From Characters to Structure: Rethinking Real-Time Collaborative Programming Models

Leon Freudenthaler
Computer Science and Digital Communications
Hochschule Campus Wien/TU Wien
Vienna, Austria
leon.freudenthaler@hcw.ac.at; ORCID:0009-0001-4063-4396

Bernhard Taufner
Computer Science and Digital Communications
Hochschule Campus Wien
Vienna, Austria
bernhard.taufner@hcw.ac.at

Karl Michael Göschka
Institute of Information Systems Engineering
TU Wien/UAS Technikum Wien
Vienna, Austria
karl.goeschka@tuwien.ac.at; ORCID: 0000-0001-6419-2732

***Abstract*—Multiple programming tasks require synchronous collaboration between developers, giving rise to real-time collaborative programming tools that enable simultaneous editing of shared source code. However, most existing tools operate at the text level, propagating every keystroke–including syntactically invalid ones–without considering program structure. This results in excessive communication overhead, frequent propagation of build-breaking states, and poor synchronization. A major consequence is noticeable lag, especially under unstable network conditions, as collaborators are overwhelmed with unnecessary updates that disrupt their workflow and degrade the shared coding experience. In this paper, we introduce a novel structure-aware propagation model that transmits only syntactically valid code changes. For evaluation we implemented our tool as an IntelliJ plugin and evaluate it against three industry-standard tools–VS Code Live Share, Code With Me, and Replit–across eight representative programming scenarios. Our results show that it significantly lowers the number and size of propagated messages while maintaining consistent, buildable program states. Our findings demonstrate the potential of structure-aware propagation as a foundation for the next generation of real-time collaborative programming environments.**

***Index Terms*—real-time collaborative programming, ast-based operations, structure-aware propagation, synchronous collaboration systems**

I. Introduction

In the daily workflow of software development, developers often encounter situations where they must guide a co-developer through specific programming tasks. Such instructions might take the form: *"Navigate to line 234... and now type L-O-G..."*. In these moments, the desire to simply take over the keyboard is palpable. A promising response to these scenarios is the emergence of *real-time collaborative programming (RCP)* tools. These systems are specifically designed to support development tasks that require high levels of synchronous interaction within a fully integrated programming environment. [1]

Software development typically requires collaboration among developers as they engage in writing, testing, debugging, and maintaining source code [2]–[4]. While asynchronous collaboration is dominant and well-supported through version control systems like Git, it does not suit all workflows. Asynchronous models rely on explicit commits and manual merges, which are often too rigid or delayed for dynamic, fast-paced tasks [5]. However, many development scenarios require real-time interaction. These include pair and mob programming sessions [6], live coding interviews, team code reviews, educational scenarios involving student-teacher or peer group collaborations [7], and live troubleshooting sessions between support engineers at a customer site and senior engineers at a remote headquarters. Moreover, real-time interactions are crucial when multiple developers must simultaneously modify a tightly coupled section of code – a single, indivisible source artifact [1] . Addressing these scenarios in an asynchronous manner introduces communication overhead and frequent merge conflicts that must be resolved manually [8].

To fill this gap, RCP tools like *JetBrains Code With Me* [9], *VS Code Live Share* [10] and *Replit* [11] emerged. These environments allow multiple developers to simultaneously edit shared codebases. They fall under the categorization of synchronous groupware defined as *"the class of applications in which two or more people collaborate in what they perceive to be real time"* [12]. While these systems technically operate in a synchronous or near real-time fashion, the term *"real-time"* has become widely adopted in the research community to describe this class of collaborative tools, despite variations in actual latency and responsiveness. Therefore, we adopt the term *real-time collaborative programming* environments in this paper, treating synchronous and real-time as synonymous.

Presented tools propagate code changes at character or text level, leveraging Operational Transformation (OT) [13] or Conflict-free Replicated Data Types (CRDT) [14]–[17] for automatic synchronization. While existing OT and CRDT algorithms may be sufficient for real-time collaborative text editing, they ignore program structure and semantic correctness, leading to deeper problems during collaborative programming

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00221

sessions [18]–[20]. As developers write code, it regularly transitions through intermediate states that are syntactically invalid. Propagating such low-level changes in real time can lead to broken builds across collaborators' environments, disrupting the development process and impairing key features like code analysis, autocompletion, and execution [8], [21].

Beyond text-based synchronization, professional RCP tools use a host-participator model, where the host retains full code access and control, while participators access files only on demand. This further limits collaboration capabilities by restricting tooling, execution, and resilience [22].

Existing research and a recent study on developer experience [1] highlight three major pain points in current RCP systems:

- **High Communication Overhead:** Every keystroke–including incomplete or invalid code–is propagated immediately. This "chattiness" leads to excessive network traffic and often visible lag [23]. In a recent survey, 71% of developers cited lagging and unstable networks as major RCP limitation [1].
- **Propagation of Buildbreaking Intermediate States:** Text-based propagation means that collaborators constantly receive incomplete or invalid program states. These transient buildbreaking conditions interrupt the development flow, inhibit local testing, and degrade understanding–especially as the number of collaborators grows [24].
- **Structural Conflict and Poor Synchronization Semantics:** Merge conflicts and inconsistencies are common when developers simultaneously edit related code in different files (e.g., modifying a function declaration and its call site). Existing systems lack semantic awareness and cannot resolve or prevent these issues gracefully [17], [18], [25].

These limitations indicate a deeper issue: current tools operate at an **inappropriate level of abstraction for collaborative real-time programming**. By treating code as plain text, they disregard the structured, language-aware nature of programming and ultimately restrict both user experience and system capabilities.

We propose *AstFlow*, a novel structure-aware propagation approach that tackles presented limitations. Instead of using text we make use of the abstract syntax tree (AST) as our core propagation model.

Although AST-based granularity also opens the door for (semi-)structured conflict resolution and semantic merging strategies [26], [27], the focus of this paper is on the propagation mechanism and the supporting system architecture. We do not attempt to solve the full synchronization problem yet. Instead, we lay the groundwork by showing that semantically valid update propagation alone can significantly improve communication efficiency and developer experience in RCP environments.

To evaluate the proposed model, we developed a benchmark comprising eight simple yet representative programming scenarios that commonly arise during collaborative editing. These scenarios were executed in controlled sessions using three prominent industry RCP tools–*VS Code Live Share*, *Code With Me*, and *Replit*–as well as our proposed system, *AstFlow*. For each scenario and tool network traffic was captured to analyze the average volume and frequency of data transmissions.

We analyzed the communication patterns of these tools and compared them to our own AST-aware propagation approach, which propagates changes only at syntactically valid points. Our analysis shows that these tools generate excessive and noisy traffic and often leave collaborators in buildbreaking program states. In contrast, our approach significantly reduces amount and size of messages needed. Additionally, it improves consistency by enforcing syntactic completeness as a propagation boundary.

In summary, this paper makes the following contributions:

- **Identification of current limitations:** We identify and analyze the propagation inefficiencies of popular industry RCP tools, namely *VS Code Live Share*, *Code With Me* and *Replit* in terms of communication granularity and architecture.
- **Novel semantic-aware propagation model:** We introduce an AST-aware propagation model that reduces unnecessary updates and preserves syntactic validity.
- **RCP propagation benchmarks:** We evaluate our approach against existing tools using a benchmark of eight programming scenarios, showing significant reduction in communication overhead while maintaining consistent, buildable program states.

