Programming by Chat: A Large-Scale Behavioral Analysis of 11,579 Real-World AI-Assisted IDE Sessions

Ningzhi Tang*
University of Notre Dame
Notre Dame, IN, USA
ntang@nd.edu

Chaoran Chen*
University of Notre Dame
Notre Dame, IN, USA
cchen25@nd.edu

Zihan Fang
Vanderbilt University
Nashville, TN, USA
zihan.fang@vanderbilt.edu

Gelei Xu
University of Notre Dame
Notre Dame, IN, USA
gxu4@nd.edu

Maria Dhakal
University of Notre Dame
Notre Dame, IN, USA
mdhakal@nd.edu

Yiyu Shi
University of Notre Dame
Notre Dame, IN, USA
yshi4@nd.edu

Collin McMillan
University of Notre Dame
Notre Dame, IN, USA
cmc@nd.edu

Yu Huang
Vanderbilt University
Nashville, TN, USA
yu.huang@vanderbilt.edu

Toby Jia-Jun Li
University of Notre Dame
Notre Dame, IN, USA
toby.j.li@nd.edu

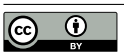
Abstract

IDE-integrated AI coding assistants, which operate conversationally within developers' working codebases with access to project context and multi-file editing, are rapidly reshaping software development. However, empirical investigation of this shift remains limited: existing studies largely rely on small-scale, controlled settings or analyze general-purpose chatbots rather than codebase-aware IDE workflows. We present, to the best of our knowledge, the first large-scale study of real-world conversational programming in IDE-native settings, analyzing 74,998 developer messages from 11,579 chat sessions across 1,300 repositories and 899 developers using Cursor and GitHub Copilot. These chats were committed to public repositories as part of routine development, capturing in-the-wild behavior. Our findings reveal three shifts in how programming work is organized: conversational programming operates as progressive specification, with developers iteratively refining outputs rather than specifying complete tasks upfront; developers redistribute cognitive work to AI, delegating diagnosis, comprehension, and validation rather than engaging with code and outputs directly; and developers actively manage the collaboration, externalizing plans into persistent artifacts, and negotiating AI autonomy through context injection and behavioral constraints. These results provide foundational empirical insights into AI-assisted development and offer implications for the design of future programming environments.

CCS Concepts

• **Software and its engineering → Software creation and management**; • **Human-centered computing → Empirical studies in HCI**.

*Both authors contributed equally to this research.



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834377

Keywords

Conversational Programming, AI Coding Assistants, Behavioral Log Analysis, Empirical Software Engineering

ACM Reference Format:
Ningzhi Tang, Chaoran Chen, Zihan Fang, Gelei Xu, Maria Dhakal, Yiyu Shi, Collin McMillan, Yu Huang, and Toby Jia-Jun Li. 2026. Programming by Chat: A Large-Scale Behavioral Analysis of 11,579 Real-World AI-Assisted IDE Sessions. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3834377

1 Introduction

IDE-integrated AI assistants such as GitHub Copilot (Chat)[1] and Cursor[2] have evolved beyond autocomplete into multi-turn conversational systems. Unlike general-purpose chat interfaces (e.g., ChatGPT in the browser), these assistants operate directly within the developer's working codebase, with access to project context, coordinated multi-file editing, terminal execution, and runtime inspection. This emerging practice, which we refer to as *AI-assisted conversational programming in IDE-native settings*, represents a shift in how developers work: rather than writing code directly, developers increasingly express intent and iteratively steer AI-generated outputs through dialogue. In a more extreme form, this delegation can resemble *"vibe coding"* [12, 47], where substantial implementation is offloaded to AI with limited inspection; existing reports indicate that some codebases are now 95% AI-generated[3].

This shift indicates that IDE-native conversational programming is rapidly becoming a prominent mode of software development. Yet, despite widespread adoption, empirical understanding of how developers actually converse with these assistants in real-world environments remains limited. Characterizing these interactions is essential both for understanding modern development workflows and

[1] https://github.com/features/copilot
[2] https://cursor.com/
[3] https://techcrunch.com/2025/03/06/a-quarter-of-startups-in-ycs-current-cohort-have-codebases-that-are-almost-entirely-ai-generated/

for designing more effective next-generation assistants. Currently, the empirical evidence is constrained in two complementary ways.

First, existing studies lack either scale or ecological validity. Video-based studies analyze publicly shared YouTube recordings in which developers code while narrating their process to an audience [12, 47], but this performative setting may systematically alter behavior through audience effects [2], and the largest prompt-level analysis of this kind covers only 254 prompts [12]. Controlled user studies assign developers researcher-defined tasks on unfamiliar codebases [10, 19, 29], a setting that differs markedly from everyday development, where goals are self-directed, context accumulates over time, and interaction is embedded in ongoing work [52]. As a result, the range of observable goals and interaction styles remains constrained. Finally, interview- and practitioner-based accounts [45, 50] are also subject to recall and self-presentation bias [20, 32]. Together, these studies provide valuable early insight, but they cannot capture how developers interact with AI coding assistants across the diverse self-directed tasks, codebases, and user populations of everyday software development.

Second, large-scale studies of developer-LLM interactions have primarily focused on browser-based chat rather than IDE-integrated environments. For example, DevGPT [62] curated developer-shared ChatGPT conversation links from GitHub, enabling subsequent studies of usage patterns [35], code quality [51], and refactoring practices [3]. More recently, CodeChat [68] assembled developer-LLM conversations from WildChat [66] and analyzed conversation-level topics and code defects. These datasets have enabled important large-scale analyses, but they do not capture interaction in IDE-native settings where assistants operate with repository context, tool access, and direct editing capabilities within an ongoing development workflow rather than as isolated query-answer exchanges.

To address these limitations, we present, to the best of our knowledge, the first large-scale empirical study of AI-assisted conversational programming in IDE-native settings. Our dataset comprises 74,998 developer messages from 11,579 chat sessions across 1,300 repositories and 899 developers using Cursor and GitHub Copilot. These sessions were neither required nor observed by researchers. Developers committed their chat histories to public repositories as part of their workflow, capturing self-directed development activity in the wild. The dataset reflects substantial diversity: tasks range from web development to AI/ML engineering, messages are written in over 20 different natural languages, and repositories span a wide range of scales and maturity levels. This large-scale behavioral evidence enables us to characterize how developers structure tasks, delegate work to AI assistants, and recover from failures in IDE-native conversational programming. Figure 1 illustrates a representative chat session.

Using this dataset, we characterize conversational programming at two complementary levels, developer message intents and session-level structure, investigating the following research questions (RQs):

- **RQ1:** *What behavioral intents do developers express in their messages to AI coding assistants?* We develop a multi-label behavioral intent taxonomy through iterative abductive coding [57], comprising 7 main categories and 20 subcategories. We then use an LLM-based classifier to label all 74,998 messages, validated against a human-annotated reference set of 400 messages (macro-averaged F1 = 0.802).
- **RQ2:** *What recurring behavioral archetypes and intent dynamics characterize conversational programming sessions?* We represent each session as an ordered sequence of intent labels and apply hierarchy-aware edit-distance-based [33] sequence clustering [28] to 4,864 sessions with at least four messages, identifying six recurring behavioral archetypes. We then analyze intent dynamics through within-session transition patterns, session-boundary transitions, and session evolution from opening turns to later messages.

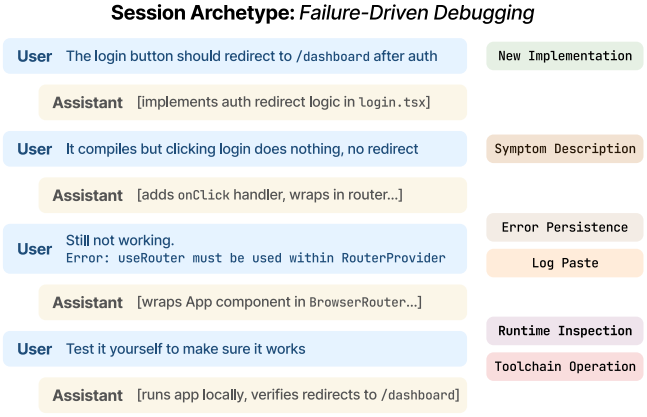


Figure 1: Illustrative conversational programming session annotated with behavioral intent labels. The interaction exemplifies the *Failure-Driven Debugging* archetype.

We highlight the following key results.

(1) **Conversational programming operates as progressive specification rather than upfront task description.** Developers steer the AI toward desired outputs through successive instructions (Finding 1); opening turns frame the task before sessions shift toward responding to AI outputs and emerging failures (Finding 11); and iterative modification and debugging tend to persist across consecutive turns (Finding 9).
(2) **Developers redistribute cognitive work to AI.** They report symptoms and machine outputs rather than diagnosing bugs (Finding 2), inquire about system behavior rather than reading code (Finding 3), and delegate validation to AI through both static code review and runtime inspection requests (Finding 4).
(3) **Developers actively manage the collaboration rather than treating AI as a passive tool.** They request persistent planning documents to externalize intent (Finding 5), negotiate AI autonomy through context injection and explicit action constraints (Finding 6), and start new chat sessions to refresh context while preserving task continuity (Finding 10).
(4) **Conversational programming exhibits structured interaction modes at the session level.** Most sessions are short and task-focused, but a long-tailed minority of sessions sustains extended iterative refinement (Finding 7). Six recurring archetypes emerge from clustering, ranging from failure-driven debugging to extended co-development (Finding 8).

