Flow-Aware Evaluation Framework

To enable process-driven, flow-aware evaluation, as defined in Problem 3.2, we present a digital twin framework that simulates developers' editing processes with corresponding metrics.

4.1 Digital Twin

To evaluate the flow-awareness of existing coding assistants at the progress level, we design a digital twin that simulates real-world editing processes guided by recovered developer flow. For any git commit containing n edit hunks $\mathbb{H}$, we construct an edit partial order graph G, as defined in Definition 3. The simulation first checkout the commit to its pre-change version, where no edit has been applied yet ($H_{\text{prior}} = \emptyset$), then randomly selects an edit hunk with minimum in-degree as the initial edit, applying it to the project. At each iteration, the digital twin interacts with the system under test (SUT) through a standard interface, supplying updated context as the simulation progresses, including: (1) the list of previously simulated edits ($H_{\text{prior}} \subset \mathbb{H}$), (2) the access to the codebase at the current simulation progress, where previously simulated edits have all been applied to the project, and (3) commit messages as edit descriptions. SUTs are expected to return their recommended edits through this unified interface. This design ensures that the simulation system remains general and system-agnostic, with specific integration details for each evaluated SUT described in Section 7.3.3. We evaluate the predicted edits through metrics defined in Section 4.2. If any of the recommended edit hunks are identified as a Keep suggestion (i.e., it correctly aligns with a valid next edit in the flow graph G, see Definition 5 for the formal definition), the simulation system randomly selects a Keep suggestion as the subsequent edit, applies it to the project, and updates the prior edits ($H_{\text{prior}} = H_{\text{prior}} \cup \{h\}$); otherwise, it selects from one-hop successor hunks ($h \in \text{Succ}(H_{\text{prior}}, G)$, as defined in Definition 4). This design assumes that, in real editing interactions, users would not apply incorrect edits to the project. The simulation continues to request the next recommendation from SUT, until all ground-truth edit hunks have been simulated. To better visualize this process, we implement this digital twin as a VS Code extension that capable of replaying this simulated editing process, with demonstration video available at our homepage [3].

This framework is used in two places throughout the paper. First, in the empirical study (Section 5), it enables a small-scale quantification of whether existing coding assistants violate developers' mental flow. Second, in the main experiments (Section 7.3), it is used to evaluate assistant behaviour with and without EditFlow, allowing us to assess the effectiveness of flow-aware optimization under the same evaluation protocol.

4.2 Metrics

We design three categories of metrics to comprehensively evaluate coding assistants, capturing perspectives including flow alignment, recommendation correctness, and resource usage.

4.2.1 Flow-aware Metrics. Based on the flow graph, we classify predicted edits according to their alignment with the developer's current mental flow state.

Definition 5 (Flow Categories)**.** Let $G = (\mathbb{H}, \mathcal{E})$ be a mental flow graph, $H_{\text{prior}} \subseteq \mathbb{H}$ the sequence of completed edits, and $h' \in H_{\text{pred}}$ denotes a predicted edit suggested by subsequent edit recommendation systems. We define four mutually exclusive categories:

$$\textsc{Keep}(h', H_{\text{prior}}, G) \equiv h' \in \mathbb{H} \setminus H_{\text{prior}} \wedge h' \in \text{Succ}(H_{\text{prior}}, G) \tag{1}$$

$$\textsc{Jump}(h', H_{\text{prior}}, G) \equiv h' \in \mathbb{H} \setminus H_{\text{prior}} \wedge h' \notin \text{Succ}(H_{\text{prior}}, G) \tag{2}$$

$$\textsc{Revert}(h', H_{\text{prior}}, G) \equiv h' \in H_{\text{prior}} \tag{3}$$

$$\textsc{Break}(h', H_{\text{prior}}, G) \equiv h' \notin \mathbb{H} \tag{4}$$

Figure 3 illustrates these categories geometrically: A Keep edit maintains cognitive continuity by following an edge in G; a Jump edit targets a valid hunk but skips intermediate logical steps; a Revert edit suggests discarding an already-applied change; and a Break edit hallucinates content not present in the ground-truth commit.

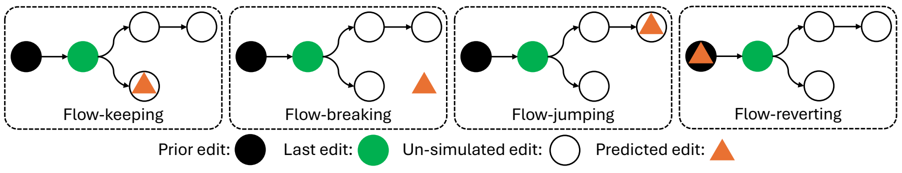


Fig. 3. Flow categories of predicted edit

Lemma 1 (Partition Property). *For any predicted edit $h' \in$ Edits and fixed H_{prior}, G, exactly one of the four predicates in Definition 5 holds.*

Proof. By case analysis on $(h' \in H_{\text{pred}})$:

- If $h' \notin \mathbb{H}$: $\textsc{Break}(h')$ holds by Eq. (4), and others fail since $h' \notin \mathbb{H} \implies h' \notin H_{\text{prior}}$ and $h' \notin \mathbb{H} \setminus H_{\text{prior}}$.
- If $h' \in H_{\text{prior}}$: $\textsc{Revert}(h')$ holds by Eq. (3). Since $H_{\text{prior}} \subseteq \mathbb{H}$, we have $h' \in \mathbb{H}$ but $h' \notin \mathbb{H} \setminus H_{\text{prior}}$, so Keep and Jump fail. Break fails since $h' \in \mathbb{H}$.
- If $h' \in \mathbb{H} \setminus H_{\text{prior}}$: By Definition 4, either $h' \in \text{Succ}(H_{\text{prior}}, G)$ (then Keep holds and Jump fails) or $h' \notin \text{Succ}(H_{\text{prior}}, G)$ (then Jump holds and Keep fails). Revert fails since $h' \notin H_{\text{prior}}$, and Break fails since $h' \in \mathbb{H}$.

The cases are exhaustive (partition Edits by membership in $\mathbb{H}$ and H_{prior}) and mutually exclusive (each case yields exactly one true predicate). □

4.2.2 Flow-independent Metrics. To avoid potential bias introduced by the construction of flow graphs in flow-aware metrics, we additionally define a set of flow-independent, ground-truth-based metrics. Given a commit with a set of ground-truth edit hunks $\mathbb{H}$, a set of recommended next code edits from a single request $\mathbb{H}'$, and a set of all recommended edits during the entire simulation process $\mathbb{H}''$, the flow-independent metrics evaluate recommendation quality based on traditional correctness criteria:

$$\text{Precision} = \frac{|\mathbb{H}' \cap \mathbb{H}|}{|\mathbb{H}'|}, \quad \text{Recall} = \frac{|\mathbb{H}'' \cap \mathbb{H}|}{|\mathbb{H}|}, \quad \text{F}_{0.5} = \frac{(1 + 0.5^2) \cdot \text{Precision} \cdot \text{Recall}}{0.5^2 \cdot \text{Precision} + \text{Recall}} \tag{5}$$

4.2.3 Resource Usage Metrics. In addition to recommendation quality, we evaluate the resource overhead introduced by code editing assistants during real-time interaction. We consider the following resource usage metrics:

- **Token Usage.** The total number of tokens consumed per recommendation, including both input tokens (prompt and context) and output tokens (generated edits or responses). This metric serves as a hardware-agnostic proxy for computational cost.
- **Latency.** The end-to-end wall-clock time required to generate a recommendation, measured from request issuance to response delivery. Latency reflects system responsiveness and directly affects interaction smoothness and mental flow.
- **Monetary Cost.** The estimated cost per recommendation is computed based on total token usage and the pricing scheme of the underlying model or service. For locally deployed open-source models, we report equivalent token-based cost to enable fair comparison.

5 Empirical Validation

To preliminarily validate our motivation, we perform an exploratory evaluation using the evaluation framework proposed in Section 4. We randomly selected 50 annotated Python commits from our training dataset with manually constructed edit partial order graph (details may refer to Section 7.1.2), with Cursor CLI [16] and Claude Code [6] as our SUTs. The results are shown in Table 1. We observe that the majority of predicted edits fall into the Break category, accounting for 55.48% in Cursor and 51.23% in Claude Code. In contrast, Keep edits, which truly align with the ongoing mental flow, represent only 28.23% and 34.16% respectively. The remaining predictions are largely distributed across Jump and Revert cases, both of which also disrupt the natural progression of edits.