The paper is structured as follows: We begin with a review of related work in Section II, categorizing prior efforts into professional tools, research prototypes, and AST-based systems. Section III derives the unique requirements of RCP by comparing it with asynchronous collaboration and illustrates the challenges of fine-grained updated propagation. Section IV presents the core idea and potentials of our approach and introduces the explicit research questions this paper aims to answer. Following we present our plugin that is implemented based on the proposed model in Section V, covering static structure and dynamic execution flow. Section VI outlines our experimental evaluation, including the programming scenarios and metrics used for evaluation. Section VII presents the results of the evaluation and a comprehensive discussion. Finally, we conclude in Section VIII and outline directions for future work in Section IX.

II. Related Work

Real-time collaborative programming systems can be broadly categorized into industry tools and academic research prototypes.

Popular professional tools like Code with me [9], VS Code Live Share [10] and Replit [11] adopt a centralized host-participator pattern where the developer that initiates the coding session shares her host machine and environment with other collaborators. [22] This architecture introduces several limitations. First, it creates a single point of failure: if the host

disconnects, participators lose their unsaved work [23]. Second, participators often suffer from limited language tooling since their local environment lacks the full project context. Files are only fetched on demand and reside in memory, preventing effective cross-file analysis and refactoring. Finally, execution and debugging capabilities are restricted to the host machine, meaning other collaborators cannot run or debug the code in parallel, which severely limits the effectiveness of collaborative testing, validation, and troubleshooting [22]. In addition, these tools rely on text-level propagation, where every keystroke is transmitted immediately.

Several research prototypes aim to address architectural limitations by enabling replicated environments, where each collaborator holds a full copy of the project. Examples include: CoVSCode, a VS Code plugin [22]; CoIDEA, an IntelliJ plugin [28]; CoEclipse [25]; and CRTC for the Eclipse IDE [24]. Other tools include Teletype for the Atom editor [29]; however, Atom IDE and therefore Teletype are no longer actively maintained. Despite their architectural improvements, all of these tools propagate text-level changes without structural awareness. This introduces the same issues found in professional tools. Thus, while research prototypes improve decentralization, they do not address the fundamental problems of propagation granularity and semantic validity.

A relevant effort in solving text-based synchronization issues is CoAST [18]. Their approach introduces a CRDT that is based on the abstractions of ASTs. It demonstrates improved convergence behavior and structural consistency compared to string-based systems. However, CoAST operates in a simplified Lisp-like language and focuses primarily on merge correctness and collaborative consistency, but does not address propagation latency, IDE integration, or communication overhead. It also lacks support for fine-grained differencing strategies, which can drastically reduce the volume of propagated changes. Finally, although they present a very promising approach to RCP, contact with the authors has revealed that no further work is being pursued. *AstFlow* builds upon this foundation to deliver a more robust and practical solution. Unlike CoAST our approach introduces communication optimizations specifically designed to address one of the most pressing limitations of current RCP tools: lag under unstable network conditions [1].

III. Motivating Example

Asynchronous code collaboration enables developers to access a shared codebase, work on different components independently, and later merge their modifications into a central repository. Frequent interaction and communication between developers is not necessary and programming tasks are typically not strongly dependent, while coordination is controlled manually by developers via explicit *commits* [28]. In contrast, RCP supports closely coupled, short-lived development tasks involving high-frequency interactions among a small group of developers. Rather than replacing traditional workflows, RCP complements them by addressing collaboration scenarios where rapid feedback and coordination are critical [1], [7], [22], [23], [28].

Both paradigms share two fundamental principles: *branching* and *merging*–but they differ in **when** and **how** these operations occur.

Understanding the unique synchronization needs of RCP requires examining the evolution of merge strategies in asynchronous collaboration. Asynchronous systemes propagate changes based on explicit *commits (and pushes)* and *pull*(s). Typically such commits take place on a daily basis or even longer periods of time and therefore contain multiple and bulkier change deltas compared to RCP [28]. Nevertheless, conflict detection and resolution mechanisms are crucial in both approaches. In Git, the process of merging begins with a *diffing algorithm* that identifies variations between files, including additions, deletions, and modifications. This algorithm, operating on a per-file basis, generates a structured *diff*. It then attempts to merge these changes automatically. However, when conflicting modifications are detected (*conflict detection*) *conflict resolution* is necessary. This resolution process may involve manual intervention by developers or automated tools to integrate competing modifications [30]. In RCP, these steps must occur at a much higher frequency (e.g. per keystroke). To ensure an uninterrupted coding workflow, changes must be propagated and synchronized automatically across all collaborators [2]. However, ensuring seamless synchronization remains a non-trivial challenge–even in asynchronous collaboration–which is why it has been the focus of extensive research [26], [27], [31]–[38].

The unstructured approach of Git fails to detect complex or subtle conflicts, like semantic inconsistencies and behavioral changes [39], [40]. Line-based diffing only accounts additions and deletions, ignoring other types of edit actions such as updates and moves. This coarse-grained granularity fails to align with the structure of code [31].

To overcome these limitations, research has explored alternative strategies that operate at finer levels of granularity. Structured merge tools, for instance, analyze code using its abstract syntax tree to identify and resolve conflicts in a more semantically aware manner [31]–[37]. While these tools are highly accurate, they tend to be computationally expensive for large change sets. [26] As a compromise, semistructured merge tools have been proposed, which aim to balance performance and precision by combining partial AST analysis with traditional text-based techniques [38]. Although generally less precise than fully structured approaches, [41], [42] some recent semistructured methods show promising potential in addressing these challenges [26], [27].

These limitations are further exacerbated in real-time collaborative programming environments. Due to the nature of RCP to face continuous changes, merging is no longer an occasional operation, but a constant background process that must be fast, accurate, and non-disruptive. Any failure in detecting or resolving conflicts in real time can immediately break the shared program state, interrupt the development flow, or even introduce subtle bugs that go unnoticed until later [8].

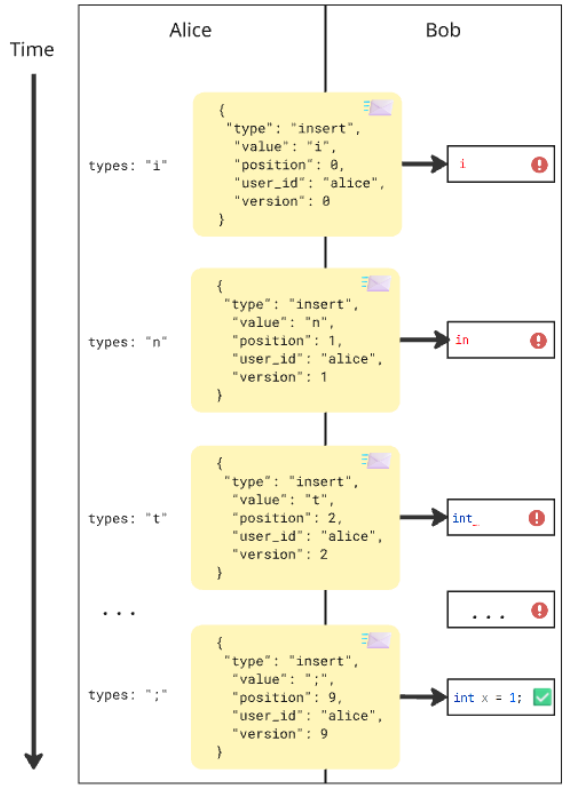

Fig. 1. Simple illustration of change propagation in text-based systems. Alice typing *'int x = 1;'*.