2 Background and Related Work

2.1 Empirical Studies of AI Programming in IDE

AI coding assistants have evolved from autocomplete tools into conversational and increasingly agentic systems. Early systems (e.g., the initial release of GitHub Copilot in 2021) supported only inline code completion, offering localized assistance within a single file or completion context, without invoking external tools, executing code, or coordinating edits across files. Empirical studies from this period [7, 36, 44, 54–56] reflect these limitations, examining interactions that were largely single-turn and narrowly scoped, in which the assistant primarily served as a reactive suggestion engine for short code spans within the developer's active cursor context. Sustained multi-turn delegation and iterative steering, which are central to more capable assistants, therefore remained largely beyond the scope of this earlier body of work.

This began to change with the rise of conversational interaction in IDEs. Tools such as Cursor and later versions of GitHub Copilot moved beyond inline completion to support chat-based interaction within the IDE and across broader project contexts. By February 2025, this shift had become culturally visible: Karpathy coined the term *"vibe coding"* to describe accepting AI-generated code without reading diffs and pasting errors without diagnosis[4].

This emerging practice has attracted growing empirical attention through several forms of evidence. Video- and observation-based studies analyze publicly shared sessions, including YouTube and Twitch livestreams, to characterize prompting, delegation, and debugging behavior [12, 47]. For example, Chou *et al.* [12] analyze livestreams and opinion videos, showing that vibe coding ranges from near-total delegation to more inspection-heavy collaboration. A smaller body of work draws on interviews, online discourse, and practitioner accounts to identify recurring themes such as co-creation, flow, trust, and concerns about quality assurance [18, 45]. Controlled and field studies have examined increasingly agentic workflows. For example, Chen *et al.* [10] compare GitHub Copilot with the CLI-based agent OpenHands and find that higher automation can reduce user effort while raising transparency concerns. Kumar *et al.* [29] further show that incremental collaboration often outperforms one-shot delegation. Together, these studies provide important early insights, but they remain largely small-scale, observed, retrospective, or task-bounded, leaving in-the-wild IDE-native conversational programming undercharacterized.

2.2 Large-Scale Studies of Coding with LLMs

Existing empirical data for AI-assisted coding span benchmarks that evaluate model capabilities under standardized conditions [11, 15, 17, 26, 65, 67], artifact-level datasets that capture development outcomes (e.g., pull request acceptance) [34], and field measurements of AI tool usage at scale [14, 49]. While valuable, these resources measure task performance or downstream products rather than the interaction process itself. A separate line of work examines developer-provided context artifacts for coding agents, such as Cursor rules and `CLAUDE.md` files [25, 41], but these capture persistent project-level specifications rather than dynamic multi-turn interaction. The closest line of work to ours examines developer-LLM conversations directly, but primarily in general-purpose chatbots rather than IDE-native settings. DevGPT [62] and CodeChat [68] curate developer conversations from browser-based interfaces, i.e., ChatGPT and WildChat [66], enabling follow-up analyses [3, 16, 23, 35, 51]. Crucially, none of these datasets capture interaction in IDE-native settings, where assistants operate directly with repository context and tool access as part of an ongoing development workflow. This distinction is reflected in our findings: we replicate the predominantly short, task-delegation-oriented conversations observed in browser-based studies [23, 35], while behaviors such as runtime inspection and persistent planning documents surface only in IDE-native interactions (e.g., Findings 4–6).

Our analytical approach draws on methodological precedents from large-scale studies of general-purpose LLM usage. Studies of ChatGPT, Claude, and Microsoft Copilot employ LLM-based classifiers to categorize conversations by topic, task, and usage context at scale [5, 8, 13, 22]. These studies provide broad accounts of LLM use, but typically at the level of conversation topics or occupational tasks. Closer to our session-level focus, Mysore *et al.* [43] classify follow-up utterances in LLM-assisted writing and identify prototypical collaboration behaviors that account for much of the variation across multi-turn sessions. Related work operationalizes interaction dynamics such as autonomy and user oversight in agentic systems [39]. These studies establish the feasibility of large-scale behavioral log analysis for characterizing human-LLM interaction. Our work applies and extends this approach to IDE-native conversational programming, a setting whose affordances and workflows remain underrepresented in prior analyses.

3 Methodology

We conducted a mixed-methods study of AI coding assistant chats from public GitHub repositories. For RQ1, we analyzed message-level intents using a taxonomy and LLM-based classification (Section 3.2). For RQ2, we analyzed session-level archetypes and intent dynamics (Section 3.3).

3.1 Data Collection and Preparation

We collected AI coding assistant chat histories from publicly available GitHub repositories via SpecStory[5], a developer tool that automatically exports chat histories as timestamped Markdown files stored in each project repository. SpecStory supports multiple AI coding tools, including IDE-integrated assistants (i.e., Cursor, GitHub Copilot) and CLI-based agents (e.g., Claude Code).

To retrieve these exported histories at scale, we queried the GitHub Code Search API[6] over `.specstory/history/`. Because the API caps results at 1,000 per query, we partitioned searches by date prefix (e.g., files starting with `2025-06-15`), leveraging SpecStory's date-prefixed filename convention. All data were collected on March 4, 2026, capturing sessions from September 2024 to March 2026, spanning SpecStory's availability and the early adoption of chat-based AI coding assistants in software development.

[4] https://x.com/karpathy/status/1886192184808149383

[5] https://specstory.com/

[6] https://docs.github.com/en/rest/search/search

For each discovered file, we retrieved its raw Markdown via the Git Blobs API and parsed it into structured records. Specifically, we identified conversational turns using the role-prefixed markers introduced by SpecStory (e.g., "User," "Assistant," and "Agent") and treated the content associated with each marker as a single turn-level message for analysis. We provide the full parser implementation and format specifications in the replication package. We manually inspected a random sample of 100 parsed sessions and found turn identification and role assignment to be accurate in all cases. We then retained only user messages for subsequent analysis, as they capture developer intents.

Data Cleaning and Exclusion. Each file is uniquely identified by its content hash (SHA), which we used as the deduplication key across search results. This removed 3,512 duplicates. Manual inspection traced them mainly to developers duplicating chat histories within a single repository as snapshot backups (one developer alone accounted for 58% of duplicates; 2,474 → 444 sessions after deduplication); and less to cross-repository copying via cloning, templating, or forking. Our initial collection included 1,705 sessions exported from CLI-based agents (primarily Claude Code). We excluded these for two reasons: (1) their logs interleave user messages with tool outputs and system callbacks, obscuring user behavior; (2) they represent a different interaction paradigm in which users issue high-level tasks and agents autonomously plan and execute multi-step actions, rather than engaging in turn-by-turn conversational collaboration [10]. We therefore focused on IDE-integrated sessions and left CLI-agent behavior to future work.

After removing duplicates and excluding CLI-agent sessions, the dataset contained 76,231 user messages over 11,655 sessions. After the classification stage described in Section 3.2.2, we excluded 1,233 messages that do not have any classifiable behavioral intent, along with sessions with no remaining classifiable messages, yielding the final analytic dataset of **74,998 user messages** across **11,579 sessions**, from **1,300 unique repositories**, involving **899 developers**, identified by unique Git commit authors of chat history files.

Dataset Characteristics. The repositories in our dataset skewed toward personal and early-stage projects rather than established open-source software, with 67.3% having a single contributor, consistent with the overall distribution of public GitHub repositories [27]. The primary languages were TypeScript (25.8%), Python (22.0%), JavaScript (13.1%), and HTML (9.4%), with a long tail of additional languages (e.g., Shell, Java, Rust, C++), each under 2.1%. Most (90.8%) were created in 2025 or later, with a median of 15 commits over 4 active days, although a long tail includes larger projects with thousands of commits and repositories dating back to 2013.

Automated language identification [37] indicated highly multilingual user messages: English comprised 59.7%, followed by Chinese (18.5%) and Japanese (8.3%), with the remainder across a long tail of other languages. As many prompts intermix natural language with code and technical tokens, these proportions should be treated as contextual indicators rather than precise linguistic annotations.

3.2 Behavioral Intent Characterization

3.2.1 Iterative Codebook Development. We developed the *behavioral intent* taxonomy through four rounds of iterative abductive coding [57]. Prior schemes such as CUPS [42] were designed for code-completion interactions and do not transfer well to multi-turn conversational dynamics, so we built the taxonomy from scratch. Drawing on Speech Act Theory [6, 48], we labeled each message by its *illocutionary act*, i.e., the intentional action performed (e.g., requesting, asserting, or directing), rather than by surface form or topic. We used a *multi-label* scheme because many developers' turns in AI coding assistant chats contain multiple communicative acts. For example, *"@config.yaml — set up the database connection"* both provides context and requests implementation.

