Informatics in Education, 2026, Vol. 0, No. 0, 1–37

DOI: https://doi.org/10.15388/infedu.2603.031

The Pedagogical Role of Source Code Comments in Programming Education: A Qualitative Systematic Review with Layered Evidence

Gonçalo SARMENTO[1], Manuel J. C. S. REIS[2,*, [0000-0002-8872-5721]]

[1] *Quinta de Prados, Vila Real, Universidade de Trás-os-Montes e Alto Douro, Portugal*
[2] *Quinta de Prados, Vila Real, Departamento de Engenharias & IEETA, Escola de Ciências e Tecnologia, Universidade de Trás-os-Montes e Alto Douro, Portugal*
e-mail: al82369@alunos.utad.pt, mcabral@utad.pt

Abstract. Source code comments are usually treated in software engineering as documentation artifacts that support readability, maintainability, and long-term comprehension. In programming education, however, comments may also function as pedagogical scaffolds by helping learners externalize reasoning, clarify intent, and reflect on code. This article presents a PRISMA-informed qualitative systematic review with layered evidence on the pedagogical role of source code comments in programming education. Searches in Scopus and ERIC produced 50 unique records; 36 were assessed in detail and resolved into 18 primary synthesis studies, 7 supporting/contextual studies, and 11 advanced-stage exclusions. Because the evidence base is heterogeneous, the review uses qualitative layered synthesis rather than meta-analysis. The findings suggest that comments are best understood pedagogically as explanation-centered learning supports, especially for code comprehension, self-explanation, debugging and reflective reasoning, and formative assessment of student thinking. Direct comment-focused evidence remains limited and concentrated mainly in higher-education and novice programming contexts; adjacent explanation-centered studies clarify plausible mechanisms but do not by themselves establish comment-specific effects. The review concludes that comments can become pedagogically meaningful when deliberately integrated as scaffolds for explanation, comprehension, and reflection.

Key words: source code comments, programming education, program comprehension, self-explanation, metacognition, qualitative systematic review.

1. Introduction

Learning to program remains a cognitively demanding process, particularly for novice learners, who must simultaneously coordinate syntax, semantics, control flow, abstraction, and problem decomposition while translating informal reasoning into executable representations (Navarro-Cota *et al.*, 2025; Shi, 2021). This complexity helps explain why introductory programming continues to be associated with persistent learning difficulties,

*, [0000-0002-8872-5721]Corresponding author.

especially in early stages of study, and why programming education research has increasingly emphasized instructional designs that reduce extraneous cognitive load and make code more interpretable (Tsai *et al.*, 2025; Shi, 2021).

A growing body of work in computing education has therefore focused on supports that improve program comprehension, debugging, and reflective reasoning, since these dimensions are foundational to successful programming performance (Yang *et al.*, 2024; Shi, 2021). In this context, source code comments occupy an interesting and potentially underexplored pedagogical space. In professional software development, comments are typically discussed as documentation artifacts that support readability, maintainability, collaboration, and long-term comprehension of software systems (Rani *et al.*, 2023; Figl *et al.*, 2025). However, in educational settings, comments may also function as instructional and metacognitive scaffolds: they can help learners articulate intent, externalize reasoning, connect individual lines of code to higher-level goals, and make problem-solving processes more visible both to themselves and to teachers.

This pedagogical interpretation is especially relevant in novice programming contexts, where learners often struggle to bridge the gap between natural-language intentions and formal code structures. When students are encouraged or required to annotate code while reading, writing, or debugging programs, comments can become a form of self-explanation—that is, an explicit attempt to explain what the code is doing, why it is doing so, and how it contributes to the overall solution. Self-explanation has long been recognized as a powerful learning mechanism because it promotes deeper processing, inferential integration, and metacognitive monitoring (Lombrozo, 2006). In programming education, these functions are particularly valuable because novice learners frequently produce syntactically plausible solutions without fully understanding the conceptual logic that underlies them.

The most direct line of evidence connecting source code comments to self-explanation in computing education comes from the work of Vieira and colleagues, who examined the use of *in-code comments* as a structured self-explanation strategy in computational science and engineering education. Their studies suggest that requiring students to write explanatory comments while interacting with worked examples can increase awareness of program structure, strengthen conceptual connections, and promote more reflective engagement with code (Vieira *et al.*, 2017). Although these studies were conducted in higher education and in computational science and engineering contexts, they provide an important starting point for understanding comments not merely as documentation, but as pedagogical devices embedded in instructional design.

More broadly, recent work in programming education reinforces the importance of instructional supports that explicitly target program comprehension, explanation, and self-regulation. For example, role-based and comprehension-oriented teaching approaches have been associated with stronger novice performance in program comprehension, debugging, and program explanation (Shi, 2021). Likewise, recent reviews of debugging instruction and related interventions indicate that programming success depends not only on writing code, but also on learning how to inspect, interpret, and reason about code in structured ways (Yang *et al.*, 2024). These findings strengthen the rationale for examin-

ing code comments as potential scaffolds for code reading, explanation, debugging, and reflective learning.

At the same time, the broader software engineering literature confirms that comments are highly relevant to code understanding, even outside explicitly educational settings. Recent evidence continues to show that comments influence perceived comprehensibility and understandability, while high-quality comments are closely linked to program comprehension and maintenance-related tasks (Rani *et al.*, 2023; Figl *et al.*, 2025). However, this literature is primarily concerned with software quality, comment consistency, and professional practice, rather than with how comments can be systematically integrated into teaching and learning. Consequently, educational decisions about commenting are often informed by general programming conventions or professional norms, rather than by a clearly synthesized pedagogical evidence base.

This creates a meaningful gap. While reviews exist on adjacent topics such as automated code assessment, code quality in education, debugging instruction, and broader programming pedagogy (Combéfis, 2022; Yang *et al.*, 2024; Ishaq *et al.*, 2024), there appears to be limited review work specifically centered on *source code comments as pedagogical tools* in programming education. The available studies are dispersed across different research traditions—including computer science education, computational science and engineering education, worked-example research, and code comprehension—and they differ substantially in their theoretical framing, learner populations, intervention formats, and outcome measures. As a result, it remains difficult to determine how source code comments have been operationalized educationally, what kinds of learning benefits have been reported, and where the current evidence remains limited.

A further complication is that the educational literature relevant to code comments is not organized as a single, clearly bounded research niche. Many potentially informative studies address adjacent constructs such as self-explanation, code explanation, worked examples, or scaffolded code comprehension without explicitly centering source code comments as the primary object of analysis. This fragmentation makes it difficult to derive pedagogically grounded conclusions from isolated studies alone and strengthens the need for a focused but conceptually sensitive review.

Against this background, the present study provides a structured, theory-informed qualitative systematic review with layered evidence, following a PRISMA-informed study identification and screening logic, of the literature on source code comments in programming education. The review aims to examine how source code comments have been conceptualized and implemented as pedagogical tools, what learning outcomes have been associated with their use, in which educational contexts they have been studied, and what methodological limitations and research gaps characterize the current evidence base.

Accordingly, the review addresses the following research questions:

- **RQ1.** How have source code comments been conceptualized and operationalized as pedagogical tools in programming education?
- **RQ2.** What learning outcomes have been associated with the use of source code comments in programming education?
- **RQ3.** In which educational contexts, learner profiles, and instructional designs have source code comments been studied?

- **RQ4.** What methodological patterns, limitations, and research gaps characterize the current evidence base?

This review makes four main contributions to the programming education literature. First, it offers a focused synthesis of an emerging and still underexplored topic within programming education. Second, it distinguishes between comments as software documentation artifacts and comments as pedagogical scaffolds for learning. Third, it organizes the available evidence into a functional framework that highlights recurring educational roles of comments, including self-explanation, code comprehension, debugging support, and metacognitive regulation. Fourth, it identifies key gaps in the literature—particularly the concentration of evidence in higher education and engineering-related contexts—and proposes directions for future research in more diverse educational settings, including secondary and vocational education.

The remainder of the article is organized as follows. Section 2 presents the conceptual background. Section 3 describes the review design and methodological procedures. Section 4 reports the descriptive results of the review, and Section 5 develops the thematic synthesis and discussion. Finally, Section 6 presents the conclusions and implications for future research and educational practice.

2. Conceptual Background

2.1. *From Software Documentation to Pedagogical Mediation*

In software engineering, source code comments are most commonly framed as documentation artifacts that support readability, maintainability, knowledge transfer, and long-term comprehension of software systems. Recent research continues to show that comments are relevant to how developers and readers perceive and understand code, while systematic review evidence indicates that comment quality has been studied through dimensions such as consistency, completeness, and readability (Rani *et al.*, 2023; Figl *et al.*, 2025). From this perspective, comments are primarily valued for their contribution to software quality and future code interpretation.

In educational settings, however, comments may serve a broader and more immediate function. Rather than merely documenting completed code, they can be used to mediate learning while code is being read, written, revised, or debugged. When students are encouraged or required to annotate their code, explain the purpose of a block, justify a control structure, or relate an implementation decision to a problem requirement, comments can support the externalization of reasoning and make thought processes more inspectable. This shift is pedagogically significant because it repositions comments from retrospective documentation to active learning support.

This distinction also implies that educational commenting does not need to follow exactly the same principles as professional commenting. In instructional contexts, comments that would be considered overly explicit or locally redundant in production code may still be valuable if they help novice learners articulate intent, trace logic, and connect low-level

syntax to higher-level algorithmic goals. Consequently, pedagogical commenting should be interpreted not only through software engineering criteria, but also through learning-oriented constructs such as self-explanation, scaffolding, and metacognitive regulation.

2.2. *Comments as a Form of Self-Explanation*

A particularly useful lens for understanding the pedagogical role of source code comments is self-explanation. Broadly speaking, self-explanation refers to the process by which learners generate explanations to themselves while studying examples, interpreting information, or solving problems. This process has long been associated with deeper learning because it prompts learners to infer relations, integrate prior knowledge, and monitor their own understanding (Lombrozo, 2006).

In programming education, writing comments can function as an externalized form of self-explanation. Instead of merely producing executable code, learners are invited to verbalize what a line or block is doing, why it is needed, and how it contributes to the intended solution. This is especially relevant for novices, who often focus on isolated commands or syntactic correctness without fully grasping the conceptual structure of the program. By asking learners to comment their code, instruction can encourage a shift from surface-level code production to more deliberate reasoning about goals, mechanisms, and relationships within the program.

The clearest empirical articulation of this idea comes from Vieira and colleagues, who investigated *in-code comments* as a structured self-explanation strategy in computational science and engineering education. Their work suggests that comment writing can support conceptual linking, reflective engagement, and stronger awareness of how computational procedures relate to disciplinary problems (Vieira *et al.*, 2017). Although these studies were conducted in higher education and in computational science and engineering settings, they remain highly relevant because they frame comments explicitly as pedagogical instruments rather than as mere code annotations.