Overall, these findings confirm our hypothesis: existing edit recommendation systems struggle to maintain mental flow alignment, with the majority of recommendations either irrelevant or disruptive. This misalignment explains why strong benchmark performance does not translate into productivity gains in real development workflows.

Table 1. Empirical study of flow-violation in existing code edit recommendation system

Baseline	**Flow categories (%)**			
	Keep	**Jump**	**Revert**	**Break**
Cursor	28.23	9.84	6.45	55.48
Claude Code	34.16	7.82	6.79	51.23

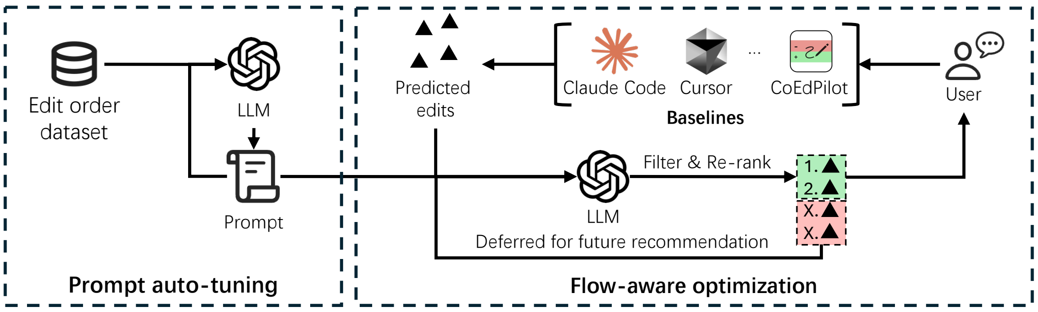


Fig. 4. Overview of EditFlow

6 Methodology

In this section, we present EditFlow, our flow-alignment optimization solution for AI code editing assistants, as illustrated in Figure 4. We first auto-tune an edit order recovery prompt from a human-labelled edit order dataset, and integrate such a prompt into the subsequent edit recommendation process, where we filter and re-rank edit suggestions produced by existing coding assistants based on their continuity with the mental flow reflected in prior edits.

6.1 Edit Order Recovery

To address Problem 3.1, we propose a prompt-based approach that enables large language models to infer the pairwise cognitive order between code edits. Specifically, we aim to train and optimize task-specific prompts that guide the LLM to infer the partial order relation for any pair of edit hunks.

Since λ is a latent cognitive property not directly observable from development histories, no existing training data is available. We therefore manually construct an annotated dataset that approximates λ by labelling pairwise order relations between edits within each commit. Two authors conducted the annotation independently, with access to the full repository and additional static analysis data, including dependency relations and the structural context of each edit (e.g., the enclosing class, function, or method call). The process followed two stages: (i) independent annotation of potential edit orders, and (ii) consensus resolution of disagreements through discussion. On average, each commit required 20 minutes to annotate, resulting in a total annotation effort of 77 person-hours.

The resulting annotations define a set of pairwise relations, which are formalized as edges in an edit partial order graph. Each pair of hunks has a label from $\mathcal{L} = \{\prec, \succ, \sim, \perp\}$. Each hunk is represented using XML tags that encapsulate contextual information, with the edit itself expressed in Git diff format. The contextual information includes the file path, the structural path of the edit, and any code elements that establish dependencies between the two hunks, as illustrated in Table 2.

Given this annotated dataset, we formalize the learning objective as identifying the optimal prompt p^* that maximizes the predictive accuracy of the LLM on the pairwise annotation set:

$$p^* = \arg\max_{p \in P} \text{Accuracy}(p; D_{tr}),$$

where P is the set of candidate prompts and D_{tr} is the manually annotated training set. As shown in Algorithm 1, we split the dataset into k batches and feed each complete batch to the LLM with only the response JSON schema, requiring the LLM to summarize an initial prompt for the edit order recovery task based on the input-output pairs (x_i, y_i). The algorithm iteratively refines

Table 2. Example of edit hunk representation

```
<file_path>kitty/launch.py</file_path>
<structural_path>
def launch(boss: Boss, ...)->Optional[Window]:
    boss.set_active_window(active, switch_os_window_if_needed=True)
</structural_path>
<code>
477 477        if opts.keep_focus and active:
478     -          boss.set_active_window(active, switch_os_window_if_needed=True)
    478 +          boss.set_active_window(active, switch_os_window_if_needed=True,
         for_keep_focus=True)
479 479            if opts.logo:
</code>
```

Algorithm 1 Prompt Auto-Tuning for Edit Order Recovery

```
1:  Input: Black-box large language model $LLM$; Training data $D_{tr} = \{(x_i, y_i)\}$, where $x_i = (h_i^0, h_i^1)$
    is edit hunk pair, $y_i = \lambda(h_i^0, h_i^1) \in \{\prec, \succ, \sim, \perp\}$
2:  output: Optimal prompt $p^*$
3:  Randomly split $D_{tr}$ into batches: $B = \{b_1, \ldots, b_k\}$
4:  Obtain initial prompt candidates: $P = \{p_k | p_k := LLM(b_k)\}$
5:  Select optimal prompt: $p^* := \arg\max_{p \in P} \text{Accuracy}(p; D_{tr})$
6:  for Epoch $e$ from 1 to $E$ do
7:      $S^+ = \{(x_i, y_i) | LLM(p^*, x_i) = y_i, (x_i, y_i) \in D_{tr}\}$
8:      $S^- = \{(x_i, y_i) | LLM(p^*, x_i) \neq y_i, (x_i, y_i) \in D_{tr}\}$
9:      Re-split $D_{tr}$ into batches where each batch contains samples from both $S^+$ and $S^-$: $B =$
        $\{b_k | b_k \cap S^+ \neq \emptyset, b_k \cap S^- \neq \emptyset\}$
10:     for each batch $b_k \in B$ do
11:         Generate feedback: $F_k = \{f_i | f_i := LLM(p^*, b_k)\}$
12:         Integrate feedback to optimize prompt: $p_k := LLM(p^*, F_k)$
13:     end for
14:     $p^* := \arg\max_{p \in \{p_1, \cdots, p_k\}} \text{Accuracy}(p; D_{tr})$
15: end for
16: Return: $p^*$
```

prompts through feedback-driven optimization. In each epoch, we partition training data into correct predictions (S^+) and failed cases (S^-) using the current optimal prompt p^*. We then re-batch the dataset, ensuring each batch contains both successful and failed examples, providing contrastive learning signals. For each batch b_k, we generate feedbacks F_k by analyzing the prompt's performance, capturing why certain predictions succeeded while others failed. The LLM integrates this feedback to optimize prompt p_k, learning to distinguish between different edit order relationships. Finally, we evaluate all candidate prompts on the entire training set and select the one with the highest accuracy as the new optimal prompt p^*. For the detailed learned prompt, please refer to our anonymous website [3].

6.2 Flow-aware Optimization

To optimize the flow alignment of edit recommendations (Problem 3.3), we implement EditFlow as a wrapper that encapsulates existing sequential edit recommendation systems, capable of improving the precision of edit suggestions by filtering out non-flow-keeping suggestions and re-ranking them based on their relevance to maintain mental flow continuity. Given a code project **P**, the sequence of prior edits $H_{\text{prior}} = \langle h_1, \ldots, h_n \rangle$, and the edit description d, EditFlow queries the edit recommendation system to obtain a set of recommended edits, denoted as $H_{\text{pred}} = \{h'_1, \ldots, h'_k\}$. EditFlow then infers the pairwise order label between each predicted edit h'_i and prior edit h_n via $\text{LLM}(h_n, h'_i, p)$, where p represents the auto-tuned edit order recovery prompt (as in Section 6.1). The LLM outputs the inferred edit order label along with the log probability of each predicted token. A prediction of $\prec$ or $\sim$ indicates a potential edit order from the user's last edit h_n to the predicted edit h'_i ($h_n \to h'_i$ or $h_n \leftrightarrow h'_i$), whereas $\succ$ denotes an edit order $h'_i \to h_n$, which reverses the user's current mental flow, and $\perp$ suggests minimal relation between h_n and h'_i. Only edits predicted as either $\prec$ or $\sim$ are retained for subsequent processing. EditFlow re-ranks the remaining suggested edits according to the average log probability of the edit order label tokens predicted by $\text{LLM}(h_n, h'_i, p)$. A higher score indicates greater confidence in the predicted order label. For edits that are predicted as $\succ$ or $\perp$, EditFlow does not discard them permanently. Instead, these edits are deferred and may be reconsidered in future iterations when the user's editing context evolves. As the sequence of prior edits grows, previously deferred edits can become flow-aligned and thus eligible for recommendation at a more appropriate time. This design ensures that EditFlow prioritizes maintaining immediate mental flow continuity while preserving potentially useful edits for later recommendation.