Three researchers with expertise in SE, AI, and HCI conducted the coding. In Round 1, we randomly sampled 300 messages with full conversational context and coded them through group discussion, resolving disagreements by consensus to establish an initial codebook. In Rounds 2–4, we used the same LLM-based classification pipeline later employed for full-dataset annotation (Section 3.2.2), but with the then-current version of the codebook, to generate *provisional* labels for all messages. Under each main category, we also included a temporary Others subcategory for messages that did not fit existing definitions. These provisional labels were used to support diagnostic sampling for codebook refinement. Specifically, from each subcategory, including Others, we randomly sampled 40 messages for human review with conversational context. When boundary cases in a subcategory remained unresolved after initial discussion, we reviewed an additional 20 messages from that subcategory. We used these reviews to assess whether category definitions were sufficiently precise, whether boundary cases were handled consistently, and whether messages assigned to Others exhibited coherent recurring patterns warranting new subcategories. Across all rounds, coding decisions were grounded in the iterative interplay between empirical data patterns and our emerging theoretical framework, following abductive coding practice [57].

We assessed structural saturation by tracking the number of subcategories added or substantively revised after each round, following previous saturation criteria [24, 38]. The counts decreased monotonically across the four rounds (9, 5, 1, 0), with no new subcategories in the final round, indicating that the codebook had stabilized. The final taxonomy comprises **7 main categories** and **20 subcategories** (Table 1). The full codebook, including definitions, key signals, examples, and boundary notes, is provided in our replication package.

3.2.2 LLM-Based Classification. We classified all 76,231 user messages with GPT-5 mini[7] via the OpenAI Batch API, with the finalized behavioral intent codebook provided as the system prompt. We chose GPT-5 mini for its balance of classification capability and cost at our scale, following a comparable large-scale classification study [8]. We adopted an LLM-based classification approach because behavioral intent labeling is multi-label, context-dependent, and semantically nuanced. Prior work [59] suggests that LLMs can achieve strong performance on text classification tasks when guided by structured instruction and schema constraints. Each classification request included the current message together with one turn of prior conversational context: the immediately preceding user message and the AI's most recent response. Context messages were truncated to 1,000 characters using a head-tail strategy

[7] https://developers.openai.com/api/docs/models/gpt-5-mini

Table 1: Behavioral Intent Taxonomy with distribution statistics. *% Msg* denotes the percentage of messages assigned to each category or subcategory, calculated over the final analytic dataset of 74,998 messages. Due to multi-labeling, percentages sum to more than 100% across categories, and subcategory percentages do not add up to their parent category total. *% Multi* denotes the percentage of messages carrying at least one additional label, computed within the same level.

Category / Subcategory	Definition	% Msg	% Multi
1. Code Authoring	**Requests to produce, modify, or adjust code.**	**34.53**	**35.71**
1.1 New Implementation	Build something that does not yet exist as functional code.	5.86	35.32
1.2 Iterative Modification	Adjust, constrain or refine existing or in-progress code.	24.84	43.06
1.3 Alignment Correction	Redirect the AI when its output runs but misses the user's actual intent.	7.21	83.63
2. Failure Reporting	**Report something broken, erroring, or behaving unexpectedly.**	**24.00**	**27.80**
2.1 Log Paste	Paste machine-generated error logs or stack traces.	8.84	39.74
2.2 Symptom Description	Describe a malfunction in natural language.	14.77	50.76
2.3 Error Persistence	Signal that a previous fix attempt has failed.	3.76	57.43
3. Inquiry	**Seek information, understanding, or advice without requesting code.**	**19.17**	**26.31**
3.1 Planning & Consultation	Plan next steps or seek advice on architectural choices.	7.81	37.48
3.2 Project Comprehension	Understand the current state of the project or existing code.	8.19	29.38
3.3 General Knowledge Query	Ask a project-agnostic technical or domain question.	3.56	11.35
4. Context Specification	**Provide information or instructions to shape AI understanding or action rules.**	**14.08**	**65.42**
4.1 Information Injection	Supply raw factual information for the AI to internalize.	8.46	61.86
4.2 Behavior Specification	Negotiate AI autonomy, operating boundaries, or persona.	6.14	77.88
5. Validation	**Ask the AI to evaluate or inspect code or execution output.**	**3.99**	**56.14**
5.1 Code Review	Inspect code for correctness, quality, or security without running it.	2.74	51.14
5.2 Runtime Inspection	Validate behavior by examining runtime output or test results.	1.26	67.69
6. Delegation	**Instruct the AI to perform development-workflow actions beyond authoring code.**	**16.48**	**45.94**
6.1 Documentation	Produce written artifacts such as docs, logs, or planning files.	6.85	55.93
6.2 Toolchain Operation	Take action in the development environment via commands or tools.	10.50	49.99
7. Workflow Control	**Manage session pace, direction, or state without new technical content.**	**11.47**	**24.36**
7.1 Confirmation	Signal that the AI's output was correct or a step succeeded (e.g., "works").	2.65	21.75
7.2 Continuation	Instruct the AI to proceed without introducing new requirements (e.g., "continue").	5.54	16.67
7.3 Deferred Debugging	Terse fix command relying entirely on session context (e.g., "fix it").	0.74	13.64
7.4 Deferred Implementation	Terse implementation command relying entirely on session context (e.g., "edit it").	0.78	23.00
7.5 Sentiment Expression	Express an emotional or social state (e.g., greetings, frustration, gratitude).	2.05	77.72

(60% prefix, 40% suffix), and the current message was truncated to 10,000 characters. This context design captures the most immediate conversational context for intent disambiguation while keeping prompts at a consistent and tractable length, consistent with prior work on LLM-based classification of conversational data [8, 43]. We enforced structured output via a JSON Schema, requiring the model to return a set of intent labels along with a brief textual justification for each. We enabled the model's built-in reasoning mode (i.e., chain-of-thought [60]), which produces intermediate reasoning steps before emitting the final JSON labels. This particularly helps disambiguate multi-intent and borderline cases.

To validate classifier accuracy, we drew a stratified random sample of 400 messages (20 per subcategory) covering all 20 subcategories; because messages carry multiple labels, a single message may appear in more than one subcategory sample. Two researchers independently annotated all 400 messages using the finalized codebook and full session context. Inter-rater reliability was assessed per subcategory by treating each label as a binary present/absent decision and computing Cohen's κ separately for each of the 20 subcategories, then macro-averaging; this yielded $\kappa = 0.669$, indicating substantial agreement [30]. Disagreements were resolved through discussion to produce an adjudicated gold-standard reference set. We then evaluated the LLM classifier against this reference set using the same per-category binary scheme, obtaining a macro-averaged F1 of 0.802 (precision = 0.774, recall = 0.851). The per-category results are reported in the replication package.

After classification, 1,233 messages (1.62%) were assigned only to the residual Others category. Manual inspection confirmed that this category captures inputs without recoverable communicative intent, e.g., accidental keystrokes, parser artifacts such as `---`. We excluded these messages from all subsequent analyses and removed sessions with no classifiable messages, yielding the final analytic dataset described earlier.

3.2.3 Inductive Qualitative Coding. To characterize behavioral patterns, we conducted an inductive qualitative analysis following established thematic coding practices [31]. For each of the 20 subcategories, we randomly sampled 80 messages. Because this analysis was interpretive in purpose, focusing on identifying recurring behavioral patterns rather than fixed categorical labels, we prioritized consensus-based thematic synthesis [40]. To ensure rigor, two researchers independently reviewed each sampled message with full session context and conducted open coding to identify recurring

behavioral patterns without predefined themes; they then discussed and reconciled their interpretations to produce consolidated thematic summaries. This sample size provided sufficient coverage for recurring themes to stabilize during analysis, consistent with prior discussions of thematic saturation in qualitative research [21]. The resulting thematic summaries and illustrative quotes are reported alongside each finding in Section 4.1.

3.3 Session Archetypes and Dynamics Analysis

3.3.1 Session Clustering. To complement the message-level analysis in Section 3.2, we examined whether AI-assisted conversational programming exhibits recurring session-level archetypes. Rather than clustering raw message text, we represented each session as a sequence of intent labels, thereby focusing on behavioral structure rather than surface linguistic variation. We restricted clustering to sessions with at least four user messages (n = 4,864), because shorter sessions contain too few within-session transitions to support meaningful sequence comparison using edit distance.

Session Representation. Each session s is represented as an ordered sequence of label sets:

$$s = (\ell_1, \ell_2, \ldots, \ell_T), \quad \ell_t \subseteq C$$

where T is the number of user messages and C is the set of 20 subcategories in Table 1. Each ℓ_t is a non-empty set of intent labels ($|\ell_t| \geq 1$), naturally accommodating our multi-label scheme.