2.3. *Comments as Scaffolds for Code Comprehension and Debugging*

Programming education research increasingly emphasizes that success in learning to program depends not only on code production, but also on code comprehension, debugging, and explanation. Recent work has shown that instructional approaches explicitly designed to support novice comprehension can improve learners' performance in understanding, explaining, and debugging programs (Shi, 2021). Likewise, a recent systematic review of debugging interventions highlights the importance of pedagogical strategies that help learners inspect, interpret, and reason about code in structured ways (Yang *et al.*, 2024).

Within this broader context, source code comments can be understood as scaffolds. In worked examples, comments may guide attention to key operations, decision points, or data transformations. In debugging activities, comments may help learners compare intended behavior with observed behavior, making faults easier to identify conceptually. In code reading tasks, comments can reduce ambiguity and clarify how local instructions

contribute to global program logic. Comments may therefore support comprehension not simply by adding textual information, but by shaping how learners navigate, interpret, and reflect upon code.

This scaffolding role is especially important in introductory contexts, where learners frequently experience difficulty coordinating multiple levels of representation at once, including natural-language problem statements, algorithmic ideas, and formal code structures. Research on individual differences in programming further suggests that programming performance is influenced by a wide range of learner-related factors, which reinforces the need for supports that make reasoning more visible and manageable for diverse student profiles (Navarro-Cota *et al.*, 2025). From this perspective, comments may serve as flexible scaffolds that can be adapted to learner expertise and instructional purpose.

2.4. *Metacognition, Reflection, and Self-Regulation*

The educational value of comments is also closely connected to metacognition and self-regulated learning. Learning to program involves planning, monitoring, error detection, evaluation of partial solutions, and revision of strategy. These processes are central to successful performance in computing education and are increasingly recognized in contemporary reviews and empirical studies (Ishaq *et al.*, 2024; Yang *et al.*, 2024). When learners are prompted to comment on what they are doing and why, they may become more aware of the assumptions, intentions, and uncertainties embedded in their code.

Comments can therefore support reflection in at least two ways. First, they can help students monitor their own understanding while programming, especially when comments are written iteratively rather than appended after completion. Second, they can provide teachers with insight into student reasoning that executable output alone does not reveal. In this sense, comments may act as traceable indicators of conceptual understanding, misconceptions, or incomplete reasoning.

This perspective aligns with recent discussions in programming education that emphasize the importance of learning designs capable of supporting not only performance outcomes, but also strategic, cognitive, and motivational dimensions of learning (Ishaq *et al.*, 2024). Although comments are not a complete solution to these challenges, they may represent a low-cost and pedagogically versatile mechanism for fostering more explicit, reflective, and self-regulated engagement with code.

2.5. *Why a Focused Review Is Needed*

Taken together, the literature suggests that source code comments can plausibly serve multiple pedagogical functions: as vehicles for self-explanation, as scaffolds for code comprehension, as supports for debugging, and as prompts for reflection and self-regulation. At the same time, the evidence base appears fragmented across different research traditions, including computer science education, computational science and engineering education, software engineering, and adjacent work on programming pedagogy.

This fragmentation creates a need for a focused synthesis. The software engineering literature provides useful evidence about the role of comments in comprehensibility and

quality, but it is not primarily concerned with teaching and learning (Rani *et al.*, 2023; Figl *et al.*, 2025). Conversely, the programming education literature contains relevant findings on comprehension, debugging, and self-explanation, yet studies that place source code comments at the center of pedagogical analysis remain comparatively limited and dispersed (Vieira *et al.*, 2017; Shi, 2021; Yang *et al.*, 2024). A focused qualitative systematic review of source code comments in programming education is therefore warranted in order to clarify how comments have been conceptualized educationally, what kinds of learning outcomes have been associated with them, and which gaps should guide future research.

3. Methodology

This study adopted a structured, theory-informed qualitative systematic review with layered evidence, following a PRISMA-informed identification and screening logic (Page *et al.*, 2021), to identify, screen, appraise, and synthesize literature on the pedagogical role of source code comments in programming education. The review combined database searches in Scopus and ERIC with a transparent multi-stage workflow that included API-based retrieval, rule-based pre-filtering, deduplication, heuristic relevance classification, and manual title/abstract and full-text screening. This design was selected because the topic is relatively narrow in its direct form—that is, studies explicitly examining source code comments in educational programming contexts—but is also closely connected to adjacent pedagogical constructs such as self-explanation, code comprehension, worked examples, metacognitive support, and instructional scaffolding. Accordingly, the review retained a broader but theoretically coherent evidence base, allowing code comments to be interpreted not only as documentation artifacts but also as explanation-centered learning supports. The review followed a PRISMA-informed logic for study identification, screening, eligibility assessment, and inclusion, while adopting a layered-evidence qualitative synthesis rather than a conventional homogeneous systematic literature review (SLR) corpus or a meta-analytic design.

3.1. *Review design and scope*

The review was designed as a PRISMA-informed qualitative systematic review with layered evidence, focused on how source code comments have been conceptualized, operationalized, and studied as pedagogical tools in programming education. Because the directly comment-focused educational literature remains relatively limited, the review explicitly distinguishes between evidence layers within a broader retained corpus. The primary synthesis corpus comprises studies mobilized directly in the main thematic synthesis and includes both direct comment-focused studies and adjacent mechanism- or methodological-supporting studies. In addition, a smaller set of supporting/contextual studies was retained to provide theoretical or interpretive background without being treated as part of the primary empirical synthesis. This layered design was maintained throughout screening, coding, appraisal, and synthesis. Importantly, comment-specific

conclusions are grounded primarily in the direct comment-focused studies within the primary synthesis corpus, whereas adjacent studies are used to clarify plausible pedagogical mechanisms and related instructional processes rather than to establish comment-specific effects directly.

3.2. *Review objective and research questions*

The main objective of this review was to investigate how source code comments have been conceptualized and used in programming education, and to examine whether the available evidence supports their interpretation as pedagogically meaningful tools rather than merely stylistic or documentation-oriented programming conventions.

To address this objective, the review was guided by the following research questions:

- **RQ1.** How have source code comments been conceptualized and operationalized as pedagogical tools in programming education?
- **RQ2.** What learning outcomes have been associated with the use of source code comments in programming education?
- **RQ3.** In which educational contexts, learner profiles, and instructional designs have source code comments been studied?
- **RQ4.** What methodological patterns, limitations, and research gaps characterize the current evidence base?

These research questions preserve the review's core focus on source code comments while accommodating the broader explanation-centered perspective that emerged during screening and synthesis. Although source code comments remained the primary object of interest, the directly comment-focused educational literature proved relatively limited. The final evidence base was therefore structured in layers: (i) studies directly centered on code comments in educational contexts, and (ii) studies examining closely related explanation-centered practices that help clarify the pedagogical mechanisms of commenting. Supporting sources used only for historical or theoretical contextualization were retained separately within the broader interpretive corpus and were not treated as part of the primary empirical synthesis; rather, they were used only to provide bounded theoretical or interpretive background where needed.

3.3. *Search strategy and database-level retrieval audit trail*

The search strategy was implemented as a set of complementary query families rather than as a single query string in order to increase recall while preserving conceptual focus. An initial Scopus search configuration was tested and then refined into a safer and more robust final version without proximity operators, while ERIC queries were designed as concise Boolean combinations compatible with the ERIC API. To improve auditability, the revised manuscript reports database-level retrieval and screening counts explicitly and summarizes the retrieval pathway in Table 1.

For the final Scopus retrieval stage, the search was implemented through multiple concrete query variants (reported in Appendix A) grouped into three comple-

mentary conceptual families: (i) direct matches for source code comments in explicitly educational programming contexts; (ii) studies linking comments or explanation-related constructs to code/program comprehension and pedagogical mechanisms; and (iii) broader pedagogical studies involving commenting practices, learners, and programming/computer science contexts. For ERIC, three complementary query families were used: `A_code_comments_prog_edu` (code comments AND programming education), `B_in_code_comments_cs_edu` (in code comments AND computer science education), and `C_self_explanation_programming` (self explanation AND programming).

This query design intentionally combined direct terminology (e.g., *source code comments*, *in-code comments*, *commenting code*) with adjacent pedagogical constructs (e.g., *self-explanation*, *code comprehension*, *worked examples*, *scaffolding*, *metacognition*) in order to capture both direct evidence and mechanism-relevant supporting literature. However, because this broader conceptual strategy increased overlap across query families and between databases, retrieval was followed by staged deduplication and conservative manual screening rather than by direct automatic inclusion.

Table 1 provides the database-level audit trail used in the revised protocol. The table distinguishes raw retrieval counts by database and conceptual query family from the progressively refined screening stages that led to the globally deduplicated pool and, ultimately, to the advanced-screening candidate set. Importantly, because multiple query variants within and across families intentionally overlapped, family-level counts should not be interpreted as additive totals for the final corpus. Instead, they document coverage breadth and retrieval provenance. The analytically meaningful consolidated counts are the globally deduplicated pool of 50 unique records, the 36 records advanced to detailed eligibility assessment, and the final resolution of those 36 records into 18 studies mobilized in the primary synthesis layer (including direct and adjacent evidence), 7 supporting/contextual studies retained for interpretive or theoretical background, and 11 records excluded at the advanced screening stage.

The revised audit trail therefore serves two purposes: first, it documents how the broader layered-evidence strategy was operationalized at the database and query-family level; second, it clarifies that the review was intentionally designed to retrieve both direct comment-centered studies and adjacent explanation-centered studies, while preserving a bounded interpretive distinction between them during synthesis.

3.4. *Eligibility criteria*

Eligibility criteria were defined to ensure that retained studies were educationally relevant and sufficiently aligned with the review focus.

3.4.1. *Inclusion criteria*

Studies were considered eligible for inclusion if they satisfied the core educational relevance requirements of the review and aligned with the following criteria:

Table 1

Database-level retrieval audit trail by source and conceptual query family. Family-level counts document retrieval breadth and provenance, but they are not additive for the final corpus because multiple query families were intentionally overlapping within and across databases. The analytically consolidated counts are shown in the final summary row.

