Commenting with Copilot: A Taxonomy and Multi-Year Analysis of Student Code-Generation Specifications

Nasser Giacaman
University of Auckland
Auckland, New Zealand
n.giacaman@auckland.ac.nz

Valerio Terragni
University of Auckland
Auckland, New Zealand
v.terragni@auckland.ac.nz

Paul Denny
University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

Viraj Kumar
University of New South Wales
Bengaluru, India
viraj.kumar1@unsw.edu.au

Abstract

As AI code tools become integrated into programming environments, students increasingly describe intended behavior in natural language and rely on these tools to generate code, shifting emphasis from code writing to specification. Yet little is known about the comments students write as specifications in AI-assisted programming tasks. We analyze a four-year dataset of undergraduate programming submissions and reflections from tasks in which students wrote comments to guide code generation and refined solutions using test-case feedback. We introduce a taxonomy spanning three dimensions: comment type, code expression level, and code construct. Using automated classification, we examine how these dimensions vary across attempts and how students describe the process in their reflections. Our findings show that students mostly wrote natural-language *What* comments, shifted toward *How* comments for more procedural constructs, and focused more on verifying generated code than on repeatedly rewriting comments.

CCS Concepts

• **Social and professional topics** → **Computing education**.

Keywords

AI-assisted programming, Code comprehension, Code generation, GitHub Copilot, Student comments

ACM Reference Format:
Nasser Giacaman, Valerio Terragni, Paul Denny, and Viraj Kumar. 2026. Commenting with Copilot: A Taxonomy and Multi-Year Analysis of Student Code-Generation Specifications. In *Proceedings of the 2nd ACM Virtual Global Computing Education Conference V.1 (SIGCSE Virtual 2026), November 12–15, 2026, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3795867.3830978

1 Introduction

AI code-generation tools are increasingly integrated into programming environments [33], changing the skills students need when learning to program [25]. Rather than writing every line, students can describe intended behavior in natural language and rely on AI tools to generate candidate implementations [7]. This has motivated new tasks in computing education. Prompt Problems ask students to solve tasks by writing prompts that elicit correct solutions from a code-generating model [8], while dialogue-based prompt programming environments support iterative natural-language interaction with AI models [27]. Other work asks students to clarify ambiguous specifications before prompting AI to generate solutions [24].

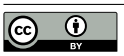
However, this skill is not straightforward for novices. Students can struggle to describe their intent, evaluate generated code, and revise prompts when the output is incorrect [13, 20]. Different patterns of AI use may have different implications for learning, with hybrid approaches involving human judgment and verification appearing more promising than simply asking an AI tool for complete solutions [16]. There are also concerns that over-reliance on AI might harm core skills such as code reading and comprehension.

In this paper, we analyze student-written comments from four years of an AI-assisted programming activity in an undergraduate object-oriented programming course. Students were shown short Java classes and asked to reproduce their behavior using GitHub Copilot. Rather than writing code themselves, they wrote comments in starter files to guide Copilot, submitted generated code to automated tests, and could revise comments across attempts. These comments are not conventional software-engineering comments, but student-written specifications for Copilot.

We introduce a multidimensional taxonomy that characterizes each comment by its purpose, level of code-like expression, and targeted code construct. Our contributions are a taxonomy for analyzing student comments in AI-assisted code generation and a multi-year analysis of how those comments vary across constructs, evolve across attempts, and are described in student reflections.

The study is guided by three research questions:

RQ1: What comments do students write to direct AI code generation, and how do they vary across code constructs?
RQ2: To what extent do students modify their comments?
RQ3: What are students' perspectives on guiding LLMs to generate code through comments?



SIGCSE Virtual 2026, Virtual Event, USA

ACM ISBN 979-8-4007-2506-7/2026/11
https://doi.org/10.1145/3795867.3830978

2 Related Work

2.1 AI Prompting and Code Generation

There is broad agreement in computing education that the skills students need are changing, with greater emphasis on reading code and writing natural-language prompts to solve programming tasks [25]. Prior work has introduced Prompt Problems, in which students solve programming tasks by writing prompts that guide an LLM to generate code [8]. These tasks position prompt writing as practice in expressing computational intent and reflect broader calls to align AI-era learning activities and assessments with intended outcomes, including the intended role of AI [2]. Related work has shown that students craft such prompts in different linguistic forms, including native-language, English, mixed-language, and code-like strategies, highlighting deliberate choices in how programming ideas are expressed in natural language [12, 26].

Reading and understanding code remain essential skills, and tasks such as "Explain in plain English" have long been used to assess code comprehension [10, 19]. Generative AI has recently been leveraged to provide feedback on such tasks by generating code from a student's explanation of a code fragment, thus connecting code comprehension with prompt formulation [9, 31, 32].

2.2 Prompt Revision and Interaction

AI-assisted programming is often iterative: students write an initial prompt, inspect generated code, test it, and decide whether to revise the prompt, edit the code, or try another strategy. Denny et al. showed that Copilot solved around half of a set of CS1 problems on its first attempt, and that many remaining failures could be resolved through natural-language changes to the problem description [7]. This demonstrates the value of prompt refinement, although their study used researcher-authored modifications rather than examining how students revise prompts during authentic learning activities. Nguyen et al. directly studied beginning programmers writing and revising prompts for code LLMs, finding that students often struggled to articulate intent, evaluate generated code, and decide how to modify prompts after failures [20].