Hierarchy-Aware Edit Distance. We measure the dissimilarity $d(s, s')$ between two sessions $s = (\ell_1, \ldots, \ell_n)$ and $s' = (\ell'_1, \ldots, \ell'_m)$ using a weighted Levenshtein edit distance [33], defined as the minimum total cost of transforming one sequence into the other via insertions, deletions, and substitutions, with operation costs:

- *Insertion or deletion* of an element ℓ_t: cost $w_{\text{indel}} = 1.0$
- *Substitution* of ℓ with ℓ': cost $\text{sub}(\ell, \ell')$

The substitution cost between two label sets is defined as the mean pairwise label cost:

$$\text{sub}(\ell, \ell') = \begin{cases} 0 & \text{if } \ell = \ell' \\ \dfrac{1}{|\ell||\ell'|} \displaystyle\sum_{a \in \ell} \sum_{b \in \ell'} c(a, b) & \text{otherwise} \end{cases}$$

where the label-level cost for subcategory labels $a, b \in C$ is:

$$c(a, b) = \begin{cases} 0 & \text{if } a = b \\ w_{\text{same}} = 0.5 & \text{if } \text{main}(a) = \text{main}(b),\ a \neq b \\ w_{\text{cross}} = 1.0 & \text{if } \text{main}(a) \neq \text{main}(b) \end{cases}$$

where $\text{main}(\cdot)$ maps a subcategory to its parent main category.

This three-tier cost structure encodes the taxonomy hierarchy: substitutions within a main category (e.g., Symptom Description → Log Paste) are penalized less than substitutions across categories (e.g., Iterative Modification → Toolchain Operation), following established practice in sequence analysis that substitution costs should decline as elements become more semantically similar [1, 53]. When $|\ell| = |\ell'| = 1$, $\text{sub}(\ell, \ell')$ reduces to $c(a, b)$, recovering the standard single-label case.

To prevent sessions of different lengths from being dominated by the cumulative cost of indel (insertion and deletion) operations, we normalized the raw edit distance by the longer sequence length:

$$d(s, s') = \frac{d_{\text{raw}}(s, s')}{\max(|s|, |s'|)}$$

where $d_{\text{raw}}(s, s')$ is the raw weighted edit distance computed by dynamic programming. Since $w_{\text{cross}} = w_{\text{indel}} = 1.0$, $d_{\text{raw}}(s, s') \leq \max(|s|, |s'|)$, so $d(s, s') \in [0, 1]$.

K-Medoids Clustering. We computed the full pairwise distance matrix over all 4,864 sessions and applied K-Medoids (PAM) [28] with k-medoids++ initialization [4], operating directly on the precomputed distance matrix and yielding actual sessions as cluster representatives (medoids). We evaluated $k \in \{2, \ldots, 20\}$ using the silhouette score [46], which compares within-cluster similarity against neighboring clusters. The score peaked at $k = 6$ (silhouette = 0.0703), which we adopted as the final model. Absolute silhouette values were low across all tested k (0.0367–0.0703), indicating weakly separated clusters. This likely reflects the nature of edit-distance sequence clustering, where sessions may share overlapping subsequences without forming sharply separated groups. We therefore interpret the six clusters conservatively as *prototypical archetypes*, i.e., descriptive anchors of dominant behavioral patterns, rather than mutually exclusive natural classes. To characterize each archetype, two researchers independently examined the medoid sequence and a random sample of 30 sessions per cluster, then consolidated observations into a cluster label and narrative description. We additionally created an interactive visualization tool displaying the t-SNE projection [58] with per-session label sequences and cluster assignments[8].

3.3.2 Intent Dynamics Analysis. To complement clustering, we conducted three auxiliary analyses of behavioral intent dynamics.

For *within-session transition analysis*, we computed lift-weighted Markov transition probabilities between consecutive intent labels within sessions. Lift is defined as $\text{lift}(i \to j) = P(j \mid i)/P(j)$, where i and j are intent labels; this normalizes raw transition counts by the marginal frequency of the target label, with lift > 1 indicating reinforced transitions and lift < 1 indicating suppressed ones. For multi-label messages, each consecutive pair was distributed over the Cartesian product of label sets with weight $1/(|\ell_t| \times |\ell_{t+1}|)$. To characterize short-range continuity, we also measured lengths of consecutive repetitions of the same subcategory within a session.

For *session-boundary transition analysis*, we paired the last message of one session with the first message of the chronologically next session in the same repository and computed boundary lift, defined as $\text{lift}_{\text{boundary}}(i \to j) = P(\text{first}_j \mid \text{last}_i)/P(\text{first}_j)$, using the same multi-label weighting scheme as above. Comparing this quantity with within-session lift reveals whether session-boundary transitions differ from ordinary turn-to-turn transitions.

For *session evolution analysis*, we compared opening messages with later turns in terms of intent-category proportions. We also examined how intent-category proportions shift across normalized message positions by mapping each session to a 100-position uniform grid via linear interpolation and fitting linear regressions to assess monotonic trends. Because opening turns strongly shaped these trajectories, we estimated an opening-excluded version that

[8] Available at https://conversational-programming-clusters.netlify.app/

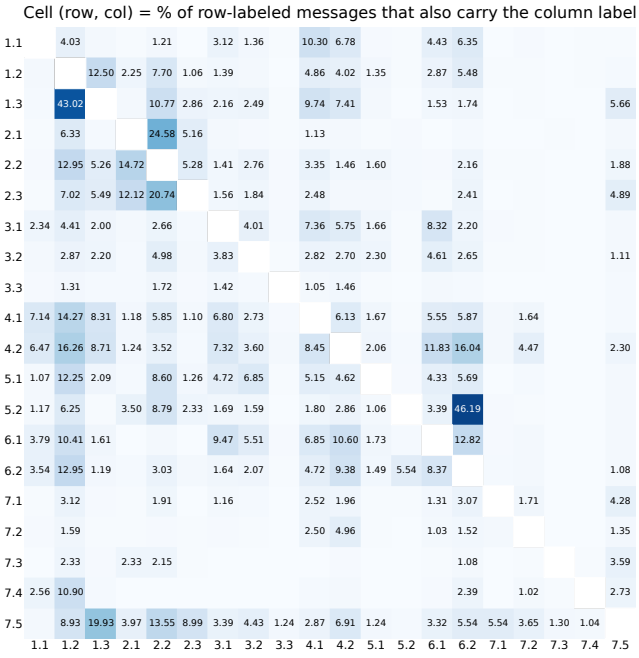


Figure 2: Co-occurrence patterns among behavioral intent subcategories. Cell values show the percentage of messages carrying the row label that also carry the column label; values below 1% are hidden. Diagonal cells are masked. Subcategory indices correspond to Table 1.

removed the first user message; unless otherwise noted, results refer to this specification. We applied the same procedure to message length (in characters) and labels per message.

4 Results

We report results in two parts. We first characterize the behavioral intent landscape at the message level (RQ1), and then examine session-level archetypes and dynamics (RQ2).

4.1 RQ1. The Behavioral Intent Landscape

Table 1 shows the overall distribution of behavioral intents. Code Authoring was the most prevalent category (34.53%), followed by Failure Reporting (24.00%), Inquiry (19.17%), Delegation (16.48%), Context Specification (14.08%), and Workflow Control (11.47%), whereas Validation was comparatively rare (3.99%). Multi-intent prompting was common: 29.87% of messages carried more than one label (mean = 1.33), though cases with three or more were rare (2.45%). As shown in Figure 2, co-occurring labels were structurally patterned rather than arbitrary. Multi-labeling rates were highest for Alignment Correction (83.63%), Behavior Specification (77.88%), and lowest for brief conversational control signals such as Confirmation (21.75%) and Continuation (16.67%). Intent distributions were broadly consistent across the two largest language subsets, TypeScript/JavaScript/HTML (n = 40,472 messages) and Python (n = 13,537): all category differences fell within 8.5 percentage points, the largest being lower Code Authoring (27.82% vs. 36.34%) and higher Delegation (21.78% vs. 15.28%) in Python.

Finding 1. Conversational programming operates through progressive refinement rather than upfront specification: iterative modification and correction dominate over new implementation requests. Although Code Authoring was the most prevalent (34.53%), New Implementation was rare (5.86%), compared to Iterative Modification (24.84%) and Alignment Correction (7.21%), the latter co-occurring with Iterative Modification in 43.02% of cases. Rather than issuing complete task descriptions upfront, developers predominantly revised and refined partially realized outputs, relying on accumulated context to keep correction turns minimal (e.g., *"sans tiret"*, *"Nope, still gray."*).

This iterative pattern arose from two sources: realigning the assistant when its output diverged from intent (e.g., *"Why aren't you following @user-preferences.mdc right now?"*), and accommodating evolving requirements as goals shifted (e.g., *"Can we convert back to cmdliner?"*) or surrounding system changed (e.g., *"I added a* `course_name` *field on the backend; update the frontend accordingly"*).

Finding 2. Developers delegated error diagnosis to AI by reporting symptoms and machine outputs rather than articulating code-level causes. Failure Reporting appeared in 24.00% of all messages, making it the second most common category, with Symptom Description (14.77%) more prevalent than Log Paste (8.84%). When problems arose, developers commonly supplied raw machine outputs (compiler errors, runtime traces, terminal prints, etc.), described observable mismatches between expected and actual behavior (e.g., *"table doesn't update until I refresh"*), or issued terse follow-ups (e.g., *"still stuck"*) that depended almost entirely on prior context. Troubleshooting thus reflected a redistribution of diagnostic work: the assistant took on causal interpretation while developers acted primarily as reporters of failure evidence.