Despite these demands, most RCP systems rely on text-based synchronization using Operational Transformation (OT) or Conflict-free Replicated Data Types (CRDTs) [14], [21], [43]. As a result, RCP tools inherit many of the same shortcomings as traditional text-based version control systems. As developers write code, it is natural for the code to temporarily transition in syntactically invalid states–such as an unclosed bracket, an incomplete statement, or a partially written function signature. Intermediate code states are propagated immediately to all participants, frequently resulting in shared build-breaking states. This issue is presented in Figure 1: as Alice types `int x = 1;`, Bob experiences build-breaking intermediate states due to character-wise propagation. The constant shift between valid and invalid code not only undermines individual focus but also introduces confusion and interference, making fine-grained synchronization a double-edged sword [8], [21].

IV. AST-based Propagation Model

To highlight the presented research gap, we refer to a recent survey [1] by Tan et al. (2024) among 103 developers. They analyzed common usage scenarios, key requirements, and persistent challenges in RCP environments. Unsurprisingly, live editing—such as real-time change tracking—was identified as the most important feature, with 87% of respondents ranking it as essential. The most frequently reported challenge was lag, attributed primarily to unstable (71%) and slow (63%) network conditions. Additional concerns included editing conflicts (57%), difficulty in undoing others' changes (61%), and loss of work after interruptions (66%). Based on these findings, we hypothesize that propagating changes at the level of individual keystrokes significantly contributes to the perceived lag. Additionally, unnecessary data transmission of invalid code states not only burdens the network but also ruins developer experience because developers are continuously interrupted by invalid states propagated by others.

To address these challenges, we propose a propagation model that strategically rethinks how and when updates are shared. Our approach focuses on minimizing latency and preserving syntactical correctness even in degraded network states. At the heart of this model lies a shift in granularity: instead of broadcasting character or text updates, our approach transmits only code changes represented as AST operations. This structural awareness enables the system to filter out transient, invalid states. The edit operations are only propagated if the underlying program is parsable. Therefore, invalid states that would otherwise disrupt other collaborators and overwhelm the network with redundant updates, are skipped. The key idea of the dynamic process flow is illustrated in Figure 2. When developers start an RCP session, an initial AST is created, based on the program structure. During the session the system listens for change events, which are triggered when the developers write code. While coding, we leverage incremental parsing techniques to validate whether the program is parsable or not. When the program reaches a parsable state we compare the last valid AST (T_1) with the current AST (T_2), with the aim to extract operations made since then using a diff algorithm. Basically, we extract the sequence of actions, which transform T_1 into T_2. The outcome of this comparison is called a *minimal edit-script*, which is eventually broadcasted to other collaborators.

By aligning propagation boundaries with meaningful syntactic units–such as complete statements, declarations, or expressions–we aim to preserve the integrity of the shared program state and reduce the cognitive load on developers.

We hypothesize that this model not only improves syntactic correctness but also significantly reduces communication overhead–one of the most frequently cited pain points in RCP systems [1]. To validate this hypothesis, we designed a set of controlled experiments that simulate realistic coding scenarios and measure the resulting network traffic across multiple tools. The experimental evaluation is presented in Section VI and results are demonstrated and discussed in Section VII

With our approach we aim to answer the following research questions:

- RQ1: How does AST-based propagation affect the frequency and volume of transmitted updates in real-time collaborative programming compared to traditional text-based systems?
- RQ2: How do different types of code edits impact network traffic across AST-based and text-based RCP tools?

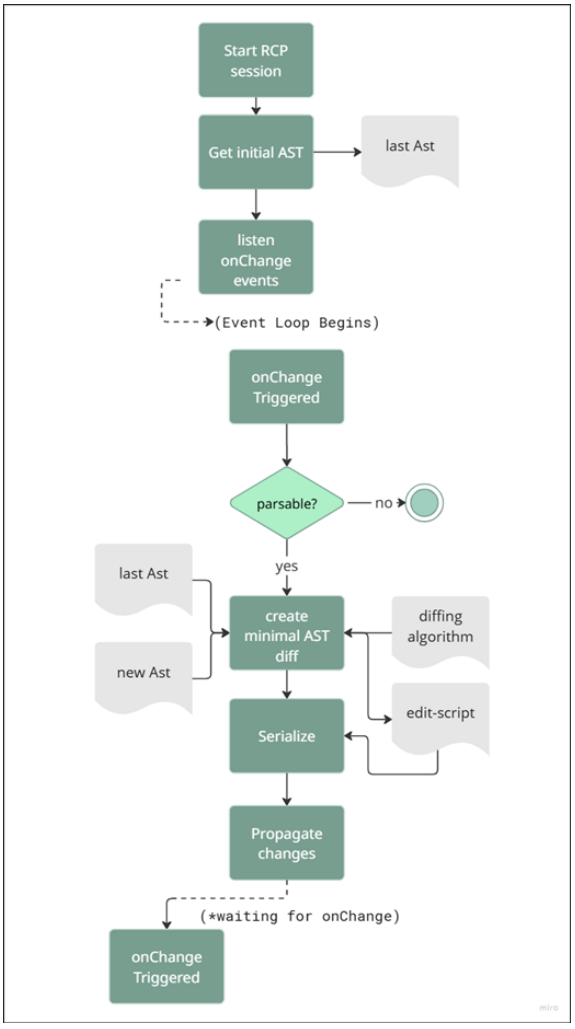

Fig. 2. Flow of propagating code changes as AST-operations in RCP

V. Implementation

AstFlow is implemented as a plugin for the IntelliJ platform using Kotlin. This design decision was motivated by IntelliJ's built-in support for parsing source code and its developer-friendly infrastructure for rapid plugin development and testing. Central to AstFlow's current implementation is the Project Structure Interface (PSI) [44] provided by the IntelliJ platform, which offers rich semantics and a comprehensive API for navigating and manipulating the underlying syntax tree of source files [44]. PSI provides convenient hooks for listening to changes in the AST via interfaces such as *PsiTreeChangeAdapter*, enabling AstFlow to react instantaneously to structural modifications within the code. Currently, the system is limited to supporting Java and Kotlin, but this constraint aligns with the prototype's goal: to demonstrate the potentials of a structure-based propagation model in collaborative code editing, which is further explored in Sections VI and VII.

```
===
insert-tree
---
VariableDeclarationStatement [119,129]
    PrimitiveType: int [119,122]
    VariableDeclarationFragment [123,128]
        SimpleName: x [123,124]
        NumberLiteral: 1 [127,128]
to
Block [109,125]
at 0
```

Fig. 3. Gumtree pretty print of edit-script for Alice coding *int x = 1;*.