Recent work examines richer forms of iterative interaction. Padurean et al. studied dialogue-based prompt programming, where students used multi-turn conversations, code execution, and reflection while solving programming tasks [27]. In subsequent work, Padurean et al. showed that students often use prompting to generate an initial solution and then move into short edit–run loops after failed executions, with manual editing increasing with task complexity [21]. These findings align with work on students' trust in Copilot, showing that attitudes toward AI-generated code are shaped by whether students can verify, debug, and understand the output [29]. Our study builds on this literature by examining a more constrained comment-based workflow, where students do not interact with the model through open dialogue, but revise comments embedded in source files to steer Copilot across attempts.

2.3 Comments as Specifications

A further strand of work examines how students use comments to express their understanding of code. In traditional contexts, comments are often treated as documentation for human readers, but they also reveal how students understand program purpose and structure. Kerschbaumer et al. analyzed student comments in a CS1 course and found that high-performing students wrote more task-related explanatory comments, while failing students more often described syntax or implementation details [17]. Prior work using the SOLO taxonomy has likewise analyzed students' natural-language explanations of programming processes, showing that such responses vary in completeness and in how they relate programming concepts [6]. Work on subgoal labels also shows that learners can identify functional units in code and express them through labels ranging from line-level descriptions to higher-level functional goals [14].

Prior work thus provides a foundation for analyzing comments as expressions of programming intent. In AI-assisted programming, however, comments can also function as prompts or executable specifications, written not only for future human readers but also to influence the code produced by an AI system. This connection is evident in code-generation-based grading and prompting-for-comprehension activities, where students' natural-language explanations of code are used to generate equivalent code and evaluate whether the description captures the intended behavior [9, 31, 32]. Our study extends this line of work by focusing on comments written inside source files to guide Copilot. We examine what comments students write, how these vary across code constructs, whether students revise comments across attempts, and how they perceive the process of guiding LLMs through comments.

3 Methods

3.1 Study Context and Dataset

This study was conducted in a second-year undergraduate object-oriented programming (OOP) course that used Java to cover classes, objects, encapsulation, interfaces, inheritance, polymorphism, abstract classes, design patterns, exception handling, and basic data structures. The Copilot activity ran in the fifth week of the semester, near the course midpoint, after students had covered core object-oriented concepts. Students had one week to complete the activity and were not explicitly taught professional prompting strategies for LLM-based code generation or traditional best practices for software comments; instead, the activity focused on using comments as a practical way to express intended behavior for GitHub Copilot.

Activity design. The handout comprised four tasks: three programming tasks and a final reflection. Students used VS Code with the GitHub Copilot[1] extension installed. GitHub's documentation presents two valid ways of obtaining Copilot suggestions[2]: (i) typing code to initiate inline suggestions, and (ii) describing intent in natural-language comments. We required students to begin with comments rather than handwritten code, both to make them articulate intended behavior before receiving suggestions and to better separate student-authored text from Copilot output.

For each programming task, students were shown a screenshot of a short completed Java class and asked to understand its behavior. The reference solution was provided only as a screenshot so students could inspect the target behavior without copying it into their

[1] Free for students through the GitHub Student Pack https://education.github.com/pack
[2] https://docs.github.com/en/copilot/how-tos/get-code-suggestions/get-ide-code-suggestions?tool=vscode#getting-code-suggestions-2

workspace. They then downloaded starter code with a blank class structure and method stubs and reproduced the behavior by writing only comments to guide Copilot, isolating how they translated that understanding into comment-based guidance for an LLM.

Students could work incrementally on parts of a class or on the whole class at once. To check progress, they copied the *entire* file, including comments and generated code, into CodeRunner [18], where automated tests showed which behaviors still failed, with no penalty for multiple attempts. Students were explicitly told not to edit the code directly, but to revise their comments in VS Code until the generated code passed the tests. Assessment was based on whether their comments led Copilot to generate code that passed the tests, not on exact match to the reference solution.

We assess test-suite adequacy on the reference implementation using two complementary criteria: code coverage with JaCoCo [11] and mutation score with PIT [4]. Coverage measures the proportion of lines and branch outcomes executed at least once, while mutation score measures the proportion of seeded syntactic faults detected by at least one test [15]. Per class, the tests achieve: `SimpleMath` 100% line, 100% branch, 82% mutation; `ShoppingCart` 100% line, 93% branch, 73% mutation; and `BankAccount` 97% line, 89% branch, 84% mutation. These values are high, and prior work shows mutation score strongly predicts real-fault detection [1, 15, 22]. This provides strong evidence that submissions passing the full test suite conform to the specified behavior, though no test suite is exhaustive.

Activity tasks. The activity included four tasks:

- `SimpleMath`: a small introductory task to help students get comfortable with the workflow; students implemented a simple arithmetic class with basic operations and a counter for negative results.
- `BankAccount`: a more substantial task in which students implemented an account class with balance management, transaction limits, transfers, and formatted account details.
- `ShoppingCart`: a task in which students implemented a shopping cart with item management, total calculation, and a cheapest-item-free discount for larger carts.
- Reflection: an open-ended reflection on students' use of Copilot, covering what they found easy or difficult, what they learned, and the tool's benefits, limitations, and dangers; minimum 150 words.