We also implement EditFlow as a VS Code extension, which wraps Cursor CLI, Claude Code and CoEdPilot as backends, filters and re-ranks their suggested edits before presenting them to users. All edits are displayed in descending order of their average log probability, with each entry clickable to navigate to the corresponding file and view the associated edit diff. For a demonstration of the interaction, please refer to our anonymous website [3].

7 Experiment

We evaluate our edit order recovery methods and various subsequent edit recommendation systems via the following research questions:

- **RQ1 (Effectiveness on annotated dataset, Section 7.1)**: Can the learned partial order prompt perform well on our annotated dataset?
- **RQ2 (Alignment with practical data, Section 7.2)**: How well does the partial order graph inferred by the learned prompt align with real-world editing data?
- **RQ3 (Performance of flow-aware optimization, Section 7.3)**: Can our learned partial order prompt improve the suggestion precision of existing subsequent edit recommendation systems?
- **RQ4 (User study, Section 7.4)**: Can our learned partial order prompt improve the real-world development productivity?

7.1 RQ1 (Effectiveness on Annotated Dataset)

7.1.1 Setup. Prompt auto-tuning was performed and tested with `Claude-Sonnet-4-20250514`, using a maximum output length of 4096 tokens and a temperature of 0.7. The auto-tuning underwent 5 epochs and a batch size of 32.

7.1.2 Benchmark. The annotated dataset consists of 100 commits from the 45 most-starred open-source GitHub Python repositories, comprising a total of 772 edit hunks and 1,747 directed edges. From these, we constructed 2,030 training samples and 871 test samples. To avoid intra-commit data leakage, we split the dataset at the commit level in a 7:3 ratio, ensuring that all samples from the same commit are assigned to the same split.

7.1.3 Baseline. We compare our auto-tuned prompt with 4 other baselines:

- **Zero-shot learning**: The LLM performs the task with only a single-sentence task description and the expected response schema, without any example or task-specific prompt tuning;

Table 3. The performance of the auto-tuned prompt on annotated dataset

Method	Accuracy (%)	Precision (%)	F1-score (%)
Zero-shot learning	50.63	82.22	61.67
Few-shot learning	47.42	77.13	57.96
Hand-crafted prompt	53.27	80.96	60.93
DSPy	53.39	62.44	57.37
Auto-tuned prompt	**87.26**	**88.01**	**87.54**

- **Few-shot learning**: The LLM is provided with 8 in-context examples, each consisting of a pair of edit hunks along with their edit order label, illustrating the edit order recovery task; the examples include four instances of each of the 2 possible labels.
- **Hand-crafted prompt**: A prompt manually designed based on the annotators' experience, summarizing key instructions for the edit order recovery task.
- **DSPy** [29]: DSPy is a declarative framework for automated prompt optimization. We design a multi-step DSPy module that sequentially extracts edit information, analyzes dependency relations, and infers the final edit order via structured chain-of-thought reasoning. MIPROv2 optimizer [46] is adopted to refine the prompt based on human-labelled training data.

7.1.4 Metric. We evaluated the quality of the generated prompts using accuracy, weighted precision, and F1 score (recall is equivalent to accuracy in this multi-class setting).

7.1.5 Result. Table 3 reports the performance of different prompting strategies on our annotated dataset. Overall, the results show that the auto-tuned prompt substantially outperforms all baselines across accuracy, precision, and F1-score. In particular, the auto-tuned prompt achieves an accuracy of 87.26%, representing a 63.81% relative improvement over the best-performing baseline (53.39%). This demonstrates the effectiveness of automated prompt optimization in capturing the underlying principles of edit ordering, surpassing other baseline strategies. Notably, despite the common adoption of DSPy as a general-purpose prompt optimization framework, its performance falls behind our auto-tuned approach, suggesting that generic example-based optimization is insufficient for capturing the fine-grained and rule-intensive nature of edit-order prediction.

7.2 RQ2 (Alignment with Practical Data)

7.2.1 Setup and Baseline. This experiment adopted the same setup and baseline as depicted in Section 7.1.1 and Section 7.1.3.

7.2.2 Benchmark. To evaluate the quality of the auto-tuned prompt in the real world, we collected an edit order dataset from our industry collaborator, an international information technology company with over 60K employees. This real-world dataset is collected inside our collaborator, generated by its employees[3] This industry collected dataset consists of 500 commits from Jun. 2025 to Aug. 2025, containing 3,059 edit hunks.

We input any two edit hunks (e.g., h_i, h_j) within the same commit into the LLM, under the instruction of the auto-tuned prompt, to infer their partial order. Thereby, we construct a predicted partial order graph $G = (\mathbb{H}, \mathcal{E})$, where $\mathbb{H}$ is the set of all edit hunks from the given commit and $\mathcal{E}$ is the set of inferred partial order edges.

[3]The data collection obtained employee consent and underwent anonymization. All data is stored in the collaborator's internal systems throughout the process, industry-collected and access to and analysis of the data during the research were conducted in a secure environment authorized by the collaborator. This dataset has not been disseminated outside the research team.

Table 4. Number of violations of the auto-tuned prompt and baselines on the real-world dataset

Method	**#Violation**
Zero-shot learning	195
Few-shot learning	203
Hand-crafted prompt	121
Auto-tuned prompt	**30**

7.2.3 Metric. We then evaluate the reliability of the predicted partial order graph, by measuring the number of forbidden partial order relationships inferred from the predicted partial order graph G, that are contradicted by the ground-truth edit order sequence S. The choice of a violation-based metric is motivated by the inherent ambiguity in evaluating partial order predictions against sequential ground truth. There may exist multiple natural and flow-keeping editing sequences for a commit with n edit hunks. However, practical data collection constraints limit us to observing a single editing sequence per commit, as it is infeasible to have developers repeatedly perform identical editing tasks to enumerate all valid orderings. This asymmetry between the prediction space (partial orders) and the observation space (single gold sequence) renders traditional accuracy-based metrics problematic. Specifically, for any predicted partial order relationship that does not appear in the observed sequence, we cannot definitively classify it as incorrect, since it may represent a valid alternative ordering that was simply not observed in our particular ground-truth instance. To address this challenge, we adopt a violation-based approach that focuses on demonstrably incorrect predictions. While this approach is also affected by the single-gold-sequence limitation and provides an underestimation, it offers substantially higher reliability compared to accuracy-based metrics. The key advantage is that every detected violation represents a definitive error, a constraint that is provably inconsistent with observed developer behaviour. The detailed algorithm to infer forbidden partial orders is available on our homepage [3].

7.2.4 Result. As shown in Table 4, the auto-tuned prompt significantly outperforms all baselines in terms of reducing violations. Specifically, while zero-shot and few-shot learning lead to 195 and 203 violations respectively, and the hand-crafted prompt reduces this number to 121, the auto-tuned prompt achieves only 30 violations. This represents a reduction of over 75% compared to the best-performing baseline, demonstrating the superior alignment of the auto-tuned prompt with real-world editing behaviors. These results confirm that prompt auto-tuning substantially enhances the reliability of partial order inference in practical development settings.

7.3 RQ3 (Performance of Flow-aware Optimization)

We evaluate the subsequent edit recommendation performance of different baselines with and without our proposed flow-aware optimization technique.

7.3.1 Experiment Design. We adopt the digital twin for this experiment, with details described in Section 4.1. For any git commit containing n edit hunks $\mathbb{H}$, we construct an edit partial order graph G using an auto-tuned prompt p. We assume the inferred partial order graph as the ground-truth graph. Notably, these graphs are not involved in EditFlow's filtering process. They are used to drive Digital Twin simulations by traversing the graph, ensuring that the simulated editing progress remains aligned with developers' mental flow, and to support the computation of flow-aware metrics.