Finding 3. When seeking to understand existing systems, developers queried AI about behavior, outputs, and feature logic, with less emphasis on code semantics. Project Comprehension accounted for 8.19% of all messages. Prompts were commonly posed at the level of behavior or interpretation (e.g., *"How does the system determine the icon's latitude and longitude?"*, *"I don't understand the results—does this test now invalidate the hypothesis?"*), or used to verify developers' own interpretation of the system (e.g., *"If I set* `progress<=0.05` *here, would that imply that … ?"*). Code-centric queries about what specific code does or how it is structured were comparatively rare (e.g., *"what is calling* `combinedUrlActions`*?"*).

Finding 4. Developers used AI to evaluate code through both static review and dynamic runtime inspection. Validation appeared in 3.99% of all messages, with Code Review (2.74%) and Runtime Inspection (1.26%) as its two components. Code review requests ranged from focused inspection to broader correctness checks (e.g., *"Is the current implementation correct in this form?"*, *"Make sure the modal is rendered properly on mobile"*), and frequently co-occurred with Iterative Modification (12.25%) and Symptom Description (8.60%), indicating that review was embedded in broader cycles of refinement and debugging. Runtime Inspection extended this evaluative role to execution results (e.g., *"Please run that command in a terminal so you can check the results"*), with nearly half of such messages (46.19%) also carrying Toolchain

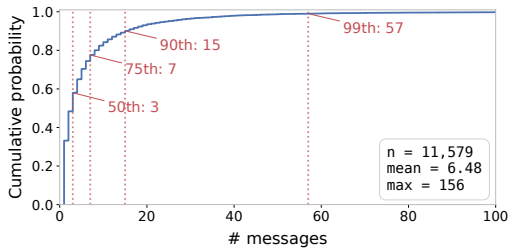


Figure 3: Cumulative distribution of user messages per session (n = 11,579). Dotted lines mark selected percentiles.

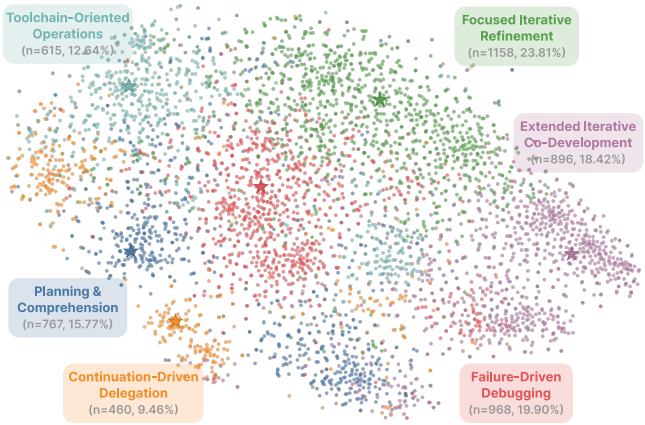


Figure 4: Session archetype structure visualized as a t-SNE projection [58] of 4,864 sessions. Stars denote medoids.

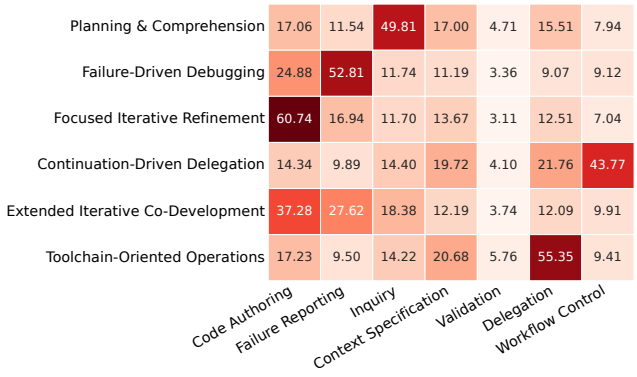


Figure 5: Behavioral intent composition of each session archetype (% of messages per main category).

Operation, suggesting that developers relied on the assistant not only to run code but also to evaluate its runtime behavior.

Finding 5. Developers used AI-generated documentation to externalize plans, explanations, and progress into persistent artifacts for both future AI use and human understanding. Documentation appeared in 6.85% of messages, commonly taking the form of Markdown-based planning, requirements, and progress files (e.g., `TODO.md`, `PROGRESS.md`). Developers created such artifacts primarily for future AI use: to guide implementation (e.g., *"read* `architecture.md` *for context then fully implement"*), to validate code against written constraints (e.g., *"check whether the implementation is consistent with the requirements and design documents"*), or to carry work across sessions (e.g., *"create a prompt for the next chat to carry out all the next steps"*). They also produced human-facing artifacts such as module summaries, deployment audits, and architecture diagrams. Across both uses, documentation served as persistent external memory, stabilizing context and decisions across turns and sessions.

Finding 6. Developers actively managed AI autonomy by shaping its context and constraining or delegating its behavior. Information Injection accounted for 8.46% of all messages, providing context developers believed the assistant should incorporate, including relevant resources (e.g., *"see the attached* `openapi.yaml` *"*), environment facts (e.g., *"I've restarted this instance of VS Code since our last interaction so the env variable activated"*), and project-state updates (e.g., *"I added two new states,* `step_dirty` *and* `code_dirty`*, to* `index.d.ts` *"*). Developers also explicitly set AI behavioral boundaries through Behavior Specification (6.14%), including hard constraints (e.g., *"only analyze; do not modify the code"*), conditions for human intervention (e.g., *"if you encounter an error, stop and ask me"*), and, in some cases, open-ended delegation (e.g., *"you decide"*), which traded direct oversight for reduced effort.

4.2 RQ2. Session Archetypes and Dynamics

We first characterize session-length distributions and recurring archetypes, then examine how behavioral intents evolve within and across sessions. Unless otherwise noted, group comparisons use Welch's t-test [61], and all reported contrasts and regression coefficients were statistically significant ($p < 0.001$).

4.2.1 Session Archetypes. **Finding 7. Although most sessions were short and task-focused, a long tail of extended sessions centered on iterative refinement and debugging.** The distribution of messages per session was right-skewed (median = 3; mean = 6.48; max = 156; Figure 3). Most sessions were short (≤3 messages; n = 6,715) and concentrated in one-shot task-completion intents, such as New Implementation (12.78% vs. 4.71%), Documentation (9.75% vs. 6.37%), and Deferred Debugging (1.98% vs. 0.54%). Longer sessions (≥4 messages; n = 4,864) instead showed higher proportions of reactive and iterative intents, including Alignment Correction (7.85% vs. 3.41%), Error Persistence (4.21% vs. 1.06%), and Confirmation (2.95% vs. 0.86%). We therefore restricted clustering to these longer sessions to ensure sufficient sequential signal for meaningful comparison (Section 3.3.1).

Finding 8. Clustering over longer sessions revealed six recurring session archetypes, each organized around a distinct behavioral mode. Figures 4 and 5 show the t-SNE projection [58] and main-category distributions. Five archetypes consisted of short sessions (median: 6–8 messages), each organized around a single dominant category, whereas *Extended Iterative Co-Development* stood apart with substantially longer sessions (median: 27) and the most balanced intent profile. We characterize each archetype below as descriptive prototypes of recurring interaction patterns rather than sharply separated clusters.

Planning & Comprehension. (n = 767, 15.77%). Sessions were dominated by Inquiry (49.81%), nearly evenly split between Planning & Consultation (20.99%) and Project Comprehension (20.37%). Developers used the assistant as a thinking partner, asking how existing systems worked and exploring architectural options. The category distribution was stable across session positions, indicating that planning and comprehension functioned as a sustained interaction mode rather than a preliminary phase before coding.

Failure-Driven Debugging. (n = 968, 19.90%). Sessions were dominated by Failure Reporting (52.81%), with Symptom Description (29.74%) and Log Paste (25.00%) as the primary subcategories. This archetype showed the clearest temporal progression among the six: sessions typically began with an Iterative Modification that triggered unexpected behavior, after which Code Authoring declined from 32.31% to 20.03% while Failure Reporting rose from 43.77% to 59.86%, indicating that an initial modification attempt gave way to sustained error-resolution cycles.

Focused Iterative Refinement. (n = 1,158, 23.81%). The largest archetype, with Code Authoring accounting for 60.74% of messages, was driven primarily by Iterative Modification (48.98%). Unlike *Failure-Driven Debugging*, Failure Reporting appeared in only 16.94% of messages, and the Code Authoring distribution remained stable across session positions (59.80% to 63.14%).

Continuation-Driven Delegation. (n = 460, 9.46%). Workflow Control was the dominant category (43.77%), with Continuation alone accounting for 31.00% of messages, the highest share of any subcategory across all archetypes. Sessions typically began with a substantive directive, often combining Toolchain Operation (13.32%) and Behavior Specification (11.69%), followed by a long sequence of continuation signals (e.g., *"continue"*, *"go on"*), suggesting that developers established the task and constraints upfront, then allowed the assistant to proceed with minimal intervention. This highly delegative pattern resembles the "vibe coding" practice.