Dataset construction. Our dataset spans four yearly offerings of this activity, from 2023 to 2026. The raw data was exported from CodeRunner as submission logs, including all attempts where students copied their code into CodeRunner and ran the tests. This allowed us to capture the iterative process of writing comments, generating code, checking test feedback, and revising comments.

For analysis, these exports were converted into anonymized per-student folders. Each recorded submission event was reconstructed as an attempt containing the Java file with comments and generated code, the test results, and the reflection response when applicable. This reconstruction preserved the full state of each attempt, which allowed us to study not only the final submitted comments but also how comments changed across repeated submissions.

Units of analysis. Our main unit of analysis was the student-written comment used to direct code generation. Scripts extracted comments, merged adjacent line comments when they formed a single multi-line comment, and compared successive attempts for each student and question to identify comments as new, unchanged, modified, or deleted. These extracted comments were then coded using the taxonomy described in Section 3.2.

The dataset included 1,161 students and 10,257 recorded submission attempts. Across all three programming tasks, 3,458 of the 3,483 student submissions reached a fully passing solution (99.3% overall: `SimpleMath` 100%, `BankAccount` 98.8%, `ShoppingCart` 98.9%). Across all attempts, we extracted 136,424 comment instances; counting unchanged carry-forward comments only once reduced this to 52,172 comment instances for content analysis.

3.2 Taxonomy

We coded each extracted student comment on three dimensions: comment type, code expression level, and code construct, assigning one label per dimension. Comments with no meaningful directive, such as empty comments or metadata-only Javadocs, were ignored.

3.2.1 Comment Types. This dimension captures the main role the comment plays as an instruction to the LLM. We adapted the familiar *what/how/why* distinction from prior work on comment purpose [23, 34].

What comments describe the intended result, state, or local behavior that the code should produce. For example:

```
// Return false if the amount is less than 0
```

How comments describe a procedure, a sequence of steps, or local control flow. For example:

```
// Loop through the arraylist if name is equal to the
name of the item in the arraylist then remove it
```

Why comments explain the reason for an action or constraint. For example:

```
// Not returning the cheapest price as we need to find
the cheapest price in the cart first
```

3.2.2 Code Expression Level. This dimension captures how closely a comment resembles executable or near-executable code.

Strict-Code comments are essentially code or pseudocode that is *almost directly compilable*, requiring at most trivial surface changes such as adding a semicolon. For example:

```
// balance = balance + amount
```

Code-Like comments are written in words and very close to the local implementation, but unlike *Strict-Code*, would *not compile as are*; instead, they read like direct paraphrases of code. For example:

```
// set balance to balance + amount
```

Natural-Language comments are ordinary instructions or explanations that do not closely resemble code. For example:

```
// If more than 5 items are bought, discount cheapest one
```

3.2.3 Code Constructs. This dimension captures the program element targeted by a comment. The final set combined Java constructs with a few study-specific aggregations that better matched the task structure and the instructions students wrote:

Table 1: Comment type and expression level by code construct (ignoring unmodified comments across attempts)

		Comment Type			Expression Level		
Construct	**Count**	**What**	**How**	**Why**	**Natural Lang**	**Code-like**	**Strict Code**
Multi-Step-Block	11,067 (16.3%)	617 (5.6%)	**10,431 (94.3%)**	19 (0.2%)	**10,854 (98.1%)**	166 (1.5%)	47 (0.4%)
Conditional-Return	8,879 (13.1%)	**8,529 (96.1%)**	260 (2.9%)	90 (1.0%)	**8,745 (98.5%)**	88 (1.0%)	46 (0.5%)
Arithmetic-Update	8,941 (13.2%)	**8,564 (95.8%)**	343 (3.8%)	34 (0.4%)	**8,310 (92.9%)**	496 (5.5%)	135 (1.5%)
Conditional	8,739 (12.9%)	**7,075 (81.0%)**	1,601 (18.3%)	63 (0.7%)	**8,627 (98.7%)**	89 (1.0%)	23 (0.3%)
Return	9,169 (13.5%)	**9,131 (99.6%)**	15 (0.2%)	23 (0.3%)	**5,714 (62.3%)**	94 (1.0%)	3,361 (36.7%)
Assignment-Init.	6,889 (10.2%)	**6,618 (96.1%)**	261 (3.8%)	10 (0.1%)	**6,643 (96.4%)**	201 (2.9%)	45 (0.7%)
Field-Declaration	4,393 (6.5%)	**4,372 (99.5%)**	4 (0.1%)	17 (0.4%)	**4,274 (97.3%)**	85 (1.9%)	34 (0.8%)
Output-Formatting	2,803 (4.1%)	**2,505 (89.4%)**	295 (10.5%)	3 (0.1%)	**2,602 (92.8%)**	174 (6.2%)	27 (1.0%)
Method-Invocation	2,171 (3.2%)	**2,097 (96.6%)**	62 (2.9%)	12 (0.6%)	**2,115 (97.4%)**	30 (1.4%)	26 (1.2%)
Loop-Iteration	1,657 (2.4%)	707 (42.7%)	**949 (57.3%)**	1 (0.1%)	**1,577 (95.2%)**	52 (3.1%)	28 (1.7%)
Other	856 (1.3%)	**822 (96.0%)**	9 (1.1%)	25 (2.9%)	**839 (98.0%)**	14 (1.6%)	3 (0.4%)
Object-Creation	704 (1.0%)	**702 (99.7%)**	2 (0.3%)	0 (0.0%)	**684 (97.2%)**	12 (1.7%)	8 (1.1%)
Local-Declaration	434 (0.6%)	**433 (99.8%)**	1 (0.2%)	0 (0.0%)	**419 (96.5%)**	6 (1.4%)	9 (2.1%)
Ignored	1,085 (1.6 %)	-	-	-	-	-	-
Overall	67,787 (100.0%)	**52,172 (77.0%)**	14,233 (21.0%)	297 (0.4%)	**61,403 (90.6%)**	1,507 (2.2%)	3,792 (5.6%)