A collaborative session in AstFlow begins when a user initiates a session through a lightweight relay server that manages WebSocket connections and assigns unique user identifiers. Upon session initiation, the current file's AST is captured and monitored for changes. As the user interacts with the code, PSI triggers a *childrenChanged* callback. At this point, AstFlow first checks whether the current file remains parsable, using PSI's capabilities. If the file contains syntax errors and is unparseable, no data is transmitted. If it is valid, AstFlow computes the structural difference between the previous and current ASTs using the Gumtree algorithm [31]. Gumtree is a highly efficient algorithm designed to determine the sequence of operations that, when executed, convert an abstract syntax tree T_1 into T_2. Figure 3 shows a formatted (pretty-printed) version of an edit-script generated by Alice in response to the addition of *int x = 1;* to the codebase. AstFlow leverages the recent *gumtree-simple* heuristic [45], which significantly improves performance and produces cleaner edit-scripts, being up to 281× faster than earlier versions [45]. Possible actions in an edit-script created by *gumtree-simple* are [45]:

- insert-node(*n, p, i*): adds a new node *n* as a child of node *p*, inserted at position *i*.
- insert-tree(*s, p, i*): inserts a new subtree *s* as a child of node *p* at position *i*.
- delete-node(*n*): removes the leaf node *n*.
- delete-tree(*s*): removes the subtree *s*.
- move-tree(*s, p, i*): moves the subtree *s* to become a child of node *p*, positioned at *i*.
- update-node(*n, l*): updates the label of node *n* to *l*.

The edit-script is mapped to a custom data transfer object (DTO) presented in figure 4. While transmitting full subtrees in some operations may introduce unnecessary overhead, little optimizations have yet been applied in this prototype. These improvements are planned for future work, where we aim to balance the data required for merge accuracy with minimal traffic overhead. Once serialized, the operations are sent via WebSocket to the relay server, which broadcasts the changes to all connected participants.

The relay server as well as the plugin source code and programming scenarios (captures, logs) are provided via GitHub

```
{
  "userId": "81db7013-9c83-4914-b5fa-521bc2373d78",
  "filePath": "AutoScript.java",
  "editScript": [
    {
      "type": "TREEINSERT",
      "nodeUrl": "0.1.3.5.0",
      "parentUrl": "0.1.3.5",
      "subtree": {
        "type": "VariableDeclarationStatement",
        "value": "",
        "children": [
          {
            "type": "PrimitiveType",
            "value": "int"
          },
          {
            "type": "VariableDeclarationFragment",
            "value": "",
            "children": [
              {
                "type": "SimpleName",
                "value": "x"
              },
              {
                "type": "NumberLiteral",
                "value": "1"
              }
            ]
          }
        ]
      }
    }
  ]
}
```


Fig. 4. editOperationDTO for action from edit-script presented in fig. 3.

[46] and Figshare [47].

VI. Experimental Evaluation

To assess the performance and communication efficiency of our proposed propagation model, we conducted a controlled experimental evaluation. The goal of this evaluation is twofold: (1) to understand how different propagation strategies affect the frequency and volume of transmitted data (RQ1), and (2) to analyze whether and how industry tools implement optimizations to mitigate communication overhead under various collaborative editing scenarios (RQ2).

We designed a set of eight representative programming scenarios that reflect common editing patterns in RCP, such as regular coding, using autocompletion, typing bursts, deletions, and copy-paste operations. Each scenario is crafted to trigger specific synchronization behaviors and stress-test the underlying propagation mechanisms. For instance, we investigate whether text-based systems transmit each character individually–even when an entire code block is pasted at once–or whether they apply throttling or batching when code is generated rapidly. The full list of scenarios and their intended evaluation goals are described in Section VI-A.

These scenarios were executed across four RCP systems: our own plugin, *AstFlow*, and three widely adopted industry tools–VS Code Live Share, Code With Me, and Replit. These tools were selected based on their popularity and widespread usage: VS Code Live Share (over 20M downloads [10]), Code With Me (over 600K downloads [9]), and Replit (over 22.5M users [48]).

To ensure consistency and reproducibility, all experiments were conducted in a controlled environment. Each scenario was executed three times per tool leveraging an automated script to simulate user input. Furthermore, the script captures network traffic during each scenario. The network traffic was recorded using Wireshark and Mitmproxy, allowing us to isolate and analyze WebSocket communication. We measured two key metrics: (1) the number of packets transmitted, and (2) the total volume of data exchanged (in bytes). These metrics provide insights into the communication behavior introduced by each tool's propagation strategy.

By comparing the results across tools and scenarios, we aim to quantify the impact of propagation granularity on network efficiency and identify patterns that correlate with lag and responsiveness issues reported in RCP literature. All scripts, scenario definitions, and captured data can be found in "/rcp-evaluation" in the repositories on GitHub [46] and Figshare [47].

A. Programming Scenarios

To investigate how different RCP tools propagate changes during coding sessions, we designed a set of eight targeted programming scenarios. These scenarios are crafted to systematically explore the relationship between types of code edits and the resulting network traffic, with the goal of answering both our research questions (presented in Section IV). Each scenario isolates specific editing patterns–ranging from simple typing to complex insertions or deletions–to uncover the underlying propagation strategies used by the tools. By observing how tools handle structurally meaningful versus purely textual edits, we aim to reveal whether they implement optimizations such as throttling, batching, or structural reasoning. Given the closed-source nature of the tools, direct access to their internal propagation logic is not possible. Therefore, we rely on empirical observation and network traffic analysis to infer their behavior. By capturing and analyzing the exchanged data during each scenario, we can assess how tools respond to different types of edits and gain insights into the granularity, structure, and timing of propagated changes.

Unless otherwise specified, a 100 ms delay between keystrokes is used to simulate average typing behavior. In cases simulating rapid actions–such as copy-paste operations or AI-assisted code insertions–delays are intentionally removed to reflect real-world usage patterns. This controlled setup allows us to reason not only about what strategies each tool employs, but also about how specific code editing actions trigger particular propagation behaviors.

The scenarios are defined as follows:

- *Basic Typing*: Creation of a class with a main method and a simple *System.out.println("Hello, world!")* statement. It introduces invalid intermediate states, allowing us to observe how tools handle propagation under syntactic instability. This scenario represents a typical editing task at a standard typing pace, serving as a baseline for comparison.

- *Copy-Paste Function*: Pasting a complete function into the class. This scenario examines whether tools detect and optimize for bulk insertions, or whether they treat them as a sequence of individual character-level changes.
- *Deleting Line Letter by Letter*: Character-by-character deletion of a statement, resulting in temporary syntax errors.
- *Deletion*: Removal of a previously typed line (e.g., the println statement) in a single action, simulating IDE macros or block deletion. This scenario complements insertion-based tests and helps evaluate how deletion events are propagated.
- *Inline Comment*: Insertion of an inline comment without keystroke delays. This simulates AI-assisted comment generation (e.g., via GitHub Copilot). Since AST-based systems typically ignore comments, this scenario helps assess whether industry tools incorporate structural awareness or rely solely on text-based propagation.
- *Insert Line Comment*: Addition of a full-line comment at default typing speed. Similar in purpose to the inline comment scenario, but without AI involvement, this case helps isolate the tool's treatment of non-functional code edits.
- *Typing String with Quotes*: Editing a string literal while maintaining syntactic validity throughout. This scenario triggers a potential drawback of AST-based systems: since the file remains parsable, every keystroke may trigger a structural diff.
- *Typing Burst*: Rapid entry of multiple lines without delay, simulating IDE autocomplete or fast typing. This scenario is designed to test whether tools implement throttling or batching mechanisms to reduce communication overhead during high-frequency input.