Database	Query family	Raw hits saved	After within-database dedup.	Retained for cross-database merge	Notes
Scopus	A_core_direct	14	13	13	Direct comment-focused educational programming retrieval
Scopus	B_self_explanation	19	16	16	Comments/self-explanation/comprehension mechanism-linked retrieval
Scopus	C_pedagogical_broad	25	14	14	Broader pedagogical commenting/learner/programming retrieval
ERIC	A_code_comments_prog_edu	222	218	5	Direct Boolean query; only high-priority candidates were forwarded after ERIC post-filtering
ERIC	B_in_code_comments_cs_edu	1000[a]	930	0	Very broad query; no records were forwarded after ERIC post-filtering
ERIC	C_self_explanation_programming	160	149	5	Adjacent pedagogical query targeting self-explanation in programming

Consolidated audit trail. Across all final retrieval queries, 58 Scopus records and 1382 ERIC records were saved by the scripts (Scopus = 43 after within-database deduplication; ERIC = 1297 after within-database deduplication). The ERIC corpus was then reduced by a conservative post-filtering step to 10 high-priority candidates (1 strong keep + 9 possible keep), which were merged with the 43 Scopus records, yielding 53 records for cross-database merging. Global deduplication reduced this set to 50 unique records (42 traceable to Scopus and 8 traceable to ERIC after cross-database deduplication). These 50 records were then screened at title/abstract level, producing 36 advanced-screening candidates for detailed eligibility assessment.

Note. The Scopus query-family counts are fully reproducible from the final Scopus API retrieval files (`A_core_direct`, `B_self_explanation`, `C_pedagogical_broad`), which together yielded 58 saved records and 43 records after within-database deduplication. The ERIC query-family counts are based on the final ERIC API retrieval files (`A_code_comments_prog_edu`, `B_in_code_comments_cs_edu`, `C_self_explanation_programming`), which together yielded 1382 saved records and 1297 records after within-database deduplication. Because the ERIC corpus was substantially broader and noisier than the Scopus corpus, an explicit conservative post-filtering step was applied before cross-database merging; this reduced the ERIC set to 10 manually reviewable candidates. Family-level counts should therefore be interpreted as *retrieval provenance* rather than as additive counts for the final corpus. The analytically meaningful consolidated counts for the review are the globally deduplicated pool of 50 records and the subsequent screening stages.

[a] The query `B_in_code_comments_cs_edu` returned 2008 records according to the ERIC API query summary, but the script saved the first 1000 records in the raw export file used for downstream processing.

- They were situated in a **formal educational context**, such as programming courses, computing education, computer science education, or computational science and engineering education;
- They addressed at least one of the following constructs in a manner relevant to the review: **source code comments**, **student-generated code explanations**, **self-explanation in programming**, **code comprehension explanations**, or **explanation-centered instructional scaffolds**;
- They focused on **programming learning**, **code comprehension**, **novice programming**, or closely related educational outcomes;
- They were available in **English**, which was used as the review language for screening and synthesis;

- They provided sufficient accessible information (preferably full text) to support reliable screening and qualitative interpretation.

3.4.2. *Exclusion criteria*

Studies were excluded if one or more of the following conditions applied:

- The study was primarily situated in **software engineering**, **software maintenance**, or **automatic code comment generation/classification** without a clear educational context;
- The publication was a **book**, **book chapter**, editorial, or other source type outside the intended evidence base, unless it was retained explicitly as a contextual or theoretical source and not as an included study in the main synthesis;
- The study was a duplicate record retrieved from multiple databases;
- The study was clearly misaligned with the review objective after title and abstract screening;
- Only limited metadata or abstract information was available and the full text could not be obtained after reasonable retrieval attempts, preventing reliable coding as a primary-study candidate or as a supporting/contextual source within the layered review design.

3.5. *Screening and study selection*

After retrieval, records were processed through a computationally assisted pre-screening pipeline implemented in Python. This stage was not used as a substitute for scholarly judgment, but as a transparent and reproducible support mechanism for reducing noise, identifying duplicate records, and prioritizing potentially relevant studies.

First, an initial noise-filtering step flagged clearly out-of-scope records based on title-level indicators associated with topics not aligned with the pedagogical focus of this review, such as automatic comment generation, code summarization, technical debt, repository mining, malware analysis, and large-language-model-oriented code analysis without a clear educational context. These records remained auditable in the intermediate screening files.

Second, records were deduplicated within each database and then across databases using a staged procedure: exact DOI matching where available, exact title–year matching, and a conservative fuzzy title-similarity pass for same-year records using a high similarity threshold. This procedure was intended to reduce duplicate retrievals while minimizing false merges across conceptually adjacent but distinct publications.

Third, retained records were assigned a preliminary relevance status using rule-based pattern matching and heuristic scoring. This classification distinguished between likely direct studies focused on source code comments in programming education, likely adjacent studies relevant through constructs such as self-explanation, code comprehension, worked examples, metacognitive prompting, or instructional scaffolding, and likely peripheral studies that lacked sufficient pedagogical alignment.

Only after this computationally assisted prioritization were records manually screened at title/abstract level and, where warranted, at full-text or equivalent advanced eligibil-

ity level. Final inclusion decisions were based on human judgment informed by the reproducible audit trail rather than on automated classification alone. Ambiguous cases, particularly studies addressing self-explanation, reflective annotation, or explanatory programming practices without explicit emphasis on source code comments, were reviewed conservatively and retained only when they provided conceptually relevant insight into the pedagogical mechanisms under study.

After global cross-database deduplication, 50 unique records were retained for title and abstract screening. At this stage, 14 records were excluded as clearly out of scope or insufficiently aligned with the review objective. The remaining 36 records advanced to detailed eligibility assessment.

At the advanced screening stage, the corpus was resolved into three outcome groups: 18 studies mobilized in the primary synthesis layer (including direct and adjacent evidence), 7 supporting/contextual studies retained only for bounded interpretive or theoretical support, and 11 studies excluded at the advanced screening stage. This disposition is summarized in Figure 1. The supporting/contextual studies were retained separately and were not treated as part of the primary synthesis corpus.

The 18 primary studies were not treated as a homogeneous body of direct evidence. Instead, they were further classified according to evidence layer. Studies in which source code comments, in-code comments, commenting behaviour, comment quality, or comment assessment were central to the intervention, analysis, or assessment were classified as direct comment-focused studies. Studies centred on self-explanation, code explanation, worked examples, subgoal learning, scaffolded code comprehension, or related explanatory practices were classified as adjacent mechanism-supporting or methodological-supporting studies. This layered classification was maintained throughout data extraction, appraisal, and synthesis.

The resulting retained interpretive corpus therefore comprised 25 studies in total: 18 studies retained in the primary synthesis layer and 7 supporting/contextual studies retained separately for bounded interpretive or theoretical support. Comment-specific claims were grounded primarily in the direct comment-focused studies within the primary synthesis corpus, whereas adjacent primary studies were used to support mechanism-oriented interpretation rather than as equivalent direct evidence of comment-specific effects.

Screening and inclusion decisions were conducted collaboratively by the authors using the structured audit trail generated by the retrieval and pre-screening pipeline. Title/abstract and advanced-stage decisions were discussed iteratively, with potentially ambiguous cases resolved through author consensus. Because the review was conducted as a small-scale two-author synthesis rather than as a formally dual-independent screening protocol, no inter-rater agreement statistics were calculated.

3.6. *Data extraction and coding framework*

Data extraction was conducted manually using a structured screening and synthesis framework developed for this review. For each included study, the following information was recorded whenever available:

- bibliographic information (title, year, venue, and authorship context);
- educational setting (e.g., introductory programming, CS1, computational science and engineering education);
- approximate participant profile (e.g., novice students, undergraduate learners);
- evidence-layer classification (direct comment-focused, adjacent mechanism-supporting, adjacent methodological-supporting, or supporting/contextual);
- focal pedagogical construct (e.g., code comments, self-explanation, code explanation, worked examples, subgoal learning);
- broad methodological design (e.g., qualitative, quantitative, mixed, exploratory, intervention-based, assessment-oriented);
- main findings relevant to the pedagogical role of comments or explanation-centered learning.

A structured extraction sheet was used to support transparent coding and cross-study comparison. The extraction framework also provided the basis for the definitive included-studies table reported in Table 2, where each study identifier is mapped to one unique bibliographic source and classified according to its evidence layer within the primary synthesis corpus.

Data extraction and evidence-layer coding were likewise conducted collaboratively by the two authors using the structured extraction sheet. When uncertainty arose regarding study classification, synthesis role, or evidentiary layer, the relevant study was re-examined and disagreements were resolved through discussion until a shared interpretation was reached.

3.7. *Appraisal, evidence layering, and interpretive weighting*

To improve transparency and align the appraisal with the layered-evidence review design, the retained literature was appraised using a two-part procedure that explicitly distinguished methodological confidence from direct relevance to the primary review object (source code comments in programming education). This distinction was introduced because the final retained corpus includes both studies directly focused on source code comments and studies that do not primarily investigate comments as software artifacts, but nevertheless provide relevant evidence about closely related pedagogical mechanisms such as self-explanation, code explanation, worked examples, and scaffolded code comprehension.

Methodological confidence was assessed using the same structured five-criterion appraisal framework applied during the revised screening and extraction process: (C1) clarity of educational context and participants; (C2) transparency of the instructional task, intervention, or pedagogical use of comments or explanation-related scaffolds; (C3) adequacy of the evidence source and data collection; (C4) analytical clarity and alignment between evidence and claims; and (C5) direct relevance to the review question. Each criterion was rated on a 0–2 scale (0 = weak or absent, 1 = partial, 2 = clear or adequate). As in the previous version of the review protocol, these ratings were used as a structured interpretive aid rather than as a strict exclusion filter, given the relatively small and methodologically diverse body of retained studies.

Table 2
Overview of the primary empirical studies mobilized in the main synthesis (S1–S18). The table maps each study identifier to a unique bibliographic source and distinguishes between direct comment-focused studies and adjacent mechanism- or methodological-supporting studies. This layered classification was used to ensure that comment-specific conclusions were grounded primarily in direct comment-focused evidence, while adjacent primary studies were used to clarify plausible pedagogical mechanisms and related instructional processes.