(1) **Field-Declaration**: declares a class-level member variable.
(2) **Local-Declaration**: declares a local variable in a method.
(3) **Assignment-Initialization**: assigns or initializes a value, including in a declaration.
(4) **Conditional**: an `if` or `if-else` branch.
(5) **Loop-Iteration**: repeated traversal using a `for`/`while` loop.
(6) **Object-Creation**: creates a new object, typically with `new`.
(7) **Method-Invocation**: calls an existing method.
(8) **Return**: returns a value directly, without the return being defined by a condition.
(9) **Conditional-Return**: returns a value because a condition is met, e.g. `if (...) return false`.
(10) **Arithmetic-Update**: changes a numeric value.
(11) **Output-Formatting**: prints, displays, or formats output.
(12) **Multi-Step-Block**: used when multiple actions or constructs are covered and no one category dominates.
(13) **Other**: fallback when none of the above fits clearly.

3.3 LLM Classification Pipelines

Recent work has explored LLM-supported labeling workflows, including deductive content analysis, collaborative codebook refinement, multi-agent approaches, and researcher-controlled coding [3, 5, 28, 30]. We therefore used GPT-5.4-family models with medium reasoning in two pipelines: one to label extracted comments, and one to organize student reflections while preserving traceability through supporting quotations and manual verification.

Comment classification. Each extracted comment was classified with an LLM using the full Java class from the corresponding attempt as context, assigning the comment type, expression level, and code construct labels described in Section 3.2. The labeled dataset was then used for the quantitative analyses reported in Section 4.

To assess the reliability of the LLM classification, we randomly sampled 200 extracted comments. One researcher independently labelled the sample using the taxonomy described in Section 3.2 while blinded to the LLM-generated labels. The human and LLM classifications showed high agreement across all three taxonomy dimensions. For Comment Type, agreement was 92.5% ($\kappa = 0.781$); for Expression Level, agreement was 94.0% ($\kappa = 0.771$); and for Code Construct, agreement was 93.5% ($\kappa = 0.927$).

Reflection classification. We analyzed the end-of-activity reflections with an LLM-assisted four-round pipeline. Reflection responses were exported from CodeRunner, assigned coded student identifiers, and analyzed in four rounds: (i) generating candidate codes for individual reflections, (ii) consolidating them into a shared code set, (iii) applying that code set back to the full set of reflections, and (iv) grouping the final codes into broader themes.

4 Results

The following results aggregate data across all four yearly offerings of the activity. As patterns were broadly similar across cohorts, results are presented in aggregate for clarity.

4.1 RQ1: Comment Types and Expression

Table 1 summarizes the distribution of comment types and expression levels across code constructs. Overall, comments were dominated by *What* comments (77.0%), followed by *How* comments (21.0%), while *Why* comments were rare (0.4%). Most constructs were likewise dominated by *What* comments. The exceptions were *Multi-Step-Block*, where 94.3% of comments were classified as *How*, and *Loop-Iteration*, where 57.3% were classified as *How*. This suggests that when logic involved multiple steps or repeated actions, students shifted from describing intended outcomes to describing the procedure needed to produce them. Comments were predominantly expressed in natural language, except for *Return* statements, which showed a fairly high proportion of *Strict-Code* expressions.

4.2 RQ2: Extent of Comment Modification

Figure 1 summarizes comment modifications across student attempts. The grey *Carried Forward* segment marks comments still present in the student's concluded attempt, rather than comments unchanged in still-active attempts. The dominance of *Unchanged* and *Carried Forward* suggests that students usually kept comments rather than substantially rewriting them. The figure shows only the first 10 attempts, covering 124,201 extracted comments (91.0%