7.3.2 Benchmark. We adopt 2 benchmarks for this research question.

- **Large scale benchmark.** The simulation benchmark consists of 500 commits from 80 most-starred open-source GitHub Python repositories, and contains 3,584 hunks in total. For commits of this benchmark, we adopt the auto-tuned prompt and `Claude-Sonnet-4-20250514` to derive the edit partial order graph. Commit selection criteria include: (1) containing 5-10 edit hunks across at least 2 source files; (2) involving real user authorship rather than automated systems; (3) excluding merge commits and filename changes; and (4) maintaining ASCII-only content with meaningful code modifications.
- **Human annotated benchmark.** To avoid the concern that improvements on flow-aware metrics are a self-fulfilling consequence of EditFlow-generated order signals, we include our human-annotated benchmark with manually labeled partial orders. This benchmark consists of 25 Python commits, and more details may refer to Section 7.1.2.

7.3.3 Baseline. We selected Claude Code (Version 1.0.113), Cursor CLI (Version 2025.09.18-7ae6800), and CoEdPilot as our baseline. For Cursor CLI and Claude Code, we relied on their default underlying models without manually specifying a particular model. Each system requires different integration approaches with our simulation framework:

- **Claude Code and Cursor CLI:** Both the Anthropic Claude Code SDK [1] and Cursor CLI [16] support headless mode to avoid GUI interaction. We supply the project path and structured requests with the same format, containing editing history and task specifications. The digital twin grants both Claude and Cursor agents comprehensive access to read, write files, and execute bash commands for project analysis.
- **CoEdPilot:** We employ CoEdPilot's discriminator model to select editing files, then incorporate prior edits and edit descriptions into the locator and generator model inputs following CoEdPilot's specific formatting requirements.

For baselines without flow-aware optimization, we refer to this configuration as **Original**, where recommendations returned from baselines are directly evaluated. For baselines with flow-aware optimization, we adopt our proposed EditFlow to filter and re-rank returned suggestions before evaluation, we refer to this configuration as **w/ EditFlow**.

7.3.4 Metric. We evaluate flow-aware optimization via three categories of metrics, including flow alignment, recommendation correctness, and resource usage. Details may refer to Section 4.2.

7.3.5 Result. For flow-aware metrics, as shown in Table 5, EditFlow brings an average boost of 87.42% to flow-keeping edits and an average drop of 22.42% to flow-breaking, on the large-scale benchmark. This improvement is not due to the LLM favouring its own derived editing order. As shown in Table 6, on the human-annotated edit ordering benchmark, EditFlow achieves an average 106.58% boost in flow-keeping edits and an average 19.15% reduction in flow-breaking edits, exhibiting a trend consistent with the large-scale benchmark results.

We further evaluate EditFlow using precision, recall, and F0.5 as flow-independent metrics to provide a more objective assessment. Across the two benchmarks, EditFlow achieves an average 66.99% improvement in precision, accompanied by a 7.09% decrease in recall and an overall 46.64% increase in F0.5, indicating a favourable trade-off that prioritizes recommendation correctness while maintaining stable coverage.

To assess the practical applicability of our approach, we further analyze its resource usage. On average across three SUTs and two benchmarks, EditFlow introduces an additional 1.71 seconds of latency, 6.58k tokens, and $0.03 in cost per query. Both Claude and Cursor incur substantial latency from their agent-based workflows, where file inspection and tool invocation already dominate the end-to-end execution time. In comparison, the additional overhead introduced by EditFlow accounts for only a small fraction of the overall latency. Separately, prior studies suggest that short

Table 5. Performance of flow-aware optimization on large-scale benchmark

Baseline	Config	Flow categories (%)				Prec. (%)	Rec. (%)	$F_{0.5}$ (%)	Resource usage per query		
		Keep	Jump	Revert	Break				Latency (s)	Tokens (K)	Cost ($)
Cursor	Original	24.00	9.02	6.83	60.15	33.02	40.01	34.22	65.09	174.47	0.0621
	w/ EditFlow	38.49	3.93	6.45	51.13	42.42	37.97	41.45	67.12	180.68	0.0869
Claude Code	Original	30.76	9.78	5.97	53.49	40.54	39.46	40.32	45.46	112.82	0.3609
	w/ EditFlow	46.89	3.56	4.03	45.52	50.45	36.15	46.75	46.70	117.15	0.3783
CoEdPilot	Original	13.30	1.48	2.31	82.91	14.78	28.08	16.33	6.49	43.59	0.0000
	w/ EditFlow	33.18	2.32	1.93	62.57	35.50	25.68	32.98	8.38	54.80	0.0445

Table 6. Performance of flow-aware optimization on human-labelled benchmark

Baseline	Config	Flow categories (%)				Prec. (%)	Rec. (%)	$F_{0.5}$ (%)	Resource usage per query		
		Keep	Jump	Revert	Break				Latency (s)	Tokens (K)	Cost ($)
Cursor	Original	29.17	14.88	3.87	52.08	44.05	45.95	44.41	51.29	195.81	0.0660
	w/ EditFlow	47.96	05.58	3.35	43.12	53.53	45.41	51.68	52.94	202.77	0.0926
Claude Code	Original	29.55	10.12	6.88	53.44	39.68	38.92	39.52	50.53	114.62	0.3661
	w/ EditFlow	46.88	2.08	4.17	46.88	48.96	36.76	45.91	52.19	119.63	0.3853
CoEdPilot	Original	8.16	1.84	3.95	86.05	10.00	15.68	10.78	6.83	45.29	0.0000
	w/ EditFlow	23.61	2.78	3.47	70.14	26.39	14.05	22.45	8.63	51.04	0.0223

delays below approximately two seconds are generally not perceived as disruptive and are unlikely to break users' mental flow [41]. Moreover, these costs can be further reduced through engineering optimizations, such as adopting recent inference acceleration techniques [10, 11, 55], leveraging more cost-efficient models as employed by Cursor.

In summary, the results demonstrate that flow-aware optimization leads to more flow-preserving recommendations, reducing disruptions while maintaining or improving overall effectiveness.

7.4 RQ4 (User Study)

To validate EditFlow performance in real-world usage, we design this user study with our implemented VS Code extension. Further details, including demographic details and instrumentation-based analysis of user interactions, are available on our homepage [3].

7.4.1 System Under Test. We evaluate the effectiveness of our EditFlow by applying it to two representative code editing systems, CoEdPilot and Claude Code. For each system, we compare two variants: (1) the original system without EditFlow, and (2) the optimized system with EditFlow integrated. This setup allows us to isolate the contribution of EditFlow by controlling for differences across systems and directly measuring its impact on editing recommendation quality in terms of flow-awareness.

7.4.2 Participant. We recruited 32 participants from 2 universities. All participants are required to complete a pre-study questionnaire to collect their background information, including educational level, programming proficiency, and prior experience with AI-assisted programming tools. The participants were aged between 20 and 30 and enrolled in computer science programs, ranging from undergraduate to PhD level. They demonstrated substantial proficiency in Python programming, engaging in coding activities on average 4.5 days per week. Additionally, 90% of participants reported prior experience with AI-assisted programming tools.

Participants using the original Claude Code constitute Control Group 1 (CG1), while those using Claude Code integrated with EditFlow form Experiment Group 1 (EG1). Similarly, participants using the original CoEdPilot comprise Control Group 2 (CG2), and those using CoEdPilot with EditFlow constitute Experiment Group 2 (EG2).

7.4.3 Setup. Participants first complete a warm-up task that follows the same procedure as the three formal tasks. The warm-up has no time limit and is intended to familiarize them with the extension and study workflow. For each task, we provide:

(1) The complete repository;
(2) Detailed edit descriptions, with the necessary background knowledge, including code functionality and method interface;
(3) An initial edit that serves as a starting point, guiding participants toward completing the remaining edits;
(4) Test cases that participants can execute to verify whether their edits meet the expected outcomes. A task is considered complete once all tests pass.

At the end of the study, participants are required to share screen recordings throughout their completion of the three tasks, along with the instrumentation data collected during their interaction.

7.4.4 Editing Task. Participants are required to complete three real-world editing tasks derived from commits in top-starred GitHub repositories. The three commits are intentionally selected to span distinct edit-topology structures observed in real-world multi-location edits, allowing us to examine both the effectiveness and the boundaries of flow-aware optimization. Tasks are simplified by providing project-specific knowledge and removing distracting edits, so that participants can complete them without requiring specialized background knowledge. Each task is limited to 30 minutes.