Study ID	Citation	Year	Main focus	Evidence layer	Primary synthesis contribution
S1	Vieira *et al.*, 2016	2016	In-code comments as a self-explanation strategy for computational science education	Direct comment-focused	Comments as pedagogical self-explanation
S2	Vieira *et al.*, 2017	2017	Writing in-code comments to self-explain in computational science and engineering education	Direct comment-focused	Comment writing as explanation and conceptual linking
S3	Kerschbaumer *et al.*, 2025	2025	Student commenting behaviour and academic success in CS1	Direct comment-focused	Commenting behaviour and performance-related associations
S4	Sukamto *et al.*, 2023	2023	Code comment assessment in basic programming using online judge systems	Direct comment-focused	Educational implementation and comment assessment
S5	Vieira *et al.*, 2019	2019	Student explanations in computational science and engineering education	Adjacent mechanism-supporting	Bridge evidence for explanation-centered learning
S6	Garces *et al.*, 2023	2023	Active exploration of programming worked examples	Adjacent mechanism-supporting	Worked examples and guided comprehension
S7	Tamang *et al.*, 2021	2021	Free self-explanations vs. Socratic tutoring for source code comprehension	Adjacent mechanism-supporting	Self-explanation and code comprehension
S8	Oli *et al.*, 2023	2023	Scaffolded self-explanations for improving code comprehension	Adjacent mechanism-supporting	Scaffolded explanation and comprehension support
S9	Vihavainen *et al.*, 2015	2015	Benefits of self-explanation in introductory programming	Adjacent mechanism-supporting	Introductory programming and self-explanation
S10	Sudol-DeLyser, 2015	2015	Self-explanation in code production and abstraction	Adjacent mechanism-supporting	Process-oriented explanation and abstraction
S11	Alhassan, 2017	2017	Self-explanation with worked examples for programming skills	Adjacent mechanism-supporting	Worked examples and programming skill development
S12	Chapagain and Rus, 2025	2025	Automated assessment of student self-explanations in code comprehension	Adjacent methodological-supporting	Scalable assessment of explanatory artifacts
S13	Lekshmi-Narayanan and Brusilovsky, 2024	2024	Evaluating correctness of student code explanations	Adjacent methodological-supporting	Explanation assessment and interpretive accuracy
S14	van der Werf *et al.*, 2022	2022	Novice students' code explanations in Python	Adjacent mechanism-supporting	Novice code explanation practices
S15	Sandoval-Medina *et al.*, 2024	2024	Self-explanation and cognitive load in basic programming	Adjacent mechanism-supporting	Cognitive load and explanation-centered scaffolding
S16	Margulieux and Catrambone, 2017	2017	Subgoal learning and self-explanation in programming education	Adjacent mechanism-supporting	Structured scaffolding and explanatory decomposition
S17	Lee and Hong, 2017	2017	Self-explanation vs. reading questions and answers in programming learning	Adjacent mechanism-supporting	Comparative evidence for self-explanation-oriented learning
S18	Rani *et al.*, 2023	2023	Automated assessment of code comments using semantic similarity methods	Direct comment-focused	Modern automated comment-assessment evidence

However, in response to the need for a more auditable and conceptually bounded synthesis, the final interpretive weighting procedure did not rely on aggregate scores alone. Instead, the synthesis explicitly introduced an *evidence-layer classification* for the empirical studies most directly mobilized in the main synthesis (S1–S18). As shown in Table 2, these studies were not treated as a homogeneous primary evidence corpus. Rather, they were separated into two higher-order analytical layers: *direct comment-focused studies*, which directly examined source code comments or comment-assessment practices in educational settings, and *adjacent supporting studies*, which addressed closely related

explanation-centered instructional mechanisms or methodological proxies in programming education without treating source code comments as the primary object of analysis. Within the adjacent layer, the synthesis further distinguished between *mechanism-supporting* and *methodological-supporting* studies where this distinction was useful for interpretive weighting.

This layered treatment was adopted to align the synthesis more closely with the actual evidentiary structure of the retained corpus. Specifically, comment-specific conclusions (e.g., about the pedagogical role of in-code comments, comment-writing behaviour, or comment-assessment practices) were grounded primarily in the direct comment-focused studies. By contrast, adjacent studies on self-explanation, code explanation, worked examples, subgoal learning, and related scaffolds were used to support interpretation of plausible pedagogical mechanisms, learner processes, and explanatory functions that may help explain why commenting can matter educationally, but they were not treated as direct evidence about source code comments themselves.

Accordingly, Table 3 reports the final appraisal and relevance-weighting summary for the retained empirical studies most directly mobilized in the synthesis. Rather than collapsing all dimensions into a single quality label, the table reports (i) the study focus, (ii) the evidence layer, (iii) the methodological confidence judgment, and (iv) the specific way in which each study was used in the synthesis. This structure makes explicit that a study may be methodologically strong while remaining only indirectly relevant to the review's primary object, and it prevents adjacent primary studies from being interpreted as equivalent to direct comment-focused evidence.

No study was excluded solely on appraisal grounds. Instead, methodological confidence and evidence layer were jointly used to calibrate interpretive emphasis during synthesis. Higher-confidence studies received greater interpretive weight than moderate-confidence studies, and direct comment-focused studies received greater weight for claims specifically about source code comments. In practice, this means that adjacent studies could strengthen or contextualize explanation-centered interpretations, but they could not, by themselves, justify comment-specific claims. This approach was designed to preserve the conceptual contribution of the review—namely, that comments may function pedagogically as explanation-oriented scaffolds—while ensuring that the strength of that claim remains proportionate to the directness of the available evidence.

3.8. *Thematic synthesis procedure*

Because the evidence base was heterogeneous in terminology, design, and outcome measures, a **qualitative thematic synthesis** was used instead of a statistical aggregation or meta-analysis. The synthesis was therefore organized around conceptually coherent themes rather than around uniform effect sizes or directly comparable intervention metrics.

The thematic synthesis followed an iterative qualitative coding procedure. First, extracted study descriptors and findings were read in full and assigned **initial descriptive codes** capturing the pedagogical function of source code comments (or, for adjacent studies, the explanation-centered mechanism most closely related to comment use). Second,

Table 3
Methodological confidence and relevance-weighting summary for the primary empirical studies mobilized in the main synthesis (S1–S18). The table separates evidence layer from methodological confidence and indicates how each study was used in the synthesis.

Study ID	Study focus	Evidence layer	Methodological confidence	Use in synthesis
S1	Vieira et al. (2016): in-code comments as self-explanation in computational science education	Direct comment-focused	Higher	Comment-specific claims and pedagogical interpretation
S2	Vieira et al. (2017): writing in-code comments to self-explain in computational science and engineering education	Direct comment-focused	Higher	Comment-specific claims and self-explanation framing
S3	Student commenting behaviour and academic success in CS1	Direct comment-focused	Higher	Comment-specific claims and performance-related interpretation
S4	Code comment assessment in basic programming using online judge systems	Direct comment-focused	Higher	Comment-specific claims and assessment implementation
S5	Student explanations in computational science and engineering education	Adjacent mechanism-supporting	Higher	Bridge evidence for explanation-centered mechanisms
S6	Active exploration of programming worked examples	Adjacent mechanism-supporting	Higher	Mechanism-level interpretation of guided code understanding
S7	Free self-explanations vs. Socratic tutoring for source code comprehension	Adjacent mechanism-supporting	Higher	Mechanism-level interpretation of code comprehension scaffolding
S8	Scaffolded self-explanations for improving code comprehension	Adjacent mechanism-supporting	Higher	Mechanism-level interpretation of scaffolded explanatory practice
S9	Benefits of self-explanation in introductory programming	Adjacent mechanism-supporting	Higher	Contextual evidence for self-explanation in novice programming
S10	Self-explanation in code production and abstraction	Adjacent mechanism-supporting	Moderate	Used cautiously for process-oriented explanation mechanisms
S11	Self-explanation with worked examples for programming skills	Adjacent mechanism-supporting	Higher	Contextual evidence for explanation-supported skill development
S12	Automated assessment of student self-explanations in code comprehension	Adjacent methodological-supporting	Higher	Methodological extension for explanation assessment
S13	Evaluating correctness of student code explanations	Adjacent methodological-supporting	Higher	Methodological extension for assessing explanation quality
S14	Novice students' code explanations in Python	Adjacent mechanism-supporting	Higher	Contextual evidence for novice explanation practices
S15	Self-explanation and cognitive load in basic programming	Adjacent mechanism-supporting	Moderate	Used cautiously for cognitive-load-related interpretation
S16	Subgoal learning and self-explanation in programming education	Adjacent mechanism-supporting	Higher	Contextual evidence for structured scaffolding and decomposition
S17	Self-explanation vs. reading questions and answers in programming learning	Adjacent mechanism-supporting	Higher	Comparative mechanism evidence for explanation-centered learning
S18	Automated assessment of code comments using semantic similarity methods	Direct comment-focused	Higher	Comment-specific claims related to automated comment assessment

Note. Methodological confidence was judged separately from direct relevance to the review's primary object, namely source code comments in programming education. The evidence-layer column indicates whether a study directly examined comments or was used as adjacent mechanism- or methodological-supporting evidence. Comment-specific conclusions were grounded primarily in the direct comment-focused studies. Adjacent studies were used to interpret plausible pedagogical mechanisms, learner processes, and assessment possibilities, but were not treated as direct evidence of comment-specific effects. No study was excluded solely on appraisal grounds; instead, methodological confidence and evidence layer jointly informed interpretive weighting in the synthesis.

these codes were compared across studies and grouped through **constant comparison** into broader thematic clusters. Third, provisional themes were refined by revisiting the original studies to verify that each theme remained grounded in the evidence.

To avoid overextension from adjacent literature, **theme derivation was anchored**

first in the direct comment-focused studies within the primary synthesis corpus, and only then supplemented by adjacent primary studies where these provided plausible mechanism-oriented or methodological interpretive support. This sequencing ensured that the final thematic structure remained evidence-led rather than theory-led.

The final synthesis converged on four main thematic strands:

1. **Direct pedagogical use of code comments**;
2. **Self-explanation as an underlying learning mechanism**;
3. **Student-generated code explanations and their assessment**;
4. **Worked examples, scaffolds, and broader explanation-centered programming pedagogy**.

This thematic organization reflects both the direct comment-focused evidence and the broader explanation-centered literature that helps interpret why comments may support programming learning. However, the strongest claims in this review are grounded primarily in the direct comment-focused studies within the primary synthesis corpus, whereas adjacent studies are used to support mechanism-oriented interpretation and contextualization rather than as direct evidence of comment-specific pedagogical effects.

3.9. *Methodological limitations of the review process*

Several methodological limitations of the review process should be acknowledged. First, the topic is still relatively specialized and fragmented, which means that a narrowly comment-only search would likely have produced an evidence base too small to support a meaningful pedagogical synthesis. The broader inclusion of adjacent self-explanation and code explanation studies was therefore a deliberate interpretive choice. While this strengthens theoretical coherence, it also means that not all conclusions rely exclusively on direct comment-focused evidence.

Second, a small number of potentially relevant studies could only be accessed at the abstract level despite reasonable retrieval efforts. To preserve the reliability of the synthesis, these studies were not treated as included studies in the main synthesis unless their accessible information was sufficient to support confident classification. This conservative approach improves internal consistency but may have excluded some relevant recent work.

Third, the included studies vary substantially in terminology, instructional setting, and methodological design. As a result, the review is best understood as a structured qualitative synthesis of an emerging literature rather than as a definitive quantitative summary of effect sizes. Nevertheless, this approach is appropriate for the present objective, which is to clarify how code comments can be understood pedagogically and how they relate to the broader literature on explanation-centered learning in programming education.