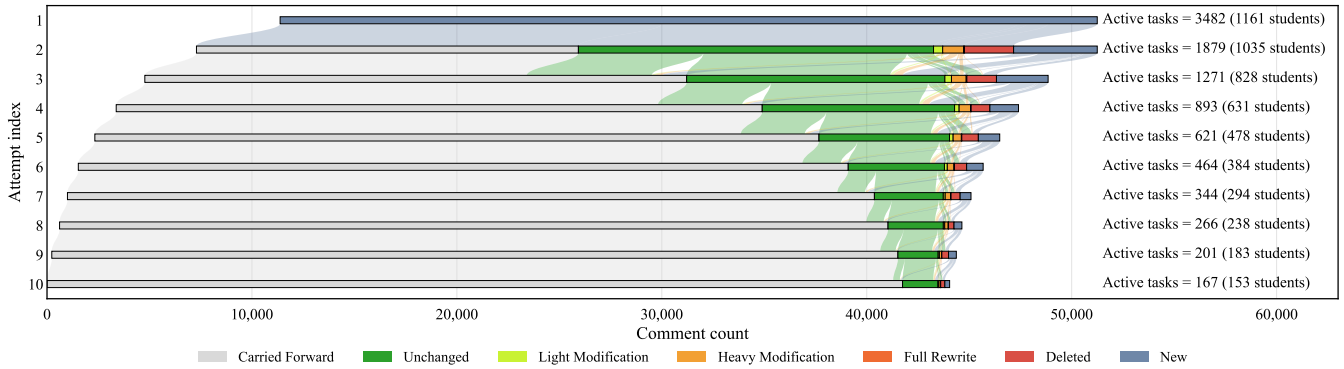


Figure 1: Pooled comment modification categories across all three exercises and four yearly cohorts for the first 10 attempts of each student–task sequence; the grey *Carried Forward* segment marks comments retained in the concluded attempt.

Table 2: Themes and code labels from student reflections

Rank	Theme	Students	Code labels
1	Speed and scaffolding	84.5% 979/1158	Faster drafting and coding (830), Autocomplete reduces writing effort (331), Intent to implementation details (198), Helps students get started (125), Time pressure boosts AI appeal (9)
2	Task fit	64.5% 747/1158	Best for boilerplate code (301), Best on familiar simple tasks (236), Struggles on complex specific tasks (219), Benefits depend on user readiness (136), Prompting slower for small tasks (135), Best when solution is known (125), Weaker on large codebases (40), Learned patterns shape quality (28)
3	Verification burden	57.3% 663/1158	Understand review and validate output (444), AI still needs oversight (418), Checking can erode time savings (99), Limited trust without understanding (77), Test against expected behavior (53), Easy suggestions invite overtrust (51), Can introduce errors (29)
4	Learning and agency	57.2% 662/1158	Overreliance can weaken learning (576), Understanding over AI output (170), AI-supported practice can teach (68), Can reduce agency and satisfaction (36)
5	Code quality	54.5% 631/1158	Output may be incomplete/wrong (409), May add unrequested code (99), Output often needs cleanup (92), Works but may be inefficient (91), Misses exact technical details (91), Plausible code can hide errors (73), Unfamiliar style hinders maintenance (25), Long outputs resist inspection (19), Mixed help on declarations details (11), May improve code consistency (10)
6	Prompt precision	54.0% 625/1158	Clear specific prompts help (405), State specifics explicitly (196), Vague prompts derail output (119), Right wording is tedious (85), Exact output details are hard (70), Right terms express intent (19)
7	Workflow adoption	53.3% 617/1158	Mixed but pragmatic value (567), Setup, responsiveness matter (60), Positive overall, few limits (29), Convenient editor use (15), AI use may keep growing (7), Compared with other tools (3)
8	Comment interface	51.6% 598/1158	Easy comment-based prompting (474), Comments express intended behavior (156), Autocompletes comments and prompts (90), Comments also aid understanding (39)
9	Context grounding	37.5% 434/1158	Comments may miss intent (254), Nearby context steers suggestions (170), Missing context hurts quality (68)
10	Selective partnership	36.5% 423/1158	Support tool, not replacement (285), Suggests simpler better methods (118), Suggestions shape next steps (38), Adapt and refine outputs (33), Scaffolds debugging when stuck (23)
11	Steering work	35.5% 411/1158	Iteratively refine prompts (218), Prompting improves with practice (92), Wrong paths can persist (66), Planning and code reading help (50), Break hard tasks into steps (44), Failures are hard to diagnose (14), Wording shifts can be unpredictable (9)
12	Predictive leaps	27.7% 321/1158	Impressively accurate (133), High-level intent, AI fills details (97), Small cues trigger large completions (66), Short cues work when obvious (57), Impressively capable, sometimes uncanny (50)
13	Ethical risks	17.9% 207/1158	Authorship, plagiarism, security concerns (122), Job replacement worries (61), Privacy and confidentiality concerns (23), Broader ethical, safety, social concerns (20), May enable harmful misuse (3)

overall) and 51,248 unchanged comments (98.2% of all unchanged comments), indicating that most comment writing and modification occurred early. Within this window, *Unchanged* (48.4%) and *New* (41.3%) dominate, followed by *Deleted* (6.0%); *Light Modification*, *Heavy Modification*, and *Full Rewrite* together account for only 4.4%.

These modification levels were assigned automatically using Levenshtein similarity between successive comments. Thresholds were selected following inspection of representative comment pairs to distinguish minor wording edits from substantial revisions. Similarity scores of at least 0.8 consistently reflected minor wording changes and were classified as *Light Modification*. Scores below 0.2 generally corresponded to comments that had been rewritten with substantially different wording or intent and were classified as *Full Rewrite*. Intermediate values were classified as *Heavy Modification*.