B. Experimental Setup

All scenarios were conducted in a controlled environment using Python (version 3.12.6). The script comprises the following modules:

- Definition and automation of programming scenarios (Section VI-A) via the keyboard library [49] to simulate keystroke events.
- Scenario execution with precise logging of start times.
- Network traffic capture per scenario using Wireshark.

To ensure controlled conditions, smart editor features such as auto-closing and paired brackets were disabled to facilitate accurate simulation. All tools under test communicated via WebSocket connections, as verified through Mitmproxy [50] traffic analysis.

For each tool, a session between two users was established in advance. To focus on measuring the frequency of update propagation (i.e. how "chatty" the client is) and the amount of data transmitted, all programming scenarios are simulated on a single user's machine. For instance, Alice initiates a Code With Me session and shares the invitation link with Bob. The scripted scenarios are then executed on Alice's behalf.

We intercepted network traffic using Mitmproxy to identify and filter WebSocket IP addresses and destination ports, ensuring analysis focused exclusively on relevant real-time data streams. Since WebSocket traffic is encrypted over TLS and not fully interpretable by Wireshark, we analyzed the raw TCP packets corresponding to WebSocket connections. TCP packet filtering was necessary because a WebSocket connection is established once, and all subsequent data is exchanged over it. Mapping TCP packets to scenario timestamps enabled detailed traffic analysis. Due to encryption constraints, only Replit's traffic contents could be decrypted and examined. Application data from Code With Me and VS Code Live Share remained opaque.

The traffic filtering process involved isolating TCP packets originating from the client and excluding TCP ACKs by selecting only those with a payload length greater than zero. However, it was not possible to filter out ping/pong messages exchanged via WebSockets. Since all evaluated tools rely on WebSocket communication, these control messages are uniformly present across all sessions. Moreover, because measurements are averaged over multiple rounds for each scenario, any overhead introduced by ping/pong exchanges remains consistent and negligible relative to the variation caused by actual code edit propagation. As a result, their presence does not bias the comparative analysis or impact the validity of the observed propagation behavior.

For each collaboration tool, scenarios were executed three times. Using the logged timestamps, network captures were correlated with scenario execution to extract:

- Average number of packets sent per scenario,
- Average total data transmitted per scenario (in bytes).

VII. Results and Discussion

The network traffic analysis reveals fundamental differences in how state-of-the-art RCP tools and our proposed system, AstFlow, manage change propagation. The results of the evaluation are presented in Table I, Figures 5 and 6.

A. Addressing High Communication Overhead in RCP Tools

Consistent with prior findings, industry RCP tools suffer from high communication overhead due to propagation of every keystroke, including incomplete and syntactically invalid code states [1], [23]. Our evaluation confirms that both VS Code Live Share (VSC) and Replit exhibit extremely high packet counts and data volumes.

For instance, in the *basic typing* scenario, VSC sends on average 951 packets compared to only 2.3 packets by AstFlow, a difference of approximately 413-fold. Similarly, Replit sends around 771 packets, about 335 times more than AstFlow. This excessive "chattiness" not only increases bandwidth consumption but also risks lag and unstable performance during collaborative editing sessions.

Our results empirically validate that conventional text-based synchronization approaches propagate transient, build-breaking states, which in turn degrade the developer experience. These propagation patterns align with user-reported

TABLE I
COMPARISON OF COLLABORATION TOOLS ACROSS EDITING SCENARIOS (PACKETS AND DATA USAGE)

Scenario	AstFlow		Code with Me		VS Code Live Share		Replit	
	$\bar{x}_{\text{packets}}$	$\bar{x}_{\text{KB}}$	$\bar{x}_{\text{packets}}$	$\bar{x}_{\text{KB}}$	$\bar{x}_{\text{packets}}$	$\bar{x}_{\text{KB}}$	$\bar{x}_{\text{packets}}$	$\bar{x}_{\text{KB}}$
basic_typing	2.3	1.4	41.3	4.7	951.3	551.3	771.0	227.8
copy_paste_function	1.3	1.0	5.3	0.4	287.3	188.9	163.0	76.4
deleting_line_letter_by_letter	7.0	2.1	28.0	2.2	523.7	281.1	512.3	156.7
deletion	1.0	0.2	6.0	0.5	27.7	13.1	30.7	10.0
inline_comment	1.0	0.1	4.7	0.4	114.0	58.7	59.0	23.3
insert_line_comment	-	-	14.0	1.1	184.3	97.5	212.3	53.3
type_string_with_quotes	25.3	6.4	16.7	1.3	545.7	279.5	270.0	78.0
typing_burst	1.3	1.5	4.0	0.3	339.3	212.0	166.7	80.7

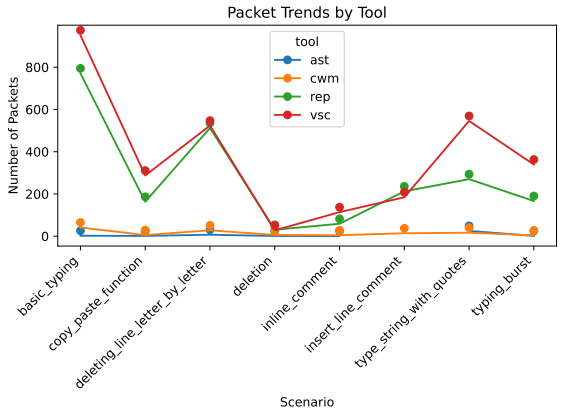

Fig. 5. Number of packets as trend per scenario and tool.

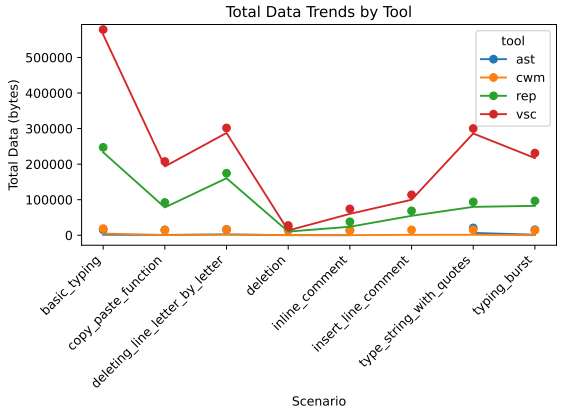

Fig. 6. Data (Bytes) as trend per scenario and tool.

frustrations in recent RCP surveys, where lag and workflow interruptions were cited as major pain points [1].

Interestingly, Code With Me (CWM) consistently transmits more packets than AstFlow across all scenarios, yet still significantly fewer than VSC and Replit. This suggests that while CWM likely uses a text-based propagation model, it does not emit operations on every keystroke. Instead, its packet pattern points to the use of throttling or batching strategies. This raises the question of how it captures fine-grained edits. A plausible explanation is that CWM relies on batch transactions (e.g., groups of atomic operations) that are generated by integrating directly with the host editor's change model. Given the encrypted nature of CWM's WebSocket traffic, we were unable to directly analyze its synchronization protocol. Our interpretations are therefore drawn from observable behavior patterns and network analysis.