To preserve conceptual clarity in the synthesis, the retained literature was organized into a layered architecture. The primary synthesis corpus comprised 18 empirical studies mobilized directly in the main synthesis. These were further divided into direct comment-focused studies, defined as studies in which source code comments, in-code comments,

comment writing, comment quality, or comment-related pedagogical uses were a central object of analysis or intervention, and adjacent mechanism- or methodological-supporting studies, which examined closely related explanation-centered constructs such as self-explanation, worked examples, scaffolded code comprehension, or student-generated code explanations without focusing primarily on source code comments themselves. In addition, 7 supporting/contextual studies were retained separately within the broader interpretive corpus because they provided relevant theoretical, historical, or pedagogical context. These supporting/contextual studies informed interpretation, but they were not treated as part of the primary empirical synthesis and were not used as equivalent evidence when formulating comment-specific conclusions.

4. Results

This section presents the descriptive results of the review. It first summarizes the screening outcomes and the composition of the final evidence base, and then reports the main descriptive patterns observed across the included studies in terms of publication characteristics, educational settings, learner populations, methodological approaches, and reported outcomes. As outlined in Section 3, the review distinguishes between *primary studies*, which form the core evidence base for the synthesis, and *supporting studies*, which are retained for contextual and interpretive purposes but are not treated as equally weighted primary evidence.

The final retained corpus was organized in a layered manner rather than treated as a single homogeneous evidence base. Within the primary synthesis corpus, two analytically distinct empirical layers were distinguished: (i) direct comment-focused studies, which explicitly addressed source code comments or commenting practices in educational programming contexts; and (ii) adjacent mechanism- or methodological-supporting studies, which did not always focus on comments as the primary object but examined closely related explanatory or instructional mechanisms such as self-explanation, code comprehension, worked examples, or scaffolding. Supporting/contextual studies were retained separately to strengthen theoretical, historical, or interpretive framing, but were not treated as part of the primary empirical synthesis.

4.1. *Screening outcomes and corpus composition*

The retained interpretive corpus comprised 25 studies, including 18 primary studies mobilized in the main synthesis and 7 supporting/contextual studies retained separately for bounded interpretive or theoretical support. Within the 18 primary studies, the synthesis distinguished between direct comment-focused studies and adjacent mechanism-supporting or methodological-supporting studies. This layered structure was maintained throughout the descriptive mapping and thematic interpretation in order to ensure that comment-specific claims remained grounded primarily in the most directly relevant evidence.

These studies emerged from a multi-stage screening process applied to 50 globally deduplicated records, of which 36 advanced to detailed eligibility assessment after title and abstract screening. At the advanced stage, the candidate corpus was resolved into three outcome groups: 18 primary studies retained for the main synthesis, 7 supporting/contextual studies retained separately for bounded interpretive or theoretical support, and 11 records excluded from the final retained corpus. Figure 1 summarizes this selection process.

The descriptive overview that follows focuses first on the primary synthesis corpus and then situates the supporting/contextual studies in relation to the broader interpretive framing of the review.

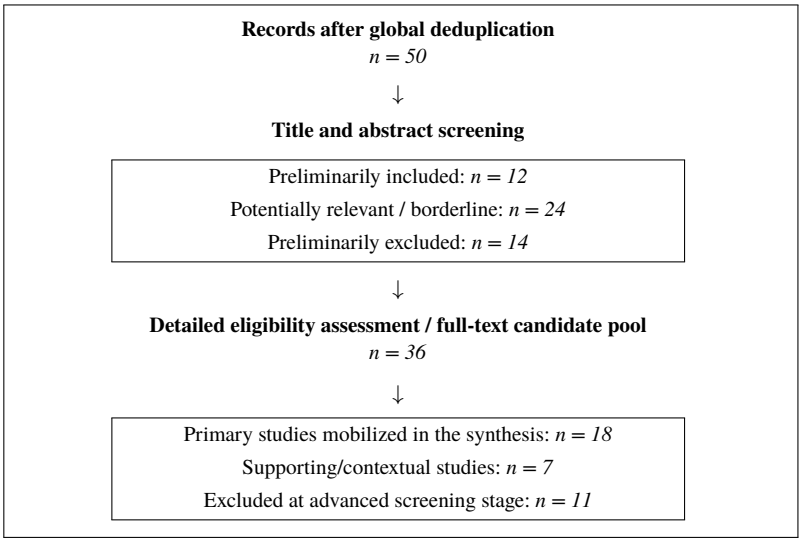

Fig. 1. PRISMA-informed study selection flow from global deduplication to the final resolution of the advanced-screening corpus into primary synthesis studies, supporting/contextual studies, and advanced-stage exclusions.

The 18 primary studies were further organized according to evidence layer. A subset of these studies directly addressed source code comments, in-code comments, commenting behaviour, comment quality, or comment assessment in educational programming contexts. These studies provided the principal evidentiary basis for comment-specific conclusions. The remaining primary studies addressed adjacent explanation-centered constructs, including self-explanation, student-generated code explanations, worked examples, subgoal learning, scaffolded code comprehension, and assessment of explanatory artifacts. These adjacent studies were retained because they clarify plausible mechanisms through which commenting may function pedagogically, but they were not treated as equivalent to direct comment-focused evidence.

In addition to the 18 primary studies, 7 supporting/contextual studies were retained because they provided relevant theoretical, historical, or pedagogical context. These supporting studies informed interpretation but were retained outside the primary synthesis

corpus and were therefore not used as equivalent evidence for comment-specific conclusions. Accordingly, the main structured synthesis was anchored in the 18 primary studies reported in Table 2, while the supporting/contextual studies informed broader interpretive framing without being treated as equivalent primary evidence.

This final configuration reflects the methodological strategy adopted in the review: to preserve a focused evidence base for claims about source code comments while also acknowledging adjacent explanation-centered literature that helps explain why comments may serve as pedagogical scaffolds in programming education.

4.2. *Overview of the primary synthesis corpus and publication patterns*

The 18 primary studies mobilized in the main synthesis constitute the core analytical basis of the review. Table 2 provides a compact overview of these studies, including their year, main educational focus, evidence-layer classification, and role in the present synthesis. For ease of reference in the synthesis, these primary studies are denoted as S1–S18.

The primary synthesis was based on 18 studies mobilized in the main synthesis, spanning direct investigations of in-code comments, comment assessment, commenting behaviour, self-explanation in programming, code explanation, worked examples, scaffolded code comprehension, and related explanation-centered pedagogical practices.

In addition to the primary evidence base, seven supporting studies were retained for contextual, theoretical, or historical relevance. These studies were not treated as equally weighted primary evidence, but they informed interpretive framing, historical continuity, and adjacent conceptual links, particularly where they clarified boundary cases, methodological extensions, or broader explanation-centered perspectives relevant to programming education.

The retained corpus suggests that research relevant to the pedagogical role of source code comments is both **recent** and **still emerging**. Although a small number of foundational or historically relevant studies predate the main publication window, the majority of the retained evidence clusters in the period from the late 2010s onward, with a noticeable concentration in the years after 2018. This temporal pattern suggests that the topic has gained visibility primarily in the context of recent growth in programming education research, code comprehension studies, and explanation-centered instructional design.

A second descriptive pattern concerns publication venues. The retained studies are distributed across a **mixed set of journals, conference proceedings, and educationally relevant edited volumes**. This distribution is consistent with the interdisciplinary nature of the topic. Some studies emerge from computing education venues, where the emphasis is on novice learning, instructional interventions, and student performance; others come from computational science and engineering education, where code explanations are embedded in worked examples or domain-oriented modeling activities; and a smaller number are adjacent to software engineering or code comprehension research but remain relevant because of their implications for how comments mediate understanding.

This mixed venue distribution is methodologically important because it suggests that the topic does not yet constitute a fully consolidated research niche with a single dominant publication outlet. Instead, evidence is dispersed across multiple communities, which

helps explain why the literature can appear fragmented and why a targeted synthesis is necessary.

4.3. *Educational contexts and learner populations*

A clear pattern across the included studies is the strong predominance of **higher education** contexts. Most primary studies were conducted with undergraduate learners, especially in introductory or early-stage programming settings, engineering-related programming courses, or computational science and engineering environments. This concentration is one of the most consistent features of the evidence base and should be regarded as a central descriptive finding of the review.

Within higher education, the retained studies cover several related instructional contexts:

- **Introductory programming and CS1-like contexts**, where novice learners are developing foundational skills in reading, writing, and reasoning about code;
- **Computational science and engineering education**, where code comments and self-explanations are often embedded in worked examples and disciplinary problem-solving;
- **Code comprehension and debugging-oriented learning activities**, in which students are asked to interpret, trace, explain, or evaluate code rather than merely produce it.

By contrast, there is **very limited direct evidence** from K–12, secondary, or vocational education settings. This is particularly noteworthy given the pedagogical relevance of the topic for novice learners outside university contexts. From the perspective of the present review, this absence is not a minor detail but a substantive gap: if comments are to be interpreted as scaffolds for early learning, then the lack of studies in younger or more diverse learner populations represents a major limitation of the current evidence base.

In terms of learner profiles, the dominant population is best characterized as **novice or near-novice learners**. Even when studies are situated in higher education, they frequently involve students at early stages of programming development rather than advanced software engineering students. This pattern strengthens the pedagogical relevance of the corpus, since the potential value of comments as explanatory and metacognitive supports is likely to be greatest precisely when learners are still building conceptual fluency.

4.4. *Methodological and intervention patterns*

The included studies exhibit substantial methodological diversity, but several broad patterns are visible. First, the corpus is dominated by **small- to medium-scale educational studies** that examine targeted instructional interventions or pedagogical activities rather than large-scale longitudinal programs. This is consistent with the exploratory status of the field and with the practical constraints of classroom-based programming education research.

Second, the reviewed studies vary in how directly they place comments at the center of the intervention. At one end of the spectrum are studies in which **students are explicitly**

required or encouraged to write in-code comments as part of the learning task. These studies most closely match the original focus of the review and provide the strongest direct evidence regarding comments as pedagogical artifacts. At the other end are studies in which the focal mechanism is not comments per se, but rather **self-explanation, code explanation, guided worked examples, or structured code-reading prompts**. These studies remain highly relevant because they illuminate the broader learning mechanisms through which comments may operate, even when the surface intervention is framed differently.