To examine where revision effort was concentrated, Figure 2 traces changes from question to construct to modification category for the top seven constructs. *BankAccount* dominates with the most change events. *Multi-Step-Block* is the clearest hotspot, followed by *Conditional-Return*, *Output-Formatting*, *Conditional*, and *Arithmetic-Update*. The widest flows end in *Heavy Modification*, especially for *Multi-Step-Block*, suggesting students more often needed substantial restructuring than minor edits.

4.3 RQ3: Student Perspectives

Table 2 summarizes 13 themes; the six most prevalent are discussed.

Speed and scaffolding. This was the dominant theme. Students often described comments as a quick way to turn ideas into code, especially when getting started or avoiding low-level syntax:

> *"It speeds up development by suggesting code in real time, reducing the need to manually write repetitive sections or search for solutions."*

Task fit. Copilot was not described as equally useful across tasks. Instead, they saw it as strongest on familiar or repetitive work, and weaker for niche, multi-step, or constrained requirements:

> *"Copilot tends to give solutions that are generic or repeated, and it can struggle to complete code that has to be done a specific way."*

Verification burden. Even when students valued the tool, they framed checking and interpreting output as their responsibility.

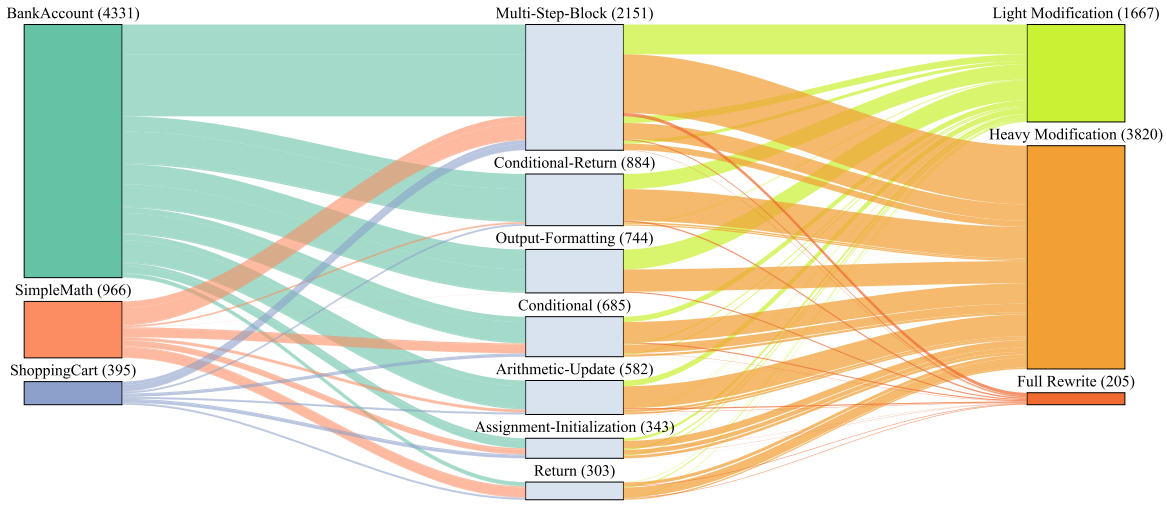


Figure 2: Changed comments from question to construct to modification category. The top seven constructs are shown (5,692 of 6,221 events); omitted constructs account for 529 events (8.5%).

This helps explain the RQ2 revision results: the low rate of explicit comment rewriting does not imply a smooth process:

> *"I had to keep checking the output and fix mistakes, especially when the logic was slightly wrong or when it misunderstood the instructions."*

Learning and agency. Some students described learning gains when using the output critically, while many worried that relying too heavily on Copilot would negatively impact learning:

> *"Heavily relying on Copilot as a crutch for programming limits your own ability to gain programming knowledge effectively and can be detrimental if you ever find yourself with a problem that generative AI cannot solve."*

Code quality. Students often noted that Copilot's output could look plausible yet be wrong or introduce unwanted behavior:

> *"Sometimes it tells you the wrong code and insists that it is correct and doesn't offer other options despite changing the description."*

Prompt precision. Students repeatedly reported that they had to spell out constraints, sequencing, and exact method choices if they wanted reliable results, especially on the more complex tasks:

> *"I learnt that you have to be very direct and specific of what you want the AI to do so it avoids going on tangent."*

5 Discussion

The results suggest that students usually approached comment-based code generation by stating intended behavior in plain language rather than writing code-like prompts. Most comments were *What* comments, especially for simpler constructs, while more procedural constructs drew more *How* comments. An interpretation is that students often described local outcomes when the behavior was easy to express, but shifted toward stepped guidance when the logic required sequencing or repetition.

The modification results support this picture, as most comments were either new or unchanged with relatively little rewriting. The reflections suggest that this does not mean the process was easy. Instead, many students described the main effort as checking generated code, interpreting test results, and deciding whether output matched intended behavior. This explains why comment revision was limited even though verification burden was a key theme; work shifted from rewriting prompts to reviewing generated code.

For teaching, these findings suggest that comment-based AI activities may be most useful when framed as exercises in understanding, expressing, and verifying behavior rather than simply obtaining code [9]. Students often described Copilot as most useful for familiar tasks, but less reliable for multi-step or tightly constrained ones. This matches the greater revision effort on procedural constructs. This supports careful AI use in which students remain responsible for interpreting and checking output, rather than treating the tool as an unquestioned answer source [29].