- **Task 1 (Moderately easy)**: Originating from [21], this commit adds a new argument `allow_scraping` to `fetch_url()`, controlling whether the system may retrieve source code from an external URL for bug localization upon a Sentry-detected error. In the call chain of `fetch_url()`, there are syntax-based modifications across three files with a total of 8 changes. We select the implementation involving the concrete use of the `allow_scraping` parameter as the initial edit, from which the remaining edits can be inferred by users.
- **Task 2 (Hard)**: Originating from [30], this commit ensures that when `-keep-focus` is used, the tab and window history remain consistent, preserving the correct focus transition behaviour by removing non-user-initiated entries from the history stack. This update involves 8 changes across 4 files, with the update of calling function `boss.set_active_window()` as the initial edit.
- **Task 3 (Easy)**: Based on [53], this commit focuses on refactoring a commonly used `if` condition into a standalone function. The change spans 14 modifications across 2 files. We treat the extracted function as the initial edit, and the user's task is to locate the corresponding `if` condition instances and replace them with this function.

7.4.5 Metric. We assess the user study along both quantitative and qualitative dimensions. Quantitatively, we measure the **average time cost** for user efficiency, compute **Mann-Whitney U test p-values** using permutation testing (10,000 resamples) to determine statistical significance between independent groups ($p < 0.05$ indicates significance), and calculate **effect sizes** (r) derived from the standardized U statistic to quantify the magnitude of differences ($r \geq 0.5$ indicates a large effect). Qualitatively, we randomly sample screen recordings from each group and examine participants' editing behaviours in conjunction with instrumentation data.

7.4.6 Results and Analysis. The performance of all participants is shown in Table 7, with the following analysis:

Table 7. Performance of Claude Code with EditFlow (EG1), CoEdPilot with EditFlow (EG2), Claude Code (CG1), and CoEdPilot (CG2), time cost in minutes

EG1	T1	T2	T3	CG1	T1	T2	T3	EG2	T1	T2	T3	CG2	T1	T2	T3
P1	6.42	8.29	3.82	**P9**	7.67	11.13	3.05	**P17**	6.83	7.61	3.59	**P25**	12.76	14.57	4.84
P2	8.05	8.93	5.33	**P10**	6.99	10.32	4.59	**P18**	8.43	6.10	4.03	**P26**	9.15	12.27	4.57
P3	6.59	6.45	3.36	**P11**	7.07	15.79	4.93	**P19**	7.48	5.86	5.29	**P27**	8.22	11.56	4.73
P4	5.45	6.33	3.89	**P12**	8.69	12.77	3.98	**P20**	6.42	6.45	4.97	**P28**	7.28	15.88	3.64
P5	7.05	6.78	3.23	**P13**	8.62	15.78	5.66	**P21**	4.32	5.91	3.31	**P29**	8.84	12.31	5.57
P6	8.07	10.94	3.40	**P14**	7.49	12.33	3.79	**P22**	5.66	8.03	2.10	**P30**	6.92	11.12	4.33
P7	5.88	8.20	3.15	**P15**	7.93	9.37	4.47	**P23**	7.24	6.98	3.64	**P31**	8.81	14.58	4.93
P8	7.51	8.10	2.08	**P16**	5.64	11.77	2.62	**P24**	9.62	5.58	3.66	**P32**	17.00	12.39	2.64
Avg.	**6.88**	**8.00**	**3.53**	**Avg.**	**7.51**	**12.41**	**4.14**	**Avg.**	**7.00**	**6.57**	**3.82**	**Avg.**	**9.87**	**13.09**	**4.41**

- **Task 1:** EG1 marginally outperforms CG1 without statistical significance ($p = 0.1966, r = 0.289$), whereas EG2 outperforms CG2 with statistical significance ($p = 0.0318, r = 0.525$)
- **Task 2:** The superiority of EG1 over CG1 is statistically significant ($p = 0.0004, r = 0.788$), and a consistent trend is also observed for EG2 compared with CG2 ($p = 0.0004, r = 0.840$).
- **Task 3:** EG1 marginally outperforms CG1, though the difference is not statistically significant ($p = 0.2186, r = 0.289$). A similar trend is observed when comparing EG2 with CG2 ($p = 0.2360, r = 0.289$).

Why is EditFlow 's advantage over the original system not evident in Task 1? Task 1 is of low-to-moderate difficulty, where most edits follow clear syntactic propagation patterns, These patterns constrain the search space for plausible edits and make it easier for users to maintain a clear mental flow. As a result, strong models such as Claude already produce fewer flow-breaking recommendations, leaving limited room for further improvement. In contrast, weaker models like CoEdPilot are more prone to incorrect or mistimed suggestions, allowing EditFlow to deliver statistically significant gains. These results indicate that EditFlow is effective at narrowing the performance gap between large and small models in edit recommendation tasks.

Why does EditFlow outperform both original systems in Task 2? Task 2 is substantially more challenging, requiring a deeper understanding of the codebase and permitting only a small set of valid edit sequences. Moreover, it involves several functions with similar names: `set_active_tab()` (ground truth), `_set_active_tab()` (distraction), and `set_active_tab_idx()` (distraction), increases the likelihood of confusion and flow-violating recommendations from both Claude and CoEdPilot. In this setting, EditFlow provides consistent and statistically significant improvements for both systems. Without EditFlow, the original systems tend to produce flow-violating suggestions, imposing a high verification burden on users. For example, user P31 had 13 locations where each verification took more than 10 seconds, including 4 locations that each took more than 20 seconds and 2 that exceeded 30 seconds. If the user does not strictly verify the recommendations, the incorrect edits are usually only discovered during test execution, requiring substantial time for debugging (approximately 60% of the total editing time for P13).

Why does EditFlow not demonstrate a statistically significant advantage in Task 3? Task 3 is a uniform refactoring task involving 14 highly similar edits, where an *if* condition is consistently replaced by a call to a refactored function, which introduce no functionality or semantic side effects that affect the surrounding code. By construction, this scenario induces an almost fully connected partial-order graph in which edits are largely interchangeable, making the task inherently order-insensitive. As a result, nearly any edit ordering aligns with developers' mental flow, and both Claude and CoEdPilot already generate correct recommendations with few flow violations. This task, therefore, serves as a boundary case where flow-aware optimization is not expected to yield substantial benefits, consistent with our observed results.

8 Failure Analysis and Metric Implications

While EditFlow consistently improves multiple flow-aware and flow-agnostic metrics, we observe several recurring scenarios in which EditFlow itself fails to make correct flow-aware decisions. In this section, we focus specifically on analyzing EditFlow's failure modes. For each failure category, we qualitatively explain the underlying mechanism in EditFlow's design or assumptions that lead to incorrect decisions, illustrate representative examples, and discuss how such failures propagate to downstream evaluation metrics.

8.1 Failure Mode I: False Rejection due to k-Context Sensitivity

Mechanism. In this scenario, EditFlow incorrectly filters out flow-aligned subsequent edit recommendations. This failure arises when EditFlow evaluates flow continuity using only the most recent edit, effectively assuming a 1-step editing context. However, in practice, a valid subsequent edit may exhibit flow continuity with earlier edits rather than with the most recent edit alone. When such broader contextual information lies outside the limited context window considered by EditFlow, the candidate edit appears locally misaligned and is therefore incorrectly rejected, despite being consistent with the developer's overall editing flow.

Table 8. Failure analysis example 1: false rejection

File: `tests/components/fritzbox/test_binary_sensor.py`	
h_{-1}	`  from homeassistant.const import (`
	`      ...`
	`-     STATE_OFF,`
	`      STATE_ON,`
h' (h_0)	`+     STATE_UNAVAILABLE,`
	`  )`
h_{-2}	`      ...`
	`      assert state`
	`-     assert state.state == STATE_OFF`
	`+     assert state.state == STATE_UNAVAILABLE`
	`      ...`

Example. This example is observed during the digital twin evaluation of Claude combined with EditFlow, where the simulation replays the editing process of a real-world commit[4]. During the simulated editing sequence, the system iteratively provides recommendations based on the evolving edit context. The relevant contextual information is summarized in Table 8. Here, h_{-x} denotes the prior edit that has already been applied at the current simulation step, indexed relative to the current time. Specifically, h_{-1} is the most recently applied edit, h_{-2} is the edit immediately before it, and so on. The symbol h' denotes the edit recommended by Claude at the current step, which corresponds to the expected next edit at the current progress and is denoted as h_0. The recovered partial-order relations can be expressed as: $h_{-1} \sim h_{-2}$, $h_{-2} \sim h_0$, and $h_{-1} \perp h_0$. Under the recovered graph, h_0 is a one-hop successor of h_{-2} and therefore constitutes a flow-keeping (and correct) subsequent edit in the global context.