A closer look at the packet trends across scenarios reveals further insights. As expected, the *basic typing* scenario resulted in the highest packet counts for all text-based tools, since it comprised the most edit operations (character insertions). Except for AstFlow, since most code edits introduced syntactically invalid code states. In contrast, the *deletion* scenario resulted in relatively few packets across all tools, suggesting that deletion operations are efficiently tracked. For example, as a range-based action rather than character-wise propagation.

The *deleting line letter by letter* scenario, however, shows a sharp increase in packet count for VSC and Replit, while CWM appears to apply some optimization, transmitting fewer packets than expected. As anticipated, the *insert line comment* and *inline comment* scenarios did not trigger AstFlow's propagation mechanism, since comments do not affect the AST. The single packet observed in these cases is likely a WebSocket ping/pong message.

One particularly revealing case is the *type string with quotes* scenario. Although AstFlow propagates updates on every keystroke due to the file remaining syntactically valid, it still transmits significantly fewer packets than VSC and Replit–by a factor of 10 to 20. Interestingly, CWM sends fewer packets than the number of keystrokes in this scenario, further supporting the assumption that it employs throttling or batching strategies.

The overall data volume trends (Fig. 6) closely follow the packet count patterns. However, despite transmitting a higher number of packets, Replit consistently uses smaller packet sizes compared to VSC. For instance, in the *insert line comment* scenario, Replit sends more packets than VSC (339.3 vs. 270), yet its total data volume is significantly lower–80.7 KB compared to VSC's 212.0 KB. This represents a 61.9%

reduction in data usage, or roughly a 2.6× decrease in volume, suggesting more compact and efficient message encoding.

While both AstFlow and CWM show relatively low network usage compared to other evaluated tools, a closer inspection reveals key differences in their underlying synchronization strategies (Figures 7 and 8). CWM, consistently transmits more packets than AstFlow–except for *type string with quotes* scenario–but less data in scenarios *copy paste function, typing burst* and *type string with quotes scenario*. This efficiency likely stems from optimization strategies such as packet batching, payload compression, or intelligent throttling, as previously hypothesized.

The structure-aware propagation approach of AstFlow potentially enables higher-level reasoning about code edits. However, in scenarios where code remains syntactically valid throughout rapid keystroke sequences–such as *type string with quotes* scenario–AstFlow exhibits behavior that closely resembles text-based propagation. Specifically, it transmits more packets (25.3 vs. 16.7) and more data (6.4 KB vs. 1.3 KB) than CWM in that scenario.

Conversely, in bulk-edit scenarios such as *copy paste function* and *typing burst*, AstFlow transmits more bytes (1.0 KB and 1.5 KB respectively) than CWM (0.4 KB and 0.3 KB respectively) but less packets (1.3 and 1.3 vs. 5.3 and 4.0). This inversion arises from the structural nature of the edits: AstFlow emits one or more tree insert operations–each with inherent metadata and structural context–whereas CWM likely transmits compact, batched text data, possibly further reduced through compression.

It is important to contextualize these findings within the maturity of the tools. AstFlow is a rapid prototype, developed in a limited timeframe, whereas CWM is a production-grade, professionally engineered tool. Despite this, AstFlow's structure-aware model demonstrates promising potential. The observed overhead in certain scenarios could be addressed through targeted optimizations such as edit aggregation, diff throttling, or data compression. With such enhancements, AST-based propagation could achieve a level of efficiency surpassing, text-based systems.

B. Semantic Granularity and Buildable States Reduce Traffic

AstFlow's structure-aware approach directly addresses these challenges by only transmitting changes when the source code reaches syntactically valid states [51]. This semantic granularity reduces the frequency of updates transmitted, as incomplete or invalid intermediate states are never broadcast. In almost all scenarios it also reduces the size of data. This mechanism is particularly advantageous for scenarios involving complex edits where code often transiently becomes invalid, such as partially typed statements or unclosed brackets. All other RCP tools propagated invalid states.

However, one notable exception to this pattern is the *type string with quotes* scenario. Here, AstFlow propagates nearly every keystroke, much like a text-based tool. This behavior arises because string literals remain syntactically valid as they are being typed, allowing the parser to produce valid

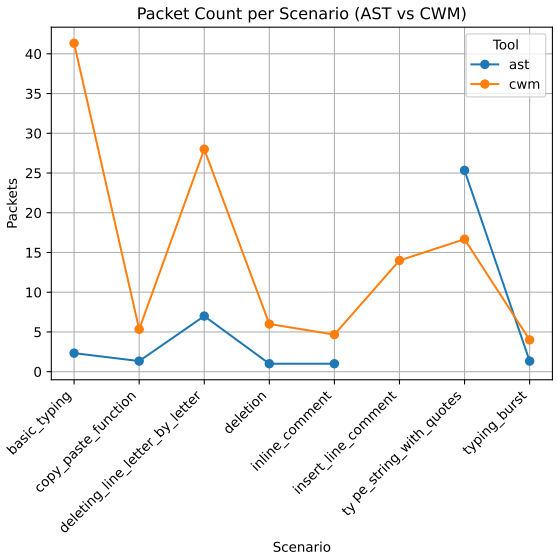

Fig. 7. Direct comparison of packets: AstFlow vs Code With Me.

ASTs at each step. This deviation highlights a key area for improvement: current semantic granularity alone is insufficient when frequent but trivial changes still yield buildable states. Future optimizations might incorporate additional heuristics. For example, incorporating cursor movement, edit intent, or syntactic scope awareness, could delay propagation until meaningful semantic units are formed.

These findings reinforce the potential of structure-aware synchronization models. Even as a rapid prototype, AstFlow demonstrates that AST-level granularity can dramatically reduce unnecessary network load–particularly in scenarios involving invalid intermediate edits–while opening the door to more intelligent and context-aware propagation strategies.

C. Threats to Validity

Several threats to validity should be considered when interpreting the results of this paper. First, the benchmark scenarios, while carefully designed to be representative, do not fully capture the complexity and variety of more complex programming tasks, such as large-scale refactoring, multi-file editing, or debugging workflows. This limits the external validity and generalization of the findings. Additionally, the experiments were conducted under controlled network conditions, which may differ significantly from the diverse and often unpredictable environments encountered in practical use, potentially affecting traffic patterns and tool behavior. A further limitation arises from the encrypted nature of the network traffic in VS Live Share and Code with me, which prevents inspection of the message payloads and thus constrains our ability to understand the detailed communication protocols and optimization strategies employed by the commercial tools.

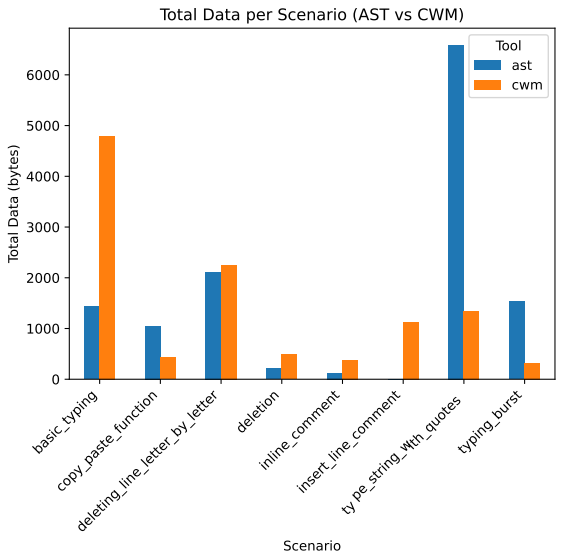

Fig. 8. Direct comparison of data: AstFlow vs Code With Me.