This study has several limitations. It comes from one course, one institution, and one activity design, so the results may not transfer directly to other settings. Students reproduced short Java examples from solution screenshots rather than solving open-ended problems, so the findings reflect a constrained programming task. The modification analysis captures only changes visible in submissions; IDE-only revisions overwritten before submission do not appear in our data. Passing the tests also does not guarantee complete semantic equivalence to the reference solutions. Finally, both the comment classification and reflection analysis relied on LLM-assisted pipelines, so classification errors exist.

6 Conclusions

This paper examined how students wrote comments as specifications in a four-year GitHub Copilot activity. Students primarily wrote natural-language *What* comments, shifted toward more procedural *How* comments for multi-step and iterative constructs, and revised comments less often than they reviewed and validated generated code. These results suggest that comment-based AI programming is not simply a matter of asking for code: it is a combined task of understanding behavior, expressing intent clearly, and judging whether the generated solution is correct. For computing education, this points toward treating specification and verification as central learning goals in AI-assisted programming, rather than framing AI tools primarily as shortcuts for code production.

References

[1] James H. Andrews, Lionel C. Briand, and Yvan Labiche. 2005. Is Mutation an Appropriate Tool for Testing Experiments?. In *Proceedings of the 27th International Conference on Software Engineering*. 402–411. doi:10.1145/1062455.1062530

[2] Claus Brabrand and Paul Denny. 2026. Constructive Alignment in the Age of AI. osf.io/preprints/edarxiv/m9yfk_v1

[3] Robert Chew, John Bollenbacher, Michael Wenger, Jessica Speer, and Annice Kim. 2023. LLM-Assisted Content Analysis: Using Large Language Models to Support Deductive Coding. arXiv:2306.14924 [cs.CL] https://arxiv.org/abs/2306.14924

[4] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*.

[5] Shih-Chieh Dai, Aiping Xiong, and Lun-Wei Ku. 2023. LLM-in-the-loop: Leveraging Large Language Model for Thematic Analysis. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). ACL, Singapore. doi:10.18653/v1/2023.findings-emnlp.669

[6] Adrienne Decker, Lauren E. Margulieux, and Briana B. Morrison. 2019. Using the SOLO Taxonomy to Understand Subgoal Labels Effect in CS1. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (Toronto ON, Canada) *(ICER '19)*. Association for Computing Machinery, New York, NY, USA, 209–217. doi:10.1145/3291279.3339405

[7] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1136–1142. doi:10.1145/3545945.3569823

[8] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett A. Becker, and Brent N. Reeves. 2024. Prompt Problems: A New Programming Exercise for the Generative AI Era. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (Portland, OR, USA) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 296–302. doi:10.1145/3626252.3630909

[9] Paul Denny, David H. Smith, Max Fowler, James Prather, Brett A. Becker, and Juho Leinonen. 2024. Explaining Code with a Purpose: An Integrated Approach for Developing Code Comprehension and Prompting Skills. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 283–289. doi:10.1145/3649217.3653587

[10] Max Fowler, Binglin Chen, and Craig Zilles. 2021. How should we 'Explain in plain English'? Voices from the Community. In *Proceedings of the 17th ACM Conference on International Computing Education Research* (Virtual Event, USA) *(ICER 2021)*. Association for Computing Machinery, New York, NY, USA, 69–80. doi:10.1145/3446871.3469738

[11] Marc R. Hoffmann, Brock Janiczak, Evgeny Mandrikov, and Mirko Friedenhagen. [n. d.]. JaCoCo Java Code Coverage Library. https://www.jacoco.org/. Accessed: 2026-05-08.

[12] David H. Smith IV, Viraj Kumar, and Paul Denny. 2024. Explain in Plain Language Questions with Indic Languages: Drawbacks, Affordances, and Opportunities. arXiv:2409.20297 [cs.CY] https://arxiv.org/abs/2409.20297

[13] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2022. Discovering the Syntax and Strategies of Natural Language Programming with Generative Language Models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 386, 19 pages. doi:10.1145/3491102.3501870

[14] Hyoungwook Jin and Juho Kim. 2024. CodeTree: A System for Learnersourcing Subgoal Hierarchies in Code Examples. *Proc. ACM Hum.-Comput. Interact.* 8, CSCW1, Article 31 (April 2024), 37 pages. doi:10.1145/3637308

[15] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, 654–665. doi:10.1145/2635868.2635929

[16] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane Ericson, David Weintrop, and Tovi Grossman. 2024. How Novices Use LLM-based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '23)*. Association for Computing Machinery, New York, NY, USA, Article 3, 12 pages. doi:10.1145/3631802.3631806

[17] David Kerschbaumer, Christoph Schatz, Thorsten Ruprechter, Christian Gütl, and Alexander Steinmaurer. 2025. Do Comments Matter? Investigating Students' Source Code Comment Behaviour and Its Relation to Academic Success in a CS1 Course. In *Futureproofing Engineering Education for Global Responsibility*, Michael E. Auer and Tiia Rüütmann (Eds.). Springer Nature Switzerland, Cham.