Across the primary studies, four recurring intervention families can be identified:

1. **Direct comment-writing interventions**, in which learners annotate code they are reading or writing;
2. **Self-explanation-oriented interventions**, where students generate explanations about code behavior, intent, or structure, sometimes through comments and sometimes through adjacent explanatory formats;
3. **Worked-example and scaffolded comprehension interventions**, where explanatory prompts are embedded in examples, code traces, or guided inquiry sequences;
4. **Assessment-oriented studies**, which analyze student-generated explanations or related artifacts as indicators of understanding, reasoning quality, or conceptual development.

Methodologically, the corpus includes a mix of **qualitative, quantitative, and mixed-methods** designs. Some studies rely on performance-based measures such as quiz scores, comprehension outcomes, or task performance; others examine the content or quality of student explanations; and others combine outcome measures with reflective or interpretive analyses of student reasoning. This diversity is a strength in terms of conceptual richness, but it also limits direct comparability across studies and reinforces the appropriateness of a qualitative synthesis rather than a meta-analytic aggregation.

4.5. *Overview of reported outcomes*

Across the primary studies, the overall pattern of reported outcomes is **cautiously positive**. Although the evidence is heterogeneous and not uniformly comparable, the majority of the retained studies suggest that explanation-centered activities related to code comments are associated with pedagogically meaningful benefits, especially in novice learning contexts.

The most frequently recurring positive outcomes can be grouped into four broad categories.

4.5.1. *Improved code comprehension and conceptual clarity*

A substantial portion of the evidence suggests that requiring learners to explain code—whether through direct comments, self-explanations, or closely related explanatory prompts—supports deeper engagement with program structure and logic. These activities appear to help students move beyond surface-level reading and toward a more explicit

articulation of what the code is doing and why. In practical terms, this often translates into improved code comprehension, clearer reasoning about control flow and program purpose, and stronger conceptual linkage between local code segments and higher-level solution goals.

4.5.2. *Support for self-explanation and reflective reasoning*
One of the strongest converging themes in the corpus is that code comments can function as a form of **externalized self-explanation**. When students are prompted to verbalize intent, justify decisions, or describe the role of a block of code, they are encouraged to engage in a more reflective mode of learning. Across the included studies, this mechanism is frequently associated with stronger awareness of program structure, better inferential integration, and more deliberate reasoning during programming tasks. Even in studies where comments are not the literal intervention format, the underlying explanatory process is repeatedly identified as educationally valuable.

4.5.3. *Potential benefits for debugging and error diagnosis*
Although debugging is not the central focus of every retained study, the evidence suggests that comments and related explanation-centered scaffolds may also support **debugging and fault localization**. When learners are encouraged to articulate expected behavior or intended logic, discrepancies between intention and execution can become more visible. This does not mean that comments automatically improve debugging performance in all contexts; rather, the reviewed literature suggests that explanatory annotation can make debugging more conceptually accessible, especially for novice learners who struggle to connect observed outputs to underlying code logic.

4.5.4. *Assessment value of student-generated explanations*
A further important pattern is that student-generated comments or explanations can provide **diagnostically useful evidence** for teachers and researchers. Unlike executable correctness alone, explanatory artifacts can reveal partial understanding, misconceptions, strategic awareness, and the degree to which learners can connect implementation details to conceptual intent. This makes comments pedagogically relevant not only as learning supports, but also as potentially useful assessment traces. Several studies in the broader corpus point toward this dual role, suggesting that comments and adjacent explanation artifacts may support both instruction and evaluation.

4.6. *Summary of descriptive results*

Taken together, the descriptive results indicate that the literature on source code comments in programming education is small but conceptually rich. The strongest comment-specific evidence comes from a relatively limited subset of the primary studies, most of which are situated in higher education and focus on novice or near-novice learners. Within this direct comment-focused subset of the primary synthesis corpus, code comments rarely appear as isolated stylistic conventions; instead, they are most educationally meaningful when positioned as supports for comprehension, reflection, and reasoning.

These results also suggest that the central question is not simply whether comments are "good practice" in the professional sense, but rather **under what instructional conditions comments become pedagogically useful**. This issue is developed further in the next section, which synthesizes the evidence thematically and discusses the implications of the review for theory, instructional design, and future research.

5. Thematic Synthesis and Discussion

While the descriptive results presented in Section 4 show that the evidence base on source code comments in programming education remains relatively small and heterogeneous, the retained studies nevertheless support a coherent pedagogical interpretation. Importantly, the thematic synthesis distinguishes between the primary synthesis corpus and the broader supporting/contextual layer. The primary synthesis corpus comprises 18 empirical studies, including both direct comment-focused studies and adjacent mechanism- or methodological-supporting studies. In addition, 7 supporting/contextual studies were retained separately to provide theoretical, historical, or interpretive background. Accordingly, the main claims of this review concerning the pedagogical role of source code comments are grounded primarily in the direct comment-focused studies within the primary synthesis corpus, whereas adjacent primary studies are used to clarify plausible mechanisms and related instructional processes, and supporting/contextual studies are used only to provide bounded interpretive framing rather than equivalent evidence for comment-specific conclusions.

Thematic synthesis of the retained studies indicates that the pedagogical role of comments is best understood through four interrelated lenses: (i) comments as direct pedagogical artifacts, (ii) comments as externalized self-explanation, (iii) comments and related student-generated explanations as assessment traces, and (iv) comments as part of a broader family of explanation-centered scaffolds in programming education. These themes are discussed below in relation to the review questions and to the broader literature on program comprehension, metacognition, and instructional support for novice learners.

5.1. *Comments as pedagogical artifacts rather than mere documentation*

The first and most central conclusion of this review is that source code comments should not be understood exclusively through the lens of professional software documentation. In software engineering, comments are typically evaluated according to maintainability, readability, consistency, completeness, and long-term interpretability. In educational settings, however, the function of comments may be substantially broader and more immediate.

The included studies suggest that comments become pedagogically meaningful when they are used to support learners while they are actively engaging with code—that is, during code reading, code writing, code tracing, debugging, or revision. Under these conditions, comments are not simply retrospective annotations added to completed code; rather, they become part of the learning process itself. This distinction is conceptually important,

because it helps explain why educational commenting may remain valuable even when it does not fully align with professional conventions. In novice learning contexts, comments that would be considered overly explicit, redundant, or locally obvious in production code may still be instructionally beneficial if they help learners articulate intent, identify the role of a statement, or connect a local instruction to a broader algorithmic goal.

This finding addresses the first research question directly. Across the reviewed literature, code comments are rarely framed as valuable merely because they resemble "good coding style." Instead, they are pedagogically relevant when they are deliberately positioned as supports for comprehension, reflection, and reasoning. In other words, the evidence does not strongly support a generic claim that "students should always comment code because professionals do so." Rather, it supports the more specific claim that comments can become educationally powerful when they are integrated into intentional instructional design.

This distinction also helps clarify an important tension in the literature. Some educational advice about commenting is inherited from software engineering norms, whereas the reviewed evidence suggests that instructional decisions should instead be guided by learning goals. If the aim is to support novice understanding, code tracing, or conceptual clarity, then the pedagogical value of comments should be judged by whether they make reasoning more visible and manageable, not only by whether they satisfy professional documentation standards.

5.2. *Comments as externalized self-explanation*

The strongest and most theoretically coherent theme emerging from the review is that code comments can function as a form of *externalized self-explanation*. This interpretation, however, should be read with appropriate evidentiary discipline. The direct basis for treating comments as externalized self-explanation comes primarily from the subset of comment-focused studies that explicitly operationalized commenting as an explanatory activity. The adjacent self-explanation literature retained in the primary synthesis corpus provides convergent theoretical and pedagogical support for this interpretation, but it does not by itself constitute direct evidence about source code comments as a distinct instructional intervention. This theme provides the most convincing conceptual bridge between the narrow focus on comments and the broader explanation-centered evidence base that ultimately shaped the review.

Self-explanation has long been recognized in the learning sciences as a mechanism through which learners deepen understanding by generating explanatory inferences, integrating prior knowledge, and monitoring the adequacy of their own reasoning. In the context of programming education, this process is especially relevant because novice learners often produce syntactically plausible solutions without fully understanding the conceptual logic that underlies them. Asking students to explain what a line or block of code is doing, why it is needed, and how it contributes to the intended solution encourages a shift from surface-level code production toward more deliberate and reflective reasoning.

The review suggests that this mechanism is not merely plausible in theory, but repeatedly visible in practice. The most direct evidence comes from studies in which learners

were explicitly prompted to write in-code comments or explanatory annotations while engaging with code. The adjacent explanation-centered literature points toward a convergent pedagogical pattern even when the intervention is not implemented literally through source code comments. However, these formats should not be treated as interchangeable at the evidentiary level. Rather, they indicate that comment-based interventions can plausibly be understood as one specific instantiation of a broader family of strategies that require learners to make their reasoning explicit.

This observation is important for interpreting the evidence responsibly. A narrow comment-only reading of the literature would risk understating the educational value of comments simply because the field is small and terminologically fragmented. By contrast, an explanation-centered interpretation makes it possible to see comments as one concrete manifestation of a larger family of pedagogical strategies that aim to externalize reasoning and strengthen conceptual engagement.

This theme also helps answer the second research question. The review indicates that the pedagogical mechanisms most consistently associated with comments are not limited to readability or memorization. Instead, the strongest recurring mechanisms are deeper processing, inferential integration, conceptual linking, and reflective monitoring. In this sense, comments appear to matter educationally not because they add text to code, but because they can induce a qualitatively different mode of cognitive engagement with code.

Taken together, this theme is best interpreted as a structured conceptual bridge rather than as a collapse of categories. That is, the review does not claim that all self-explanation or explanation-centered programming studies are, in effect, studies of source code comments. Rather, it argues that a subset of the direct comment-focused literature is most coherently understood through a self-explanation lens, while the adjacent literature helps explain why such comment-based interventions may be pedagogically effective under specific instructional conditions.

5.3. *Comments as scaffolds for code comprehension and debugging*

A third major theme is that source code comments can function as scaffolds for program comprehension and, in some contexts, for debugging and error diagnosis. This theme is especially important because it connects the review to the broader contemporary literature showing that successful programming learning depends not only on writing code, but also on reading, interpreting, explaining, and evaluating code.

Across the retained studies, comments and related explanatory supports appear most useful when learners must bridge multiple levels of representation at once: natural-language problem statements, algorithmic ideas, program structure, control flow, and concrete syntax. For novice learners, this coordination is cognitively demanding. Comments can reduce some of this burden by drawing attention to the purpose of a block, clarifying the relationship between local instructions and global goals, or helping learners identify the intended behavior of a section of code before evaluating its execution.