Extended Iterative Co-Development. (n = 896, 18.42%). Distinguished primarily by session length (median: 27 messages; IQR: 20–40, compared with 6–8 in all other archetypes), this archetype also exhibited the most balanced intent profile, with no main category exceeding 37.3% and no strong temporal trends, suggesting that these long sessions sustained a mixed-mode collaborative process across a full development cycle. Qualitative inspection revealed recurring patterns, e.g., developers interleaved implementation, debugging, and validation within a single session, often across multiple work contexts.

Toolchain-Oriented Operations. (n = 615, 12.64%). Sessions were centered on Delegation (55.35%), driven by Toolchain Operation (38.55%) and Documentation (20.50%), with Context Specification elevated (20.68%) to supply environment details (e.g., port numbers, file paths). Developers instruct the assistant to perform environment and project operations, e.g., starting servers, managing Git branches, installing dependencies, and generating documentation, rather than to produce or debug application code.

4.2.2 Intent Dynamics. **Finding 9. Within-session transitions were strongly self-reinforcing, producing sustained runs of iterative modification and failure reporting, and recurrent micro-cycles centered on debugging and runtime validation.** All 20 subcategories exhibited self-loop lift (min = 1.80, max = 17.36; Figure 6). Measured as consecutive turns with the same intent, Iterative Modification showed the strongest continuity (mean run length = 1.57; 29.96% multi-turn runs; longest run = 27 turns), followed by Log Paste (1.43; 25.01%; 13 turns). Beyond self-loops, several cross-category transitions formed recurrent micro-cycles: a debugging loop linked Symptom Description → Error Persistence (lift = 2.87) and back (lift = 1.70); a validation loop connected Toolchain Operation and Runtime Inspection bidirectionally (lift = 2.25 and 1.88); and Iterative Modification frequently transitioned to Alignment Correction (lift = 1.57), reflecting corrective redirection when the assistant's output diverged from intent.

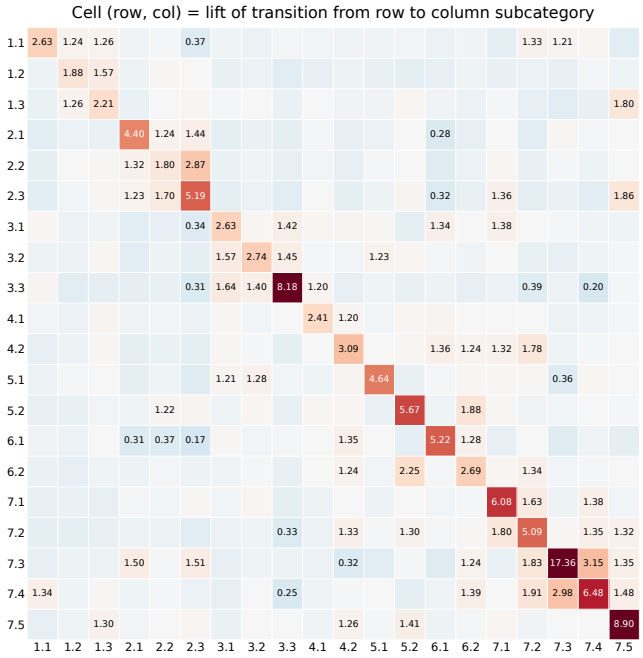


Figure 6: Transition patterns between behavioral intent subcategories within sessions, measured as Markov lift. Diagonal entries represent self-loop lift. Numerical values are shown for notably reinforced (lift ≥ 1.2) or suppressed (lift ≤ 0.4) transitions; all other cells display color only. Subcategory indices correspond to Table 1.

Finding 10. Session boundaries preserved task-level intent continuity while resetting conversational states, suggesting that developers used session breaks to refresh accumulated context rather than to switch tasks. Across session boundaries, self-loop lifts remained above 1 in all subcategories (min = 1.19, max = 20.77), indicating preserved task-level intents. However, context-dependent subcategories exhibited substantial declines: Confirmation dropped from a within-session lift of 6.08 to nearly zero, and Error Persistence declined from 5.19 to 1.19.

Finding 11. Sessions are not structurally uniform: developers begin by establishing task scopes and constraints, then progressively shift toward responding to intermediate AI outputs and emerging failures. Opening messages served a distinct

framing function. They were roughly twice as long as later turns (1,003 vs. 499 characters) and disproportionately featured setup-oriented acts, including New Implementation (15.89% vs. 4.03%), Information Injection (11.82% vs. 7.84%), and Behavior Specification (7.46% vs. 5.90%). In contrast, intents that depend on prior conversational context, such as Alignment Correction (0.73% vs. 8.40%) and Continuation (1.73% vs. 6.24%), were nearly absent.

Beyond the opening turn, sessions became increasingly reactive to AI outputs. Inquiry and Context Specification declined monotonically across session position ($R^2 = 0.70$ and 0.90), while Failure Reporting and Delegation increased ($R^2 = 0.66$ and 0.61). Message length also grew substantially (slope = 198.4, $R^2 = 0.91$), whereas labels per message remained nearly flat ($R^2 = 0.02$). Notably, Continuation, nearly absent at session onset, peaked in the early post-opening phase before declining steadily ($R^2 = 0.91$), suggesting that developers first established the task, then briefly sustained momentum before shifting to reactive engagement.

5 Threats to Validity

External validity. Our dataset reflects developers who use SpecStory and commit chat histories to public GitHub repositories, introducing selection bias that may underrepresent private and enterprise-level projects. Rather than approximating the full developer population, this dataset more directly reflects early adopters and heavy users of conversational programming workflows. The dataset is further scoped to IDE-integrated assistants and may not generalize to other forms of AI-assisted programming (e.g., CLI-based agents). At the same time, the naturalistic and unsolicited nature of the data avoids the observation bias common in lab or task-assigned settings. Notably, survivorship bias, i.e., developers preferentially sharing successful sessions, is partially mitigated by SpecStory's design: histories are auto-exported in batches with little manual curation, and our data contains substantial unresolved-failure signals that strong survivorship filtering would underrepresent. As model capabilities and user habits evolve rapidly, our results should be interpreted as a snapshot of early-stage conversational programming practice rather than stable long-term distributions.

Data completeness and construct validity. Our analysis is limited to text-based developer behavior preserved in SpecStory exports. Non-textual attachments (e.g., images or binary files), inline code references, and UI-level actions (e.g., regeneration, cancellation, code accept/reject) are not consistently recorded. Consequently, non-text user behaviors expressed through the interface, particularly Information Injection, are not represented. More broadly, our taxonomy captures developers' expressed behavioral intents, not their downstream code quality, correctness, or productivity effects.

Internal validity. Several stages of the analysis involve methodological choices that may influence the results. Codebook development and qualitative interpretation involve subjective judgment of researchers, which we mitigated through multi-researcher coding, iterative refinement, and consensus discussion. The LLM classifier was validated on a stratified sample covering all 20 subcategories. While some classification errors may remain and could affect downstream frequency estimates, our main findings rely on aggregate distributions and recurring patterns unlikely to be driven by isolated misclassifications. Other design choices, including sampling sizes, context truncation, minimum session length for clustering, and edit-distance parameterization, are theoretically motivated [1, 53] but non-unique; alternative specifications may yield different distributions or archetypes. We therefore interpret the clusters as descriptive archetypes rather than sharply bounded classes.

6 Discussion

6.1 Conversational Programming as Progressive Specification

Classical requirements engineering treats specification as a precondition for implementation. Conversational programming disrupts this sequencing: Alignment Correction and Symptom Description not only fix broken code but also introduce previously unarticulated constraints, distributing requirements articulation across the dialogue rather than concentrating it upfront (Finding 1, 11).

This underspecified communication is consistent with the bimodal distribution of message length: 33.77% of messages were 50 characters or shorter, while 9.02% exceeded 1,000 characters. At one extreme, terse cues such as *"Nope, still gray."* carried meaning only through context; at the other, developers embedded raw logs (Log Paste, median: 718 characters) or lengthy context blocks (Information Injection, median: 162 characters), leaving the assistant to determine what was relevant. This pattern likely differs from browser-based chatbot interactions [62, 68]: whereas both accumulate conversational history, IDE-native assistants additionally have direct access to the full codebase, runtime outputs, and development environment, and are expected to actively gather relevant context through file reads, codebase search, and command execution rather than relying on what the developer has made explicit.

One response to this challenge is to externalize plans into persistent documents, a practice formalized as *spec-driven development*: tools such as GitHub Spec Kit[9] and Amazon Kiro[10] treat lightweight documents (e.g., `design.md`, `tasks.md`) as the source of truth guiding AI agents through implementation. Our data suggest developers are already doing this organically: Documentation appeared in 6.85% of messages, with developers creating planning documents to record intent and progress across turns and sessions (Finding 5). However, because these documents are themselves generated from underspecified prompts, they introduce a potential compounded validation problem: developers must assess not only whether the code is correct, but whether the document captures their intent and whether the assistant's actions are consistent with it, a three-way alignment problem that warrants further investigation.