[18] Richard Lobb and Jenny Harlow. 2016. Coderunner: a tool for assessing computer programming skills. *ACM Inroads* 7, 1 (Feb. 2016), 47–51. doi:10.1145/2810041

[19] Laurie Murphy, Renée McCauley, and Sue Fitzgerald. 2012. 'Explain in plain English' questions: implications for teaching. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) *(SIGCSE '12)*. ACM, 385–390. doi:10.1145/2157136.2157249

[20] Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How Beginning Programmers and Code LLMs (Mis)read Each Other. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. ACM, New York, NY, USA, Article 651, 26 pages. doi:10.1145/3613904.3642706

[21] Victor-Alexandru Padurean, Alkis Gotovos, Ahana Ghosh, Paul Denny, Juho Leinonen, Andrew Luxton-Reilly, James Prather, and Adish Singla. 2026. Interleaving Natural Language Prompting with Code Editing for Solving Programming Tasks with Generative AI Models. https://arxiv.org/abs/2509.14088

[22] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation Testing Advances: An Analysis and Survey. In *Advances in Computers*, Atif Memon (Ed.). Vol. 112. Elsevier, 275–378. doi:10.1016/bs.adcom.2018.03.015

[23] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. 2019. Classifying code comments in Java software systems. *Empirical Software Engineering* 24, 3 (2019).

[24] Mrigank Pawagi and Viraj Kumar. 2024. Probeable Problems for Beginner-level Programming-with-AI Contests. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1* (Melbourne, VIC, Australia) *(ICER '24)*. Association for Computing Machinery, New York, NY, USA, 166–176. doi:10.1145/3632620.3671108

[25] James Prather, Juho Leinonen, Natalie Kiesler, Jamie Gorson Benario, Sam Lau, Stephen MacNeil, Narges Norouzi, Simone Opel, Vee Pettit, Leo Porter, Brent N. Reeves, Jaromir Savelka, David H. Smith, Sven Strickroth, and Daniel Zingaro. 2025. Beyond the Hype: A Comprehensive Review of Current Trends in Generative AI Research, Teaching Practices, and Tools. In *2024 Working Group Reports on Innovation and Technology in Computer Science Education* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 300–338. doi:10.1145/3689187.3709614

[26] James Prather, Brent N Reeves, Paul Denny, Juho Leinonen, Stephen MacNeil, Andrew Luxton-Reilly, João Orvalho, Amin Alipour, Ali Alfageeh, Thezyrie Amarouche, Bailey Kimmel, Jared Wright, Musa Blake, and Gweneth Barbre. 2025. Breaking the Programming Language Barrier: Multilingual Prompting to Empower Non-Native English Learners. In *Proceedings of the 27th Australasian Computing Education Conference (ACE '25)*. Association for Computing Machinery, New York, NY, USA, 74–84. doi:10.1145/3716640.3716649

[27] Victor-Alexandru Pădurean, Paul Denny, Alkis Gotovos, and Adish Singla. 2025. Prompt Programming: A Platform for Dialogue-based Computational Problem Solving with Generative AI Models. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Nijmegen, Netherlands) *(ITiCSE 2025)*. Association for Computing Machinery, New York, NY, USA, 458–464. doi:10.1145/3724363.3729094

[28] Tingrui Qiao, Caroline Walker, Chris Cunningham, and Yun Sing Koh. 2025. Thematic-LM: A LLM-based Multi-agent System for Large-scale Thematic Analysis. In *Proceedings of the ACM on Web Conference 2025* (Sydney NSW, Australia) *(WWW '25)*. ACM, 649–658. doi:10.1145/3696410.3714595

[29] Anshul Shah, Thomas Rexin, Elena Tomson, William G Griswold, Leo Porter, and Adalbert Gerald Soosai Raj. 2025. Evolution of Programmers' Trust in Generative AI Programming Assistants. In *Proceedings of the 25th Koli Calling International Conference on Computing Education Research (Koli Calling '25)*. ACM, New York, NY, USA, Article 12, 11 pages. doi:10.1145/3769994.3770029

[30] Ansh Sharma and James R Wallace. 2025. DeTAILS: Deep Thematic Analysis with Iterative LLM Support. In *Proceedings of the 7th ACM Conference on Conversational User Interfaces (CUI '25)*. Association for Computing Machinery, New York, NY, USA, Article 28, 7 pages. doi:10.1145/3719160.3735657

[31] David H. Smith, Paul Denny, and Max Fowler. 2024. Prompting for Comprehension: Exploring the Intersection of Explain in Plain English Questions and Prompt Writing. In *Proceedings of the Eleventh ACM Conference on Learning @ Scale* (Atlanta, GA, USA) *(L@S '24)*. ACM, 39–50. doi:10.1145/3657604.3662039

[32] David H. Smith and Craig Zilles. 2024. Code Generation Based Grading: Evaluating an Auto-grading Mechanism for "Explain-in-Plain-English" Questions. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 171–177. doi:10.1145/3649217.3653582

[33] Valerio Terragni, Annie Vella, Partha Roop, and Kelly Blincoe. 2025. The future of ai-driven software engineering. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025), 1–20.

[34] Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: automatically classifying and propagating natural language comments via program analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1359–1371. doi:10.1145/3377811.3380427