VIII. Conclusion

This paper introduces a structure-aware propagation model for real-time collaborative programming (RCP), addressing the inefficiencies of keystroke-level synchronization in widely-used text-based systems. By leveraging abstract syntax trees (ASTs), our approach transmits only syntactically valid code changes, reducing network traffic and improving collaboration stability.

We addressed the following research question:

RQ1: How does AST-based propagation affect the frequency and volume of transmitted updates in real-time collaborative programming compared to traditional text-based systems?

To evaluate this, we implemented our model as an IntelliJ plugin, and compared it to three widely used RCP tools–VS Code Live Share, Code With Me, and Replit–which rely on text-based propagation. Our tool demonstrated substantial reductions in both the number of updates and the volume of data transmitted, by avoiding the broadcast of intermediate, non-buildable states.

To deepen this analysis, we formulated a secondary question:

RQ2: How do different types of code edits impact network traffic across AST-based and text-based RCP tools?

To answer this question we designed eight representative coding scenarios and simulated diverse real-world edits. These scenarios enabled us to analyze propagation behavior in detail and reason about each tool's propagation strategy. Our results revealed that while text-based tools transmit edits continuously–even during syntactically invalid states–our tool suppresses such noise, significantly lowering traffic. However, in scenarios such as typing string literals–where partial edits remain syntactically valid–AstFlow propagated more frequently, exhibiting a pattern more similar to text-based systems. While it still transmitted significantly fewer updates than VS Code Live Share (by a factor 22×) and Replit (11×), it propagated about 1.5× more frequently than Code With Me. This reveals an opportunity for further optimization, such as integrating semantic context or user intent to suppress trivial but syntactically valid changes.

Despite being a research prototype, our tool's performance in both packet count and data volume matched or surpassed that of professional RCP tools. Although it incurs higher computational cost due to AST diffing, this trade-off is balanced by its structural awareness and the reduced need for downstream conflict resolution.

In summary, our findings highlight the advantages of propagating code at a semantically meaningful granularity. AST-based propagation mitigates many of the core challenges faced by current RCP systems–including bandwidth waste, unstable shared states, and limited semantic context. This work lays the groundwork for more robust, scalable, and developer-friendly collaboration models in modern software development.

IX. Future Work

Future work will focus on extending our evaluation to more complex programming scenarios, including cross-file refactorings, function relocations, and structural code reorganizations, to better understand how tools handle semantically rich changes. A key direction will be to compare structure-based and text-based synchronization approaches in these scenarios, analyzing their respective limitations and advantages. This comparative analysis aims to highlight when structure-awareness provides tangible benefits over text-level tracking and where it may introduce overhead or complexity. AstFlow currently focuses on defining a propagation model that ensures the transmission of syntactically valid program states. We intentionally scoped synchronization strategies, such as handling concurrent edits or multi-user conflicts, out of this initial work to first establish the benefits of structure-aware propagation. Our evaluation demonstrates the potential of this model compared to existing tools. Building on this foundation, we plan to explore transactional propagation and structural conflict resolution mechanisms, with a strong emphasis on developer experience. For instance, while fully automated conflict resolution (as in Conflict-free Replicated Data Types or Operational Transformation) and fully manual approaches (as in Git) each have their merits, neither may offer an ideal solution in real-time collaborative programming. We aim to investigate, through user studies, what balance between automation and manual control best supports developer workflows.

The computational overhead of AST diffing, especially for large files or complex refactoring, is an important concern in real-time settings. GumTree was selected for its widespread adoption and proven effectiveness in generating concise edit scripts, which are essential for RCP. Its use in tools that operate across entire repositories further supports its scalability. Recent studies [45], [52] confirm its superior performance in producing minimal and meaningful change sets. Nonetheless,

we recognize the limitations of full-file parsing and plan to investigate incremental parsing techniques (such as those enabled by Tree-Sitter [53]) as part of our future work. We intend to benchmark both approaches on large open-source codebases to assess their impact on performance perceived by developers.

While AstFlow currently targets Java and Kotlin, its structure-aware core model for propagation is inherently language-agnostic, and we see significant potential in leveraging frameworks such as Tree-Sitter, which already offers incremental parsing for a wide range of languages, to extend support beyond the JVM ecosystem while improving performance on larger files.

Given that code is often in an incomplete state during development, we also propose the addition of an intent-aware propagation layer. Detecting user intent in collaborative editing is a non-trivial challenge, but one we believe can be effectively integrated into AstFlow. Our future work will explore mechanisms for manual propagation (e.g., user-triggered updates via shortcuts like Ctrl+S) and automatic detection of complete edits (e.g., closing brackets or statements), potentially supported by incremental parsing. This could delay synchronization until a statement is likely complete, thus reducing noise from transient edits.

Additionally, we will explore optimizations in message transmission, including data minimization and compression techniques, to further enhance network efficiency.

Overall, these directions aim to evolve our approach into a more semantically aware and performant framework for real-time collaborative programming, pushing the boundaries of RCP systems in both usability and technical sophistication.

REFERENCES

[1] X. Tan, X. Lv, J. Jiang, and L. Zhang, "Understanding real-time collaborative programming: a study of visual studio live share," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 4, pp. 1–28, 2024.

[2] H. Fan, *Any-time collaborative programming environment and supporting techniques*. PhD thesis, Nanyang Technological University, Singapore, 2013.

[3] K. Constantino, S. Zhou, M. Souza, E. Figueiredo, and C. Kästner, "Understanding collaborative software development: An interview study," in *Proceedings of the 15th international conference on global software engineering*, pp. 55–65, 2020.

[4] J. Whitehead, "Collaboration in software engineering: A roadmap," in *Future of Software Engineering (FOSE'07)*, pp. 214–225, IEEE, 2007.

[5] Y. Ma, B. Qi, W. Xu, M. Wang, B. Du, and H. Fan, "Integrating real-time and non-real-time collaborative programming: Workflow, techniques, and prototypes," *Proceedings of the ACM on Human-computer Interaction*, vol. 7, no. GROUP, pp. 1–19, 2023.

[6] D. Sun, F. Ouyang, Y. Li, and H. Chen, "Three contrasting pairs' collaborative programming processes in china's secondary education," *Journal of Educational Computing Research*, vol. 59, no. 4, pp. 740–762, 2021.

[7] D. Sun and F. Xu, "Real-time collaborative programming in undergraduate education: A comprehensive empirical analysis of its impact on knowledge, behaviors, and attitudes," *Journal of Educational Computing Research*, vol. 63, no. 1, pp. 33–63, 2025.

[8] L. Hattori and M. Lanza, "An environment for synchronous software development," in *2009 31st International Conference on Software Engineering-Companion Volume*, pp. 223–226, IEEE, 2009.

[9] JetBrains, "Code with me," 2025. A tool for collaborative development that lets you share your IDE with others in real time.

[10] Microsoft, "Visual studio live share," 2025.

[11] Replit, "Replit - The collaborative browser-based IDE." https://replit.com/, 2025. Accessed: 2025-05-22.

[12] M. Day, "What synchronous groupware needs: Notification services," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*, pp. 118–122, IEEE, 1997.