This scaffolding function is especially relevant in code-reading and worked-example contexts. When explanatory comments are embedded in examples, they can guide atten-

tion to critical operations, decision points, and transformations that might otherwise remain opaque to beginners. Likewise, in debugging contexts, explanatory annotations can help students compare intended behavior with observed outcomes, making discrepancies easier to reason about conceptually. The review does not support an overly strong claim that comments automatically improve debugging performance in all instructional settings, but it does suggest that explanation-centered commenting can make debugging more pedagogically accessible.

This is a particularly useful finding for computing educators. Much introductory programming instruction still emphasizes code production disproportionately, even though novices often struggle most with interpretation, explanation, and diagnosis. The evidence reviewed here suggests that comments can be valuable precisely because they support these less visible but foundational dimensions of programming competence. In that sense, the pedagogical role of comments aligns closely with recent shifts in programming education toward comprehension-oriented teaching, trace-based reasoning, and structured debugging instruction.

5.4. *Comments and student-generated explanations as assessment traces*

A further important theme emerging from the review concerns the assessment value of code comments and related student-generated explanations. Here, the review reveals a particularly promising but still underdeveloped line of evidence: comments may be valuable not only as instructional supports, but also as diagnostically useful traces of student thinking.

Executable correctness alone often conceals important differences in understanding. Students may produce correct outputs through trial-and-error, partial pattern matching, or fragile procedural habits without being able to explain why their solution works. By contrast, comments and adjacent explanation artifacts can reveal the learner's underlying mental model more directly. They can expose whether a student understands the purpose of a variable, the role of a conditional branch, the logic of an iteration, or the relationship between a local step and the overall solution strategy.

From an assessment perspective, this makes comments pedagogically valuable in at least two ways. First, they can serve as *formative indicators* of understanding, helping teachers identify misconceptions, partial knowledge, or strategic confusion before these become entrenched. Second, they may support more nuanced evaluation of learning than traditional output-based assessment alone. This is particularly relevant in introductory programming, where visible code behavior is often an incomplete proxy for conceptual mastery.

At the same time, the review also suggests caution. Although the idea of using comments or code explanations as assessment evidence is pedagogically attractive, the literature is not yet mature enough to support strong standardization claims. The retained studies vary substantially in what counts as a "good" explanation, how explanation quality is judged, and whether explanations are evaluated for correctness, completeness, abstraction level, or pedagogical usefulness. This variability means that comments should currently

be seen as a promising assessment complement rather than as a fully stabilized assessment instrument.

Even so, the overall direction is clear: student-generated comments can help educators see more of the reasoning process than final code alone. This may be one of the most important implications of the review, especially in contexts where teachers need practical, low-cost ways to access student thinking during programming tasks.

5.5. *The broader explanation-centered interpretation*

One of the most consequential methodological and conceptual outcomes of this review is that the pedagogical role of source code comments cannot be fully understood if the topic is defined too narrowly. The screening process repeatedly showed that many of the most informative studies do not present themselves primarily as "code comment" studies, even though they address closely related mechanisms such as self-explanation, code explanation, worked examples, subgoal labeling, guided code reading, or reflection prompts.

This broader evidence base should not be treated as a methodological compromise or as mere supplementation. Rather, it reveals something substantive about the topic itself: source code comments are pedagogically meaningful largely because they belong to a wider family of explanation-centered learning supports. In other words, the educational importance of comments lies less in the artifact as such and more in the function it serves within the learning process.

This insight helps resolve an apparent contradiction in the literature. At the same time, the broader explanation-centered evidence suggests that requiring learners to explain code can be educationally beneficial under appropriate instructional conditions. The most coherent interpretation is therefore that code comments should be understood as one especially practical, low-friction, and authentic format through which explanation-centered pedagogy can be enacted in programming education.

This broader interpretation also increases the theoretical relevance of the review. Rather than concluding only that "there are few studies on code comments," the review can more meaningfully conclude that code comments are best conceptualized as pedagogical vehicles for explanation, reflection, and reasoning visibility. This makes the topic more educationally substantial and better connected to current work on program comprehension, debugging instruction, and metacognitive support.

5.6. *Implications for teaching practice*

Several practical implications for teaching emerge from the thematic synthesis.

First, comments appear to be most effective when they are used *strategically*, not ritually. Requiring students to "comment everything" without instructional purpose is unlikely to produce strong pedagogical benefits and may even encourage superficial or formulaic annotation. By contrast, prompting students to comment specific decision points, summarize the purpose of a block, explain a non-obvious transformation, or justify a chosen approach is more consistent with the mechanisms identified in the review.

Second, comments seem especially useful when integrated into *code reading*, *worked examples*, and *debugging tasks*, not only during final code submission. This aligns with the evidence that comments are most valuable when they mediate reasoning in progress rather than when they are appended after the fact as a compliance exercise.

Third, educators may benefit from treating comments as *windows into student thinking*. Even brief explanatory annotations can help instructors distinguish between correct code produced with genuine understanding and correct code produced through fragile or incomplete reasoning. This suggests a strong case for incorporating comments into formative assessment routines, code reviews, or guided lab activities.

Fourth, the review implies that commenting pedagogy should be *developmentally sensitive*. What is appropriate for novice learners may differ substantially from what is appropriate for more advanced learners. In early learning contexts, explicit and local comments may be pedagogically justified. As expertise develops, instruction can shift toward more selective, abstract, and professionally aligned commenting practices. This developmental view helps reconcile educational utility with software engineering norms instead of positioning them as mutually exclusive.

5.7. *Limitations of the evidence base*

Although the overall picture is encouraging, the review also reveals important limitations in the current evidence base.

The first limitation is **scope**. Directly comment-focused studies remain relatively few. As a result, the strongest comment-specific conclusions in the present review must remain appropriately bounded and should be interpreted primarily at the level of pedagogical plausibility, recurring patterns, and theoretical coherence rather than as definitive estimates of effect magnitude. The broader explanation-centered literature strengthens the interpretive framework of the review, but it does not eliminate the need for more direct evidence focused specifically on source code comments.

The second limitation is **contextual concentration**. Most of the retained studies are situated in higher education, especially undergraduate and novice programming contexts. There is comparatively little direct evidence from K–12, vocational education, adult reskilling programs, or non-university pathways. This limits the generalizability of the current evidence and should be treated as a major gap rather than a minor omission.

The third limitation is **methodological heterogeneity**. The included studies vary widely in terminology, intervention design, duration, outcome measures, and analytic approach. This diversity is intellectually productive, but it reduces direct comparability and prevents strong claims about magnitude or consistency of effects across contexts.

The fourth limitation concerns **access and publication type**. A small number of potentially relevant studies were only available at abstract level or through partial access. The review addressed this conservatively by distinguishing between primary synthesis studies and supporting/contextual studies, but this inevitably narrows the directly analyzable corpus.

Although computational pre-filtering and heuristic relevance classification improved transparency and auditability, these steps necessarily reflected design choices in exclusion

patterns, keyword families, and thresholding. To mitigate this, automated decisions were used only for prioritization and documentation, while final inclusion remained based on manual scholarly review.

These limitations do not invalidate the review. Rather, they clarify the appropriate level of inference: the present study supports a theoretically coherent and educationally meaningful interpretation of code comments, but it should be read as a structured synthesis of an emerging literature rather than as a definitive final word on effect sizes or universal best practices.

5.8. *Directions for future research*

The review points to several high-priority directions for future research.

First, there is a need for **more direct experimental and quasi-experimental studies** that isolate the pedagogical contribution of source code comments more explicitly. Many existing studies are informative but embed comments within broader intervention bundles. Future work should compare carefully designed commenting conditions, explanation formats, and control conditions in ways that make the role of comments easier to interpret.

Second, research should expand into **more diverse educational contexts**. The current concentration in higher education leaves important questions unanswered about how comments function in secondary education, vocational computing education, teacher training, and adult learning environments. This is particularly relevant because the pedagogical case for comments may be especially strong in earlier learning stages.

Third, there is a strong opportunity to investigate **assessment frameworks for student-generated code explanations**. If comments are to be used formatively or summatively, more work is needed to define reliable criteria for evaluating explanation quality, conceptual adequacy, abstraction level, and alignment with intended learning outcomes.

Fourth, future studies should examine **how commenting practices evolve with expertise**. A developmental trajectory from novice-supportive explanatory commenting toward more selective and professionally authentic commenting would be especially useful for curriculum design. This would help educators avoid the false dichotomy between "educational commenting" and "real-world commenting."

Finally, the rapid emergence of **AI-supported programming environments** creates an especially timely research opportunity. As learners increasingly rely on AI systems that generate, explain, and annotate code, the pedagogical role of comments may shift substantially. Future research should therefore examine whether AI-generated comments support or undermine learner explanation, how students interpret machine-produced annotations, and whether requiring learners to critique or revise generated comments can strengthen conceptual understanding.

5.9. *Discussion summary*

Overall, the thematic synthesis supports a clear conclusion: source code comments can be pedagogically meaningful in programming education, but their educational value does not

derive primarily from their status as software documentation artifacts. Instead, comments matter most when they function as explanation-centered learning supports—that is, when they help learners externalize reasoning, interpret program behavior, structure attention, reflect on intent, and make their thinking visible to themselves and to instructors.

This conclusion does not justify simplistic prescriptions such as requiring comments in all code or equating more comments with better learning. Rather, it suggests a more nuanced instructional principle: *comments are most valuable when they are used deliberately to support explanation, comprehension, and reflection, especially in novice learning contexts*. This principle integrates the direct evidence on code comments with the broader literature on self-explanation, scaffolding, and program comprehension, and it provides a strong foundation for the conclusions and recommendations presented in the next section.

6. Conclusions and Implications

This review set out to examine the pedagogical role of source code comments in programming education through a structured qualitative synthesis of a small but conceptually layered evidence base. The main synthesis was anchored in a primary corpus that included both direct comment-focused studies and a limited set of adjacent explanation-centered studies retained to clarify related pedagogical mechanisms. Supporting/contextual studies were retained separately to provide bounded theoretical, historical, or interpretive framing. Accordingly, the principal comment-specific conclusions of the review are grounded primarily in the direct comment-focused studies, while adjacent and supporting/contextual studies are used only to clarify mechanisms, strengthen interpretive coherence, and support pedagogical contextualization rather than as equivalent direct evidence.

Taken together, the reviewed literature supports three overarching conclusions.

First, **source code comments can be pedagogically meaningful**. The available evidence suggests that comments are not merely stylistic conventions inherited from professional software practice. In educational contexts, they can serve as instructional artifacts that help learners make sense of code, articulate intent, connect local statements to global goals, and engage more reflectively with programming tasks.

Second, **the strongest pedagogical interpretation of comments is explanation-centered**. The most consistent and theoretically grounded pattern in the literature is that comments become educationally valuable when they function as a form of externalized self-explanation or as part of broader explanation-centered learning designs. In this sense, their educational importance lies less in the textual artifact itself and more in the cognitive and metacognitive processes they can provoke: deeper processing, conceptual linking, reflective monitoring, and reasoning visibility.