These findings carry direct implications for benchmark design. Dominant benchmarks such as SWE-bench [26] and HumanEval [9] assume a complete, self-contained task description, an assumption systematically violated in IDE-native use where specifications emerge over multiple turns from underspecified signals. A model optimized under this assumption may perform well on benchmarks while failing at tolerating ambiguity, tracking evolving intent, and proactively surfacing missing constraints. The emergence of CursorBench[11], which explicitly emphasizes underspecified tasks, suggests that tool builders are beginning to recognize this gap.

[9] https://github.github.com/spec-kit/
[10] https://aws.amazon.com/documentation-overview/kiro/
[11] https://cursor.com/blog/cursorbench

6.2 Distributed Cognition in Debugging, Comprehension, and Validation

Conversational programming appears to redistribute cognitive labor from developer to assistant: tasks such as fault diagnosis, code comprehension, and output validation are increasingly offloaded to AI, while developers retain the roles of symptom reporter, intent clarifier, and final evaluator. This redistribution is most visible in debugging. Although Failure Reporting was the second most prevalent category, developers rarely articulated code-level causes (Finding 2). Instead, they often relied on terse signals such as *"still not working"*, leaving diagnosis almost entirely to the assistant. These minimal messages function as low-cost probes: if the assistant recovers, the developer avoids the effort of diagnosis; if not, additional detail can be provided incrementally. This norm differs markedly from bug reporting to human collaborators, where detailed context signals both competence and respect for others' time. Sentiment data reinforced this interpretation: negative affect co-occurred most frequently with Alignment Correction (24.20%), Symptom Description (15.58%), and Error Persistence (11.01%), suggesting that emotional expressions serve less as social commentary than as compressed signals of unresolved failure.

As AI generates code faster than developers can inspect it, comprehension and validation are also displaced onto the assistant. Rather than reading implementations directly, developers queried the assistant about system behavior and feature logic to reconstruct how the code worked from the outside in (Finding 3). Validation followed the same pattern: developers delegated both static code review and runtime inspection to the assistant (Finding 4). This shift reduces the immediate burden of engaging with large volumes of generated code, but raises a deeper concern: when diagnosis, comprehension, and validation are all delegated to the same system, the assistant may become the only judge of both what the code does and whether it is correct, with no independent check on its own errors.

6.3 Managing an Opaque Collaborator

Conversational programming introduces a new coordination problem: although the state of the collaboration is nominally preserved in the chat history, it is difficult for developers to access in practice. In agentic IDE sessions, file reads, terminal commands, code edits, and long textual responses accumulate in a single stream, making it hard to monitor what the assistant knows and what it has done.

This opacity creates two related problems. First, developers may struggle to validate what the assistant has already done. To compensate, they issued explicit status checks (e.g., *"is your task finished?"*) and asked the assistant to externalize its own actions (e.g., *"Write down the changes in* `PROGRESS.md` *for me to track"*) (Finding 5). In effect, developers relied on assistant self-report because the interaction history had become too long and dense to inspect directly.

Second, developers may not reliably steer what the assistant should do next. To manage this, they injected missing context and imposed behavioral constraints, ranging from explicit hard stops to open-ended handoffs (Finding 6). The *Continuation-Driven Delegation* archetype represents the strongest form of the latter. As agentic systems take on longer task horizons, this coordination burden is likely to increase.

6.4 Future Work

Several directions remain beyond the scope of this study. First, our analysis captures what developers do but not whether those interactions achieved their goals; addressing this would require retrospective accounts of which requests and responses developers regarded as failures and why. Second, we exclude CLI-based agent sessions (e.g., Claude Code), which operate with greater autonomy and less turn-by-turn human direction; comparing conversational and more agentic workflows could clarify how increasing AI autonomy affects developer-AI interaction. Third, conversational programming may demand skills underemphasized in traditional programming education [63, 64], including requirement articulation, AI output evaluation, and context management, warranting curricular investigation. Finally, breakdowns by organization, domain, and time period are limited by our predominantly single-contributor, hobby-skewed sample and the tools' recent emergence. Extending such comparisons using our public taxonomy is a promising direction.

7 Conclusion

We presented the first large-scale empirical study of AI-assisted conversational programming in IDE-native environments, characterizing developers' behavioral intents and recurring session-level patterns from real-world developer-AI conversations. We hope this empirical foundation informs future AI coding assistants that better support iterative, context-dependent developer-AI collaboration.

Data Availability Statement

The replication package, including the codebook, classification prompts, and all analysis code, is available on GitHub[12] and archived at DOI 10.5281/zenodo.19248043. Raw chat-session data are excluded due to copyright and privacy considerations, as most source repositories do not carry explicit licenses permitting redistribution. However, the data can be fully recollected via the provided scraping pipeline[13]. Researchers interested in obtaining direct access may contact the first author.

Acknowledgments

This research was supported in part by a Google Cloud Research Credit Award, a Google Research Scholar Award, a gift from Adobe, an Edison Innovation Fellowship from the Notre Dame IDEA Center, and NSF grants CCF-2211428, CCF-2315887, CCF-2100035, CCF-2211429, CCF-2442682, and DGE-2622415. Any opinions, findings, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the sponsors. The authors thank Yuqi Wang from CREVIK for introducing us to SpecStory, without which this study would not have been possible.

References

[1] Andrew Abbott and Angela Tsay. 2000. Sequence analysis and optimal matching methods in sociology: Review and prospect. *Sociological methods & research* 29, 1 (2000), 3–33.

[2] Abdulaziz Alaboudi and Thomas D LaToza. 2019. An exploratory study of live-streamed programming. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 5–13.

[3] Eman Abdullah AlOmar, Anushkrishna Venkatakrishnan, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. 2024. How to refactor this code? an

[12] https://github.com/ND-SaNDwichLAB/empirical-conversational-programming
[13] https://github.com/TTangNingzhi/vibe-coding-scraper