[13] C. Sun and C. Ellis, "Operational transformation in real-time group editors: issues, algorithms, and achievements," in *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pp. 59–68, 1998.

[14] G. Litt, S. Lim, M. Kleppmann, and P. van Hardenberg, "Peritext: A crdt for collaborative rich text editing," vol. 6, Nov. 2022.

[15] W. Yu, "A string-wise crdt for group editing," in *Proceedings of the 2012 ACM International Conference on Supporting Group Work*, GROUP '12, (New York, NY, USA), p. 141–144, Association for Computing Machinery, 2012.

[16] W. Yu, "Supporting string-wise operations and selective undo for peer-to-peer group editing," in *Proceedings of the 2014 ACM International Conference on Supporting Group Work*, pp. 226–237, 2014.

[17] M. Weidner and M. Kleppmann, "The art of the fugue: Minimizing interleaving in collaborative text editing," *arXiv preprint arXiv:2305.00583*, 2023.

[18] A. Munsters, A. Scull Pupo, and J. Nicolay, "Coast: A conflict-free replicated abstract syntax tree," in *17th International Conference on Software Technologies*, vol. 1 of *Proceedings of the 17th International Conference on Software Technologies - ICSOFT*, pp. 187–196, Scitepress, July 2022.

[19] M. Kleppmann, D. P. Mulligan, V. B. Gomes, and A. R. Beresford, "A highly-available move operation for replicated trees," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 7, pp. 1711–1724, 2021.

[20] L. Stewen and M. Kleppmann, "Undo and redo support for replicated registers," in *Proceedings of the 11th Workshop on Principles and Practice of Consistency for Distributed Data*, pp. 1–7, 2024.

[21] D. Iovescu and C. Tudose, "Real-time document collaboration—system architecture and design," *Applied Sciences*, vol. 14, no. 18, p. 8356, 2024.

[22] H. Fan, K. Li, X. Li, T. Song, W. Zhang, Y. Shi, and B. Du, "Covscode: a novel real-time collaborative programming environment for lightweight ide," *Applied Sciences*, vol. 9, no. 21, p. 4642, 2019.

[23] H. Zhou, Y. Ma, W. Xu, M. Wang, B. Du, and H. Fan, "Context-based operation merging in real-time collaborative programming environments," in *2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pp. 1426–1431, IEEE, 2022.

[24] S. Levin and A. Yehudai, "Collaborative real time coding or how to avoid the dreaded merge," *arXiv preprint arXiv:1504.06741*, 2015.

[25] H. Fan and C. Sun, "Achieving integrated consistency maintenance and awareness in real-time collaborative programming environments: The coeclipse approach," in *Proceedings of the 2012 IEEE 16th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pp. 94–101, IEEE, 2012.

[26] G. Cavalcanti, P. Borba, L. d. Anjos, and J. Clementino, "Semistructured merge with language-specific syntactic separators," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1032–1043, 2024.

[27] J. Dong, J. Sun, Y. Lin, Y. Zhang, M. Ma, J. S. Dong, and D. Hao, "Revisiting the conflict-resolving problem from a semantic perspective," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 141–152, 2024.

[28] W. Xu, Y. Ma, H. Zhou, M. Wang, B. Du, and H. Fan, "A multiple locking group scheme for flexible semantic conflict prevention in real-time collaborative programming," in *2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pp. 1432–1437, IEEE, 2022.

[29] "Teletype for atom." https://atom-editor.cc/teletype/. Accessed: 2025-03-31.

[30] Y. S. Nugroho, H. Hata, and K. Matsumoto, "How different are different diff algorithms in git? use–histogram for code changes," *Empirical Software Engineering*, vol. 25, pp. 790–823, 2020.

[31] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, (New York, NY, USA), p. 313–324, Association for Computing Machinery, 2014.

[32] Y. Higo, A. Ohtani, and S. Kusumoto, "Generating simpler ast edit scripts by considering copy-and-paste," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 532–542, IEEE, 2017.

[33] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "Cldiff: generating concise linked code differences," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pp. 679–690, 2018.

[34] B. Fluri, M. Wursch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[35] O. Leßenich, S. Apel, C. Kästner, G. Seibt, and J. Siegmund, "Renaming and shifted code in structured merging: Looking ahead for precision and performance," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 543–553, IEEE, 2017.

[36] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Jdiff: A differencing technique and tool for object-oriented programs," *Automated Software Engineering*, vol. 14, pp. 3–36, 2007.

[37] B. Shen, W. Zhang, H. Zhao, G. Liang, Z. Jin, and Q. Wang, "Intellimerge: A refactoring-aware software merging technique," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

[38] A. T. Tavares, P. Borba, G. Cavalcanti, and S. Soares, "Semistructured merge in javascript systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1014–1025, IEEE, 2019.

[39] P. Alikhanifard and N. Tsantalis, "A novel refactoring and semantic aware abstract syntax tree differencing tool and a benchmark for evaluating the accuracy of diff tools," *ACM Trans. Softw. Eng. Methodol.*, Sept. 2024. Just Accepted.

[40] M. Ellis, S. Nadi, and D. Dig, "Operation-based refactoring-aware merging: An empirical evaluation," 2022.

[41] G. Cavalcanti, P. Borba, G. Seibt, and S. Apel, "The impact of structure on software merging: Semistructured versus structured merge," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1002–1013, IEEE, 2019.

[42] G. Seibt, F. Heck, G. Cavalcanti, P. Borba, and S. Apel, "Leveraging structure in software merge: An empirical study," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4590–4610, 2021.

[43] P. S. Almeida, A. Shoker, and C. Baquero, "Efficient state-based crdts by delta-mutation," in *International Conference on Networked Systems*, pp. 62–76, Springer, 2015.

[44] JetBrains, "Program structure interface (psi)," 2024. Accessed: 2025-05-22.

[45] J.-R. Falleri and M. Martinez, "Fine-grained, accurate and scalable source differencing," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–12, 2024.

[46] L. Freudenthaler, B. Taufner, and K. M. Göschka, "ast_flow: Intellij plugin for ast-based propagation of code changes, a small relay server and evaluation scripts." https://github.com/leonardo1710/ast$_flow$, 2025.

[47] L. Freudenthaler, B. Taufner, and K. M. Göschka, "Code repository for "from characters to structure: Rethinking real-time collaborative programming models"," 2025.

[48] N. Bello, K. Harrison, and A. Chatterjee, "Replit business breakdown & founding story," Mar. 2025. Contrary Research.

[49] Boppreh, "keyboard: Hook and simulate global keyboard events on windows and linux." https://pypi.org/project/keyboard/, 2023. Accessed: 2025-05-20.

[50] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors, "mitmproxy: A free and open source interactive HTTPS proxy," 2010–. [Version 12.0].

[51] D. Wendel and P. Medlock-Walton, "Thinking in blocks: Implications of using abstract syntax trees as the underlying program model," in *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, pp. 63–66, IEEE, 2015.

[52] M. T. Hasan, N. Tsantalis, and P. Alikhanifard, "Refactoring-aware block tracking in commit history," *IEEE Transactions on Software Engineering*, 2024.

[53] Max Brunsfeld et. al, "Tree-sitter is a parser generator tool and an incremental parsing library," 2024. Accessed: 2024-10-25.