However, during flow-aware filtering, EditFlow adopts a 1-context sensitivity assumption and does not evaluate $\lambda(h, h')$ for all $h \in h_{<0}$. Instead, it only evaluates the relation $\lambda(h_{-1}, h')$, considering the candidate edit solely with respect to the most recent prior edit. In this case, EditFlow yields $\lambda(h_{-1}, h') = \perp$, as the local flow continuity between h_{-1} and h' is not apparent. As a result,

[4]https://github.com/home-assistant/core/commit/4241c4c

the critical mental-flow context established by h_{-2} is ignored, leading EditFlow to misclassify h' as non-flow-keeping edit, and to incorrectly filter out an otherwise correct recommendation.

Metric implications. Filtering out a flow-keeping edit directly reduces the proportion of flow-keeping recommendations in the final evaluation. Although a rejected recommendation may enter the recycling loop and be deferred until a more suitable moment (details may refer to Section 6.2), in this example, the rejected edit continues to suffer from the 1-context sensitivity issue discussed above, namely that its flow relation is only evaluated against the most recent prior edit. As no subsequent edits occur that would render h_0 a one-hop successor of the immediately preceding edit, the recommendation is eventually discarded, resulting in a False Negative in flow-graph-independent metrics. Specifically, a correct recommendation is permanently removed from the candidate set, leading to simultaneous decreases in both precision and recall.

8.2 Failure Mode II: Acceptance of seemingly Flow-Keeping but Incorrect Edits

Mechanism. In this scenario, EditFlow fails to filter out certain recommendations because it deems them to be flow-keeping based on local flow continuity. However, the inferred flow continuity does not correspond to the user's intended editing flow, leading EditFlow to incorrectly accept edits that are locally coherent but misaligned with the user's actual intent. This failure typically stems from the ambiguity of editing intentions, where a single prior edit can logically lead to multiple divergent flow paths based on different intentions. While EditFlow is designed to verify flow continuity, it primarily validates whether a recommendation constitutes *a* plausible continuation, rather than ensuring it is *the specific* continuation intended by the developer. When a recommendation aligns with a logical but unintended flow branch (e.g., deletion instead of modification), EditFlow lacks the sufficient intent awareness to distinguish it from the ground truth, leading to the acceptance of a locally coherent but globally incorrect edit under the developer's intended flow.

Table 9. Failure analysis example 2: false acceptance

File: `zerver/lib/scheduled_messages.py`		
h_{-1}	1	`  from homeassistant.const import (`
	2	`      ...`
	3	`-     StreamScheduledMessageAPI,`
	4	`      UserProfile,`
	5	`  )`
File: `zerver/models.py`		
h_0	6	`- class StreamScheduledMessageAPI(TypedDict):`
	7	`+ class APIScheduledStreamMessageDict(TypedDict):`
	8	`      scheduled_message_id: int`
	9	`      to: int`
	10	`      ...`
h'	6	`- class StreamScheduledMessageAPI(TypedDict):`
	7	`-     scheduled_message_id: int`
	8	`-     to: int`
	9	`-     type: str`
	10	`      content: str`

Example. This example[5] is observed during the digital twin evaluation of Cursor combined with EditFlow. The relevant edits are available in Table 9. Here, h_{-1} denotes the most recently applied prior edit at the current simulation step, h_0 denotes the expected ground-truth next edit, and h' denotes the edit recommended by Cursor at the current step. In this case, h_{-1} removes the import of

[5]https://github.com/zulip/zulip/commit/02fafb03

`StreamScheduledMessageAPI`. The ground-truth next edit h_0 then renames the corresponding class in `zerver/models.py`, changing `StreamScheduledMessageAPI` to `APIScheduledStreamMessage Dict`. However, Cursor recommends h', which instead removes (or substantially deletes) the existing definition. EditFlow identifies the transition from h_{-1} to h' as flow-continuous, e.g., $\lambda(h_{-1}, h') \in \{\prec, \sim\}$, since both edits operate on the same API and appear cognitively coherent as a follow-up to removing its import. As a result, EditFlow does not filter out h'.

Nevertheless, h' does not align with the developer's actual intent in this commit, which is to consistently *rename* the API type rather than delete its definition. This constitutes a false acceptance: a recommendation that appears flow-keeping under the immediate context, yet is incorrect and flow-breaking with respect to the human-intended ground-truth edit h_0.

Metric implications. As h' is accepted as an incorrect recommendation with respect to ground-truth, it is counted as a false positive and classified as a flow-breaking outcome with respect to the human ground-truth edit sequence. Consequently, this case increases the proportion of flow-breaking recommendations and simultaneously lowers precision in flow-graph-independent metrics. Meanwhile, recall is not directly affected by this false positive.

9 Threats to Validity

We discuss three major types of threats to the validity of our study.

External validity. Our benchmark primarily consists of 100 Python commits and one industrial dataset that cannot be publicly released due to compliance restrictions. The dataset uses Git commits solely as a data source and does not reflect GitHub-specific workflows such as pull requests or code reviews. Although prior work suggests that developer editing behaviours are broadly consistent across programming languages [18, 51, 56], we acknowledge that our data composition may limit the generalizability of our findings to other programming languages, development workflows, or industrial settings. Validating the mental-flow model on additional languages and non-GitHub workflows would further strengthen external validity. To mitigate this concern, we selected projects covering diverse functionalities and repository sizes, and we plan to extend our benchmark to multi-language repositories and additional industrial contexts in future work.

Construct validity. The notion of *mental-flow alignment* is inherently abstract, and our operationalization through the Keep/Jump/Revert/Break taxonomy may only approximate developers' cognitive states. Although we employed double annotation and inter-annotator agreement checks to ensure labelling reliability, some subjectivity is inevitable.

Internal validity. First, the digital-twin simulation assumes that developers always make correct decisions, while abstracting away other cognitive and behavioural factors, such as developers' trust in recommendations and individual programming habits. These simplifying assumptions reduce the complexity of real developer behaviour and may therefore overestimate the effectiveness of flow-aware optimization. Nevertheless, this abstraction provides a reproducible and controlled environment for benchmarking heterogeneous agents (e.g., Cursor vs. Claude) by eliminating human variance.

Second, the collected edit-order data, while reflecting real developer actions, does not guarantee the most mental-flow-compatible editing sequence. Developers may occasionally follow suboptimal or counter-intuitive paths due to habits, local context, or external constraints. Although our manual inspection suggests such cases are rare, this imperfection may introduce noise into the learned flow model.

Third, in **RQ2**, we estimate ordering quality based on the proportion of *violations* against the observed ground-truth edit order. This evaluation protocol may introduce an optimistic bias, as some false-positive relations in the predicted partial order cannot be falsified by a single observed editing trajectory. Consequently, the measured performance may slightly overestimate the true ordering accuracy. Nevertheless, this controlled setup enables consistent comparison across systems by eliminating human variance.

Fourth, our approach relies on large language models for edit order inference, which are inherently stochastic. As a result, identical inputs may occasionally yield different inferred partial orders, potentially introducing variability into flow-aware filtering and downstream evaluation. While this randomness does not affect the conceptual validity of our approach, it may influence the stability of individual predictions. In practice, such variability can be mitigated through standard engineering techniques, such as lowering decoding temperature, applying majority voting across multiple runs, or aggregating predictions over multiple samples. We leave a systematic exploration of these stabilization strategies to future work.

Future work will explore incorporating more nuanced cognitive and behavioral factors into the simulation, as well as real-world IDE traces and multi-path simulation, to better approximate realistic developer behavior.