exploratory study on developer-chatgpt refactoring conversations. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 202–206.
[4] David Arthur and Sergei Vassilvitskii. 2006. *k-means++: The advantages of careful seeding*. Technical Report. Stanford.
[5] Malika Aubakirova, Alex Atallah, Chris Clark, Justin Summerville, and Anjney Midha. 2026. State of AI: An Empirical 100 Trillion Token Study with OpenRouter. *arXiv preprint arXiv:2601.10088* (2026).
[6] John Langshaw Austin. 1975. *How to do things with words*. Harvard university press.
[7] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.
[8] Aaron Chatterji, Thomas Cunningham, David J Deming, Zoe Hitzig, Christopher Ong, Carl Yan Shan, and Kevin Wadman. 2025. *How people use chatgpt*. Technical Report. National Bureau of Economic Research.
[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[10] Valerie Chen, Ameet Talwalkar, Robert Brennan, and Graham Neubig. 2025. Code with me or for me? how increasing ai automation transforms developer workflows. *arXiv preprint arXiv:2507.08149* (2025).
[11] Wayne Chi, Valerie Chen, Ryan Shar, Aditya Mittal, Jenny Liang, Wei-Lin Chiang, Anastasios Nikolas Angelopoulos, Ion Stoica, Graham Neubig, Ameet Talwalkar, et al. 2025. EDIT-Bench: Evaluating LLM Abilities to Perform Real-World Instructed Code Edits. *arXiv preprint arXiv:2511.04486* (2025).
[12] Yi-Hung Chou, Boyuan Jiang, Yi Wen Chen, Mingyue Weng, Victoria Jackson, Thomas Zimmermann, and James A Jones. 2025. Building Software by Rolling the Dice: A Qualitative Study of Vibe Coding. *arXiv preprint arXiv:2512.22418* (2025).
[13] Beatriz Costa-Gomes, Sophia Chen, Connie Hsueh, Deborah Morgan, Philipp Schoenegger, Yash Shah, Sam Way, Yuki Zhu, Timothé Adeline, Michael Bhaskar, et al. 2025. It's About Time: The Temporal and Modal Dynamics of Copilot Usage. *arXiv preprint arXiv:2512.11879* (2025).
[14] Kevin Zheyuan Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. 2026. The effects of generative AI on high-skilled work: Evidence from three field experiments with software developers. *Management Science* (2026).
[15] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. 2025. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941* (2025).
[16] Zihan Fang, Jiliang Li, Anda Liang, Gina R Bai, and Yu Huang. 2025. A comparative study on chatgpt and checklist as support tools for unit testing education. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 871–882.
[17] Zihan Fang, Yifan Zhang, Yueke Zhang, Kevin Leach, and Yu Huang. 2025. DPO-F+: Aligning Code Repair Feedback with Developers' Preferences. *arXiv preprint arXiv:2511.01043* (2025).
[18] Ahmed Fawzy, Amjed Tahir, and Kelly Blincoe. 2025. Vibe Coding in Practice: Motivations, Challenges, and a Future Outlook–a Grey Literature Review. *arXiv preprint arXiv:2510.00328* (2025).
[19] Francis Geng, Anshul Shah, Haolin Li, Nawab Mulla, Steven Swanson, Gerald Soosai Raj, Daniel Zingaro, and Leo Porter. 2025. Exploring student-AI interactions in vibe coding. *arXiv preprint arXiv:2507.22614* (2025).
[20] Erving Goffman. 2023. The presentation of self in everyday life. In *Social theory re-wired*. Routledge, 450–459.
[21] Greg Guest, Arwen Bunce, and Laura Johnson. 2006. How many interviews are enough? An experiment with data saturation and variability. *Field methods* 18, 1 (2006), 59–82.
[22] Kunal Handa, Alex Tamkin, Miles McCain, Saffron Huang, Esin Durmus, Sarah Heck, Jared Mueller, Jerry Hong, Stuart Ritchie, Tim Belonax, et al. 2025. Which economic tasks are performed with ai? evidence from millions of claude conversations. *arXiv preprint arXiv:2503.04761* (2025).
[23] Huizi Hao, Kazi Amit Hasan, Hong Qin, Marcos Macedo, Yuan Tian, Steven HH Ding, and Ahmed E Hassan. 2024. An empirical study on developers' shared conversations with ChatGPT in GitHub pull requests and issues. *Empirical Software Engineering* 29, 6 (2024), 150.
[24] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 1110–1121.
[25] Shaokang Jiang and Daye Nam. 2025. Beyond the Prompt: An Empirical Study of Cursor Rules. *arXiv preprint arXiv:2512.18925* (2025).
[26] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).
[27] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*. 92–101.
[28] Leonard Kaufman and Peter J Rousseeuw. 2009. *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons.
[29] Aayush Kumar, Yasharth Bajpai, Sumit Gulwani, Gustavo Soares, and Emerson Murphy-Hill. 2025. Why AI Agents Still Need You: Findings from Developer-Agent Collaborations in the Wild. *arXiv preprint arXiv:2506.12347* (2025).
[30] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.
[31] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. 2017. *Research methods in human-computer interaction*. Morgan Kaufmann.
[32] Timothy C Lethbridge, Susan Elliott Sim, and Janice Singer. 2005. Studying Software Engineers: Data Collection Techniques for Software Field Studies: Lethbridge, Sim and Singer. *Empirical software engineering* 10, 3 (2005), 311–341.
[33] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.
[34] Hao Li, Haoxiang Zhang, and Ahmed E Hassan. 2025. The rise of ai teammates in software engineering (se) 3.0: How autonomous coding agents are reshaping software engineering. *arXiv preprint arXiv:2507.15003* (2025).
[35] Ruiyin Li, Peng Liang, Yifei Wang, Yangxiao Cai, Weisong Sun, and Zengyang Li. 2025. Unveiling the role of chatgpt in software development: Insights from developer-chatgpt interactions on github. *ACM Transactions on Software Engineering and Methodology* (2025).
[36] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2024. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*. 1–13.
[37] Marco Lui and Timothy Baldwin. 2012. langid. py: An off-the-shelf language identification tool. In *Proceedings of the ACL 2012 system demonstrations*. 25–30.
[38] Yunbo Lyu, Zhou Yang, Jieke Shi, Jianming Chang, Yue Liu, and David Lo. 2025. "My productivity is boosted, but…" Demystifying Users' Perception on AI Coding Assistants. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 191–203.
[39] Miles McCain, Thomas Millar, Saffron Huang, Jake Eaton, Kunal Handa, Michael Stern, Alex Tamkin, Matt Kearney, Esin Durmus, Judy Shen, Jerry Hong, Brian Calvert, Jun Shern Chan, Francesco Mosconi, David Saunders, Tyler Neylon, Gabriel Nicholas, Sarah Pollack, Jack Clark, and Deep Ganguli. 2026. *Measuring AI agent autonomy in practice*. https://anthropic.com/research/measuring-agent-autonomy
[40] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for CSCW and HCI practice. *Proceedings of the ACM on human-computer interaction* 3, CSCW (2019), 1–23.
[41] Seyedmoein Mohsenimofidi, Matthias Galster, Christoph Treude, and Sebastian Baltes. 2025. Context engineering for AI agents in open-source software. *arXiv preprint arXiv:2510.21413* (2025).
[42] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. In *Proceedings of the 2024 CHI conference on human factors in computing systems*. 1–16.
[43] Sheshera Mysore, Debarati Das, Hancheng Cao, and Bahareh Sarrafzadeh. 2025. Prototypical human-AI collaboration behaviors from LLM-Assisted writing in the wild. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. 16830–16857.
[44] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[45] 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).
[46] Peter J Rousseeuw. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 20 (1987), 53–65.
[47] Advait Sarkar and Ian Drosos. 2025. Vibe coding: programming through conversation with artificial intelligence. *arXiv preprint arXiv:2506.23253* (2025).
[48] John R Searle. 1969. *Speech acts: An essay in the philosophy of language*. Cambridge university press.
[49] Agnia Sergeyuk, Eric Huang, Dariia Karaeva, Anastasiia Serova, Yaroslav Golubev, and Iftekhar Ahmed. 2026. Evolving with AI: A Longitudinal Analysis of Developer Logs. *arXiv preprint arXiv:2601.10258* (2026).
[50] Judy Hanwen Shen and Alex Tamkin. 2026. How AI Impacts Skill Formation. arXiv:2601.20245 [cs.LG]
[51] Mohammed Latif Siddiq, Lindsay Roney, Jiahao Zhang, and Joanna Cecilia Da Silva Santos. 2024. Quality assessment of chatgpt generated code and their use by developers. In *Proceedings of the 21st international conference on mining software repositories*. 152–156.

[52] Dag IK Sjoberg, Bente Anda, Erik Arisholm, Tore Dyba, Magne Jorgensen, Amela Karahasanovic, Espen Frimann Koren, and Marek Vokác. 2002. Conducting realistic experiments in software engineering. In *Proceedings international symposium on empirical software engineering*. IEEE, 17–26.

[53] Matthias Studer and Gilbert Ritschard. 2016. What matters in differences between life trajectories: A comparative review of sequence dissimilarity measures. *Journal of the Royal Statistical Society Series A: Statistics in Society* 179, 2 (2016), 481–511.

[54] Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2024. Developer behaviors in validating and repairing llm-generated code using ide and eye tracking. In *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 40–46.

[55] Ningzhi Tang, David Meininger, Gelei Xu, Yiyu Shi, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2025. NaturalEdit: Code Modification through Direct Interaction with Adaptive Natural Language Representation. *arXiv preprint arXiv:2510.04494* (2025).

[56] Ningzhi Tang, Emory Smith, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2025. Exploring Direct Instruction and Summary-Mediated Prompting in LLM-Assisted Code Modification. In *2025 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 68–80.

[57] Stefan Timmermans and Iddo Tavory. 2012. Theory construction in qualitative research: From grounded theory to abductive analysis. *Sociological theory* 30, 3 (2012), 167–186.

[58] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[59] Zhiqiang Wang, Yiran Pang, and Yanbin Lin. 2023. Large language models are zero-shot text classifiers. *arXiv preprint arXiv:2312.01044* (2023).

[60] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[61] Bernard L Welch. 1947. The generalization of 'STUDENT'S'problem when several different population varlances are involved. *Biometrika* 34, 1-2 (1947), 28–35.

[62] Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2024. Devgpt: Studying developer-chatgpt conversations. In *Proceedings of the 21st international conference on mining software repositories*. 227–230.

[63] Yinuo Yang, Ashley Ge Zhang, Steve Oney, and April Yi Wang. 2026. SPARK: Real-Time Monitoring of Multi-Faceted Programming Exercises. *arXiv preprint arXiv:2601.22256* (2026).

[64] Yinuo Yang, Zheng Zhang, Ningzhi Tang, Xu Wang, Alex Ambrose, Nathaniel Myers, Patrick Clauss, and Toby Jia-Jun Li. 2026. Lessons from real-world deployment of a cognition-preserving writing tool: Students actively engage with critical thinking and planning affordances. *arXiv preprint arXiv:2603.15777* (2026).

[65] Songwen Zhao, Danqing Wang, Kexun Zhang, Jiaxuan Luo, Zhuo Li, and Lei Li. 2025. Is vibe coding safe? Benchmarking vulnerability of agent-generated code in real-world tasks. *arXiv preprint arXiv:2512.03262* (2025).

[66] Wenting Zhao, Xiang Ren, Jack Hessel, Claire Cardie, Yejin Choi, and Yuntian Deng. 2024. Wildchat: 1m chatgpt interaction logs in the wild. *arXiv preprint arXiv:2405.01470* (2024).

[67] Ming Zhong, Xiang Zhou, Ting-Yun Chang, Qingze Wang, Nan Xu, Xiance Si, Dan Garrette, Shyam Upadhyay, Jeremiah Liu, Jiawei Han, et al. 2025. Vibe Checker: Aligning Code Evaluation with Human Preference. *arXiv preprint arXiv:2510.07315* (2025).

[68] Suzhen Zhong, Ying Zou, and Bram Adams. 2025. Developer-llm conversations: An empirical study of interactions and generated code quality. *arXiv preprint arXiv:2509.10402* (2025).

Received 2026-03-26; accepted 2026-06-18