10 Related Work

10.1 Code Edit Recommendation

Static analysis methods for code edit recommendations primarily rely on extracting edit patterns. For instance, CCDemon [34] exploits clone differences for suggesting pasted-code modifications, Overwatch [58] leverages IDE-logged traces, Pyevolve [17] mines frequently repeated code change patterns (CPATs) from the version histories of Python projects, and AppEvolve [19] extracts API changes from other applications. Solutions of this kind prioritize efficiency and syntactic correctness. However, they suffer from poor generalization when extended to more diverse editing scenarios.

Recent advances in large language models have significantly reshaped software engineering, enabling applications such as code completion, bug fixing, debugging, refactoring, and automated testing [12, 33, 35, 39, 49, 52]. Beyond single-shot tasks, recent work has increasingly explored code edit recommendation, such as CoditT5 [57], CODEEDITOR [31], and CCT5 [32], propose pre-trained models specifically designed for code editing tasks. To further enhance performance, various forms of contextual retrieval have been explored; for instance, GrACE [24] incorporates relevant prior edits, while SARGAM [37] adopts a Search-Generate-Modify paradigm. Other efforts, including Codit [14] and Recoder [59], integrate abstract syntax tree (AST) representations to improve the model's understanding of code structure and syntax. Moreover, recent work such as CodePlan [8] integrates LLMs with static analysis tools, including dependency graphs, to support more accurate reasoning over code changes. However, these approaches remain largely confined to evaluations on offline benchmarks and have not been designed or validated for project-wise subsequent edit recommendation in realistic development scenarios.

For end-to-end editing scenarios, the most relevant work in the academic community is CoEdPilot [36], which proposes a Transformer-based framework for jointly localizing and generating subsequent edits. Building upon this framework, [38] incorporates static analysis tools and fine-grained code edit representations to further enhance performance. Meanwhile, numerous products have been developed in the industry to support similar workflows. Cursor [2] was the first IDE to support proactive subsequent edit suggestions, introduced in November 2023. In this design, suggested edits are proactively rendered as ghost text. Developers can press Tab once to navigate to the suggestion and press Tab a second time to accept it. Cursor also supports the command-response paradigm, in which users describe their intentions, and the agent actively searches inside the codebase and returns corresponding code edit suggestions along with an explanation. The above 2 solutions are then quickly adopted by other products, including Copilot [22] and Windsurf [4]. More recently, agentic approaches have emerged that operate directly through command-line interfaces. Claude

Code [6], released in late 2024, enables developers to delegate coding tasks to an AI agent that can read, write, and execute code autonomously within a specified project directory. Other similar products include Gemini CLI [23] and Qwen Code [50].

10.2 Developer Flow and Productivity

Flow as a core construct in productivity. Mental flow is a well-established psychological construct defined as a mental state in which a person performing an activity is fully immersed in a feeling of energized focus, full involvement, and enjoyment [15]. Since its introduction, flow has been widely adopted in software engineering research as a key lens for understanding developer productivity and experience. The SPACE framework, now widely recognized as an industry standard for evaluating developer productivity, treats *Efficiency and Flow* as one of its five independent dimensions [20]. Similarly, the DevEx framework [44] distills developer productivity into three core dimensions: feedback loops, cognitive load, and flow state, positioning flow as a first-class driver of effective software development. Together, these frameworks establish flow not as a subjective afterthought, but as a foundational construct for reasoning about productivity in both academic and industrial contexts.

Quantitative relationship between flow and productivity. Prior work provides strong quantitative evidence linking sustained flow to productivity gains and interruptions to substantial productivity loss. Meyer *et al.* report that over 50% of developers associate productive workdays with uninterrupted flow and minimal context switching [42]. Large-scale GitHub and DevEx studies further show that developers who sustain deep, uninterrupted work report 30–50% higher perceived productivity and up to 20% higher innovation [28, 44]. Conversely, the cost of breaking flow is disproportionate to the frequency of interruptions. After an interruption, developers require on average 23 minutes and 15 seconds to fully resume their original task (*recovery tax*) [40]. Even a modest interruption rate of 10% can nearly double total task completion time, indicating a non-linear, compounding effect of flow disruption on productivity [54].

Disruptions of flow by modern coding tools. While flow is beneficial, a growing body of evidence shows that modern development tools, particularly AI-based coding assistants, can both support and disrupt it. Beyond their well-known positive effects, prior studies document substantial downsides of LLM-based coding assistants. Unwanted or poorly timed suggestions, verbose outputs, interface switching, and the need to constantly verify generated code introduce frequent interruptions that break developers' flow continuity [43]. Controlled and observational studies further demonstrate that such interruptions lead to excessive context switching between thinking, reading, and debugging, resulting in increased cognitive load and degraded task performance [48]. Qualitative evidence from both professional and novice developers highlights frustration, loss of focus, and exhaustion when tools intrude into the developer's mental flow, for example by flooding the screen with unsolicited suggestions or requiring sustained attention to tool interaction rather than problem solving [47, 48]. These findings align with earlier work showing that interruptions, delays, and tool friction are among the primary barriers to experiencing flow in software engineering tasks [27, 42].

Taken together, prior work establishes flow as a central, measurable determinant of developer productivity, quantifies its impact, and provides converging evidence that poorly aligned tooling can significantly disrupt flow. This body of research motivates the need for developer assistance systems that explicitly preserve cognitive continuity rather than inadvertently fragment it.

11 Conclusion

This paper presents EditFlow, the first framework for benchmarking and optimizing code edit recommendation systems from the perspective of developers' mental flow. We identified an essential gap between technical accuracy and developer productivity, showing that existing assistants often disrupt developers' cognitive continuity due to their reliance on static commit snapshots.

To bridge this gap, EditFlow integrates three synergistic components: a prompt auto-tuning mechanism that reconstructs edit orders, a digital twin for flow-aware evaluation, and a unified optimization wrapper for filtering and re-ranking recommendations. Extensive experiments on annotated and industrial datasets show that EditFlow improves edit-order reconstruction accuracy by 63.81%, reduces flow violations by 75%, and increases recommendation precision by 66.99%. A controlled user study with 32 participants further confirms 25.11% faster task completion and significantly higher perceived recommendation quality.

Overall, EditFlow establishes flow-awareness as a new dimension for evaluating and enhancing AI-assisted code editing. By reconstructing and simulating developers' editing processes, our framework provides a foundation for future research on cognitively aligned recommendation systems and seamless human-AI code collaboration.

Data-Availability Statement

The source code, auto-tuned prompt, dataset, and experiment results are available at [3]. Please note that a portion of the dataset provided by our industry collaborator cannot be publicly released due to compliance and confidentiality agreements.

Acknowledgments

This research is supported in part by the National Natural Science Foundation of China (62572300), the Ministry 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.

References

[1] 2025. Claude Code SDK - Anthropic. https://docs.anthropic.com/en/docs/claude-code/sdk.

[2] 2025. Cursor - The AI Code Editor. https://www.cursor.com/. [Accessed 2025-03-31].

[3] 2025. EditFlow — sites.google.com. https://sites.google.com/view/editflow. [Accessed 2025-08-02].

[4] 2025. Windsurf Editor. https://windsurf.com/editor. [Accessed 2025-03-31].

[5] Anthropic. 2025. Claude 3.7 Sonnet and Claude Code. https://www.anthropic.com/news/claude-3-7-sonnet. [Accessed 2025-08-31].

[6] Anthropic. 2025. Claude Code: Deep coding at terminal velocity. https://www.anthropic.com/claude-code. [Accessed 2025-03-31].

[7] Anthropic. 2025. Introducing Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet. [Accessed 2025-08-31].

[8] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 675–698. doi:10.1145/3643757

[9] Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. 2025. Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. *arXiv preprint arXiv:2507.09089* (2025). doi:10.48550/arXiv.2507.09089

[10] Branden Butler, Sixing Yu, Arya Mazaheri, and Ali Jannesari. 2024. Pipeinfer: Accelerating llm inference using asynchronous pipelined speculation. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–19. doi:10.1109/SC41406.2024.00046

[11] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774* (2024).

doi:10.48550/arXiv.2401.10774

[12] Yufan Cai, Zhe Hou, David Sanán, Xiaokun Luan, Yun Lin, Jun Sun, and Jin Song Dong. 2025. Automated program refinement: Guide and verify code large language model with refinement calculus. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 2057–2089. doi:10.1145/3704905

[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. doi:10.1109/TSE.2020.3020502

[15] Mihaly Csikszentmihalyi and Mihaly Csikzentmihaly. 1990. *Flow: The psychology of optimal experience*. Vol. 1990. Harper & Row New York.

[16] Cursor. 2025. Cursor CLI Overview. https://docs.cursor.com/en/cli/overview. [Accessed 2025-08-31].

[17] Malinda Dilhara, Danny Dig, and Ameya Ketkar. 2023. Pyevolve: Automating frequent code changes in python ml systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 995–1007. doi:10.1109/ICSE48619.2023.00091

[18] Sedick David Baker Effendi, Berk Çirisci, Rajdeep Mukherjee, Hoan Anh Nguyen, and Omer Tripp. 2023. A language-agnostic framework for mining static analysis rules from code changes. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 327–339. doi:10.1109/ICSE-SEIP58684.2023.00035

[19] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage update for Android apps. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 204–215. doi:10.1145/3339066

[20] Nicole Forsgren, Margaret-Anne Storey, Chandra Maddila, Thomas Zimmermann, Brian Houck, and Jenna Butler. 2021. The SPACE of Developer Productivity: There's more to it than you think. *Queue* 19, 1 (2021), 20–48. doi:10.1145/3453928

[21] getsentry/sentry. 2025. Ensure releases can be used when scraping is disabled. https://github.com/getsentry/sentry/commit/de2de4a9da751041baa329be551ee4e0b12021d9. [Accessed 2025-08-31].

[22] GitHub. 2023. GitHub Copilot. https://github.com/features/copilot [Accessed 2025-08-31].

[23] Google. 2025. google-gemini/gemini-cli: An open-source AI agent that brings the power of Gemini directly into your terminal. https://github.com/google-gemini/gemini-cli. [Accessed 2025-03-31].

[24] 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. doi:10.1145/3611643.3616253

[25] Zhen Huang, Zengzhi Wang, Shijie Xia, and Pengfei Liu. 2024. OlympicArena Medal Ranks: Who Is the Most Intelligent AI So Far? arXiv:2406.16772 [cs.CL] doi:10.48550/arXiv.2406.16772

[26] Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. 2024. Language models for code completion: A practical evaluation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639138

[27] Sabine Janssens and Vadim Zaytsev. 2022. Go with the flow: software engineers and distractions. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 934–938. doi:10.1145/3550356.3559101

[28] Eirini Kalliamvakou and GitHub Staff. 2025. Yes, good DevEx increases productivity. Here is the data. https://github.blog/news-insights/research/good-devex-increases-productivity/. [Accessed 2025-12-31].

[29] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *The Twelfth International Conference on Learning Representations*.

[30] kovidgoyal/kitty. 2025. When using –keep-focus ensure active history list is not affected. https://github.com/kovidgoyal/kitty/commit/c4c62c15. [Accessed 2025-08-31].

[31] 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 Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–22. doi:10.1145/3597207

[32] 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). doi:10.1145/3611643.3616339

[33] Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. 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*. 1068–1080. doi:10.1145/3468264.3468619

[34] 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. doi:10.1145/2786805.2786871

[35] Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. 2020. 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*. 440–451. doi:10.1145/3395363.3397358

[36] 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* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 466–478. doi:10.1145/3650212.3652142

[37] Changshu Liu, Pelin Cetin, Yogesh Patodia, Baishakhi Ray, Saikat Chakraborty, and Yangruibo Ding. 2024. Automated code editing with search-generate-modify. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 398–399. doi:10.1145/3639478.3643124

[38] Chenyan Liu, Yun Lin, Yuhuan Huang, Jiaxin Chang, Binhang Qi, Bo Jiang, Zhiyong Huang, and Jin Song Dong. 2025. Learning Project-wise Subsequent Code Edits via Interleaving Neural-based Induction and Tool-based Deduction. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1377–1389. doi:10.1109/ASE63991.2025.00117

[39] Ruofan Liu, Xiwen Teoh, Yun Lin, Guanjie Chen, Ruofei Ren, Denys Poshyvanyk, and Jin Song Dong. 2025. GUIPilot: A Consistency-Based Mobile GUI Testing Approach for Detecting Application-Specific Bugs. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 753–776. doi:10.1145/3728909

[40] Gloria Mark, Daniela Gudith, and Ulrich Klocke. 2008. The cost of interrupted work: more speed and stress. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. 107–110. doi:10.1145/1357054.1357072

[41] Mykola Maslych, Mohammadreza Katebi, Christopher Lee, Yahya Hmaiti, Amirpouya Ghasemaghaei, Christian Pumarada, Janneese Palmer, Esteban Segarra Martinez, Marco Emporio, Warren Snipes, et al. 2025. Mitigating response delays in free-form conversations with LLM-powered intelligent virtual agents. In *Proceedings of the 7th ACM Conference on Conversational User Interfaces*. 1–15. doi:10.1145/3719160.3736636

[42] André N Meyer, Thomas Fritz, Gail C Murphy, and Thomas Zimmermann. 2014. Software developers' perceptions of productivity. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 19–29. doi:10.1145/2635868.2635892

[43] Amr Mohamed, Maram Assi, and Mariam Guizani. 2025. The impact of LLM-assistants on software developer productivity: A systematic literature review. *arXiv preprint arXiv:2507.03156* (2025). doi:10.48550/arXiv.2507.03156

[44] Abi Noda, Margaret-Anne Storey, Nicole Forsgren, and Michaela Greiler. 2023. DevEX: What actually drives productivity? *Commun. ACM* 66, 11 (2023), 44–49. doi:10.1145/3610285

[45] OpenAI. 2025. OpenAI o1-mini | OpenAI. https://openai.com/index/openai-o1-mini-advancing-cost-efficient-reasoning. [Accessed 2025-03-31].

[46] Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. 2024. Optimizing instructions and demonstrations for multi-stage language model programs. *arXiv preprint arXiv:2406.11695* (2024). doi:10.48550/arXiv.2406.11695

[47] Veronica Pimenova, Sarah Fakhoury, Christian Bird, Margaret-Anne Storey, and Madeline Endres. 2025. Good Vibrations? A Qualitative Study of Co-Creation, Communication, Flow, and Trust in Vibe Coding. *arXiv preprint arXiv:2509.12491* (2025). doi:10.48550/arXiv.2509.12491

[48] James Prather, Brent N Reeves, Paul Denny, Brett A Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It's weird that it knows what i want": Usability and interactions with copilot for novice programmers. *ACM transactions on computer-human interaction* 31, 1 (2023), 1–31. doi:10.1145/3617367

[49] Binhang Qi, Yun Lin, Xinyi Weng, Yuhuan Huang, Chenyan Liu, Hailong Sun, and Jin Song Dong. 2025. Intention-Driven Generation of Project-Specific Test Cases. *arXiv preprint arXiv:2507.20619* (2025). doi:10.48550/arXiv.2507.20619

[50] Qwen. 2025. Qwen3-Coder: Agentic Coding in the World. https://qwenlm.github.io/blog/qwen3-coder/. [Accessed 2025-03-31].

[51] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 155–165. doi:10.1145/3126905

[52] Xiaoxue Ren, Xinyuan Ye, Yun Lin, Zhenchang Xing, Shuqing Li, and Michael R Lyu. 2023. 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*. 1320–1332. doi:10.1145/3611643.3616269

[53] sqlmapproject/sqlmap. 2025. fix for a bug reported by ToR: "AttributeError: 'NoneType' object has no attribute 'isdigit'". https://github.com/sqlmapproject/sqlmap/commit/2cc167a42eb18030536107803dfbca6a71694f65. [Accessed 2025-08-31].

[54] John Sum and Kevin Ho. 2015. Analysis on the Effect of Multitasking. In *2015 IEEE international conference on systems, man, and cybernetics*. IEEE, 204–209. doi:10.1109/SMC.2015.48

[55] Zhuofan Wen, Shangtong Gui, and Yang Feng. 2024. Speculative decoding with CTC-based draft model for LLM inference acceleration. *Advances in Neural Information Processing Systems* 37 (2024), 92082–92100. doi:10.48550/arXiv.2412.00061

[56] Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. Multilingual code co-evolution using large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 695–707. doi:10.1145/3611643.3616350

[57] 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*. doi:10.48550/arXiv.2208.05446

[58] 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. doi:10.1145/3563302

[59] 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. doi:10.1145/3468264.3468544

Received 2025-10-10; accepted 2026-02-17