Third, **the current evidence base remains promising but still limited**. Direct comment-focused studies are relatively few, and the retained literature is concentrated primarily in higher education and novice programming contexts. Methodological heterogeneity, partial access to some records, and variation in terminology and design all mean that the field is not yet mature enough to support highly standardized prescriptions or

strong quantitative generalizations. The present review should therefore be understood as a PRISMA-informed qualitative systematic review with a layered-evidence synthesis strategy rather than as a definitive meta-analytic verdict.

Despite these limitations, the review provides a meaningful contribution to programming education research and practice. It helps distinguish between comments as software documentation and comments as pedagogical scaffolds, and it offers a coherent framework for interpreting comments through four recurring educational functions: (i) direct support for code comprehension, (ii) externalized self-explanation, (iii) support for debugging and reflective reasoning, and (iv) traceable evidence of student understanding for formative assessment.

From a practical perspective, the review suggests that educators should avoid treating commenting as a purely stylistic requirement or as a generic marker of "good practice." Instead, comments should be used selectively and intentionally, especially in tasks that involve code reading, worked examples, debugging, tracing, and guided reflection. Prompting students to explain the purpose of a block, justify a choice, or describe intended behavior is likely to be more educationally valuable than requiring exhaustive or formulaic comments throughout all code.

The review also points to several important research priorities. More direct empirical work is needed to isolate the contribution of comments within controlled instructional designs. More studies are needed in secondary, vocational, and non-university settings. Stronger assessment frameworks are needed for evaluating student-generated comments and code explanations. Finally, the rise of AI-assisted programming environments creates a timely need to understand how human-generated and machine-generated code comments may interact in future learning contexts.

Accordingly, the main conclusions of this review should be read in line with this evidentiary structure: they are grounded primarily in the direct comment-focused studies retained in the main synthesis, while adjacent primary studies are used mainly to clarify plausible pedagogical mechanisms and related instructional processes. Supporting/contextual studies serve only to provide bounded theoretical or interpretive framing rather than equivalent evidence for comment-specific conclusions.

In conclusion, the central message of this review is not simply that source code comments "help" programming learning in a generic sense. Rather, the more defensible and educationally meaningful conclusion is that source code comments can become pedagogically meaningful tools when they are designed and used as supports for explanation, comprehension, reflection, and metacognitive engagement. This perspective provides a more robust foundation for both instructional practice and future research, and it positions comments not as peripheral stylistic artifacts, but as potentially important components of explanation-centered programming pedagogy.

A. Final database query strings

The strings below correspond to the final retrieval configuration used for the review after iterative refinement. Earlier exploratory query variants were used only for testing and

protocol adjustment and are therefore not reported as part of the final reproducible search configuration.

A.1. *Scopus (final retrieval configuration)*

```
A_core_direct:
TITLE-ABS-KEY(
  ("source code comments" OR "code comments" OR
   "in-code comments" OR "inline comments" OR
   "commenting code" OR "commented code")
  AND
  ("programming education" OR "computer science education" OR
   "computing education" OR "introductory programming" OR "CS1" OR
   "novice programmers" OR "learning programming" OR
   "teaching programming")
)

B_self_explanation:
TITLE-ABS-KEY(
  ("source code comments" OR "code comments" OR
   "in-code comments" OR "self-explanation" OR
   "self explanation")
  AND
  ("program comprehension" OR "code comprehension" OR
   "scaffolding" OR "metacognition" OR "worked examples")
  AND
  ("programming education" OR "computer science education" OR
   "learning programming" OR "introductory programming" OR "CS1")
)

C_pedagogical_broad:
TITLE-ABS-KEY(
  ("commenting practices" OR "commenting source code" OR
   "code commenting" OR "comments in source code")
  AND
  ("students" OR "learners" OR "education" OR "teaching" OR "learning")
  AND
  ("programming" OR "computer science")
)
```

A.2. *ERIC*

```
A_code_comments_prog_edu:
code comments AND programming education

B_in_code_comments_cs_edu:
in code comments AND computer science education

C_self_explanation_programming:
self explanation AND programming
```

Declarations

Funding

This work was funded by the project Digital Path—Capacitação para o Futuro, operation no. 11059, approved under the Call for Expressions of Interest 03/C06-i07/2023 and the Invitation to submit proposals for the conclusion of programme contracts with DGES

07/C06-i07/2024, under Impulsos Mais Digital – sub-measure Reforço das Competências Digitais, financed by the European funds allocated to Portugal under the Recovery and Resilience Plan (PRR), within the framework of the European Union Recovery and Resilience Facility (RRF), NextGenerationEU, for the period 2021–2026.

Competing interests

The authors declare no competing interests.

Ethics approval and consent to participate

This study is a systematic literature review based exclusively on published and publicly accessible scholarly literature. It did not involve human participants, human data, animal subjects, or any new data collection. Therefore, ethics committee approval and informed consent were not required.

Data/materials availability

The data presented in this study are available from the corresponding author upon reasonable request.

Generative AI disclosure

During the preparation of this work, the authors used ChatGPT to support language polishing and readability. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

Author contributions

Conceptualization, G.S. and M.J.C.S.R.; methodology, G.S. and M.J.C.S.R.; formal analysis, G.S. and M.J.C.S.R.; investigation, G.S.; data curation, G.S.; writing–original draft preparation, G.S.; writing–review and editing, G.S. and M.J.C.S.R.; supervision, M.J.C.S.R. All authors have read and agreed to the published version of the manuscript.

References

Alhassan, R. (2017). The Effect of Employing Self-Explanation Strategy with Worked Examples on Acquiring Computer Programing Skills. *Journal of Education and Practice*, 8(6), 186–196.

Beck, P., Mohammadi-Aragh, M.J., Archibald, C. (2019). An Initial Exploration of Machine Learning Techniques to Classify Source Code Comments in Real-time. In: *2019 ASEE Annual Conference & Exposition*, Tampa, Florida.

Beck, P.J., Jean Mohammadi-Aragh, M., Archibald, C., Jones, B.A., Barton, A. (2018). Real-time Metacognition Feedback for Introductory Programming Using Machine Learning. In: *2018 IEEE Frontiers in Education Conference (FIE)*, pp. 1–5. https://doi.org/10.1109/FIE.2018.8658973.

Chapagain, J., Rus, V. (2025). Automated assessment of student self-explanation in code comprehension using pre-trained language models. In: *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*. AAAI'25/IAAI'25/EAAI'25. AAAI Press. https://doi.org/10.1609/aaai.v39i28.35169.

Chen, C.-Y. (2025). Effects of Worked Examples with Explanation Types and Learner Motivation on Cognitive Load and Programming Problem-Solving Performance. *ACM Transactions on Computing Education*, 25(2), Article 23, 1–19. https://doi.org/10.1145/3732791.

Chen, M.-P., Feng, C.-Y. (2026). Goal-Setting and Self-Explanation in Elementary Programming: A Self-Regulated Learning Study. *Journal of Computer Assisted Learning*, 42(1), e70177. https://doi.org/10.1002/jcal.70177.

Cheng, Y.-P., Hsiao, I., Starčič, A.I. (2025). Integrating Generative Artificial Intelligence with Self-explanations Scaffolding to Enhance Students' Problem-Solving Skills and Learning Performance. In: *Innovative Technologies and Learning: 8th International Conference, ICITL 2025, Oslo, Norway, August 5–7, 2025, Proceedings, Part II*. Springer-Verlag, Berlin, Heidelberg, pp. 279–288. https://doi.org/10.1007/978-3-031-98197-5_30.

Combéfis, S. (2022). Automated Code Assessment for Education: Review, Classification and Perspectives on Techniques and Tools. *Software*, 1(1), 3–30. https://doi.org/10.3390/software1010002.

DePasquale, P.J., Locasto, M.E., Kaczmarczyk, L., Martinovic, M. (2012). // TODO: Help Students Improve Commenting Practices. In: *2012 Frontiers in Education Conference Proceedings*, pp. 1–6. https://doi.org/10.1109/FIE.2012.6462504.

Ding, Y., Wang, B., Zhang, H., Li, S. (2025). Exploration and Practice of Code Commenting Based on Prompt Engineering. In: *2025 7th International Conference on Computer Science and Technologies in Education (CSTE)*, pp. 98–102. https://doi.org/10.1109/CSTE64638.2025.11092128.

Figl, K., Kirchner, M., Baltes, S., Felderer, M. (2025). The Influence of Code Comments on the Perceived Helpfulness of Stack Overflow Posts. *Empirical Software Engineering*, 30, Article 178. https://doi.org/10.1007/s10664-025-10727-w.

Garces, S., Vieira, C., Ravai, G., Magana, A.J. (2023). Engaging Students in Active Exploration of Programming Worked Examples. *Education and Information Technologies*, 28(3), 2869–2886. https://doi.org/10.1007/s10639-022-11247-6.

Huang, Y., Jia, N., Zhou, Q., Chen, X.-P., Xiong, Y.-F., Luo, X.-N. (2018). Method Combining Structural and Semantic Features to Support Code Commenting Decision. *Ruan Jian Xue Bao/Journal of Software*, 29, 2226–2242. https://doi.org/10.13328/j.cnki.jos.005528.

Ishaq, K., Alvi, A., Haq, M.I.u., Rosdi, F., Choudhry, A.N., Anjum, A., Khan, F.A. (2024). Level Up Your Coding: A Systematic Review of Personalized, Cognitive, and Gamified Learning in Programming Education. *PeerJ Computer Science*, 10, e2310. https://doi.org/10.7717/peerj-cs.2310.

Jennings, J., Muldner, K. (2021). Investigating Students' Reasoning in a Code-Tracing Tutor. In: *Artificial Intelligence in Education: 22nd International Conference, AIED 2021, Utrecht, The Netherlands, June 14–18, 2021, Proceedings, Part I*. Springer-Verlag, Berlin, Heidelberg, pp. 203–214. https://doi.org/10.1007/978-3-030-78292-4_17.

Kerschbaumer, D., Schatz, C., Ruprechter, T., Gütl, C., Steinmaurer, A. (2025). Do Comments Matter? Investigating Students' Source Code Comment Behaviour and Its Relation to Academic Success in a CS1 Course. In: Auer, M.E., Rüütmann, T. (Eds.), *Futureproofing Engineering Education for Global Responsibility: Proceedings of the 27th International Conference on Interactive Collaborative Learning, ICL 2024*. Lecture Notes in Networks and Systems*: Vol.* 1261. Springer, pp. 519–530. https://doi.org/10.1007/978-3-031-85649-5_51.

Lee, N. (2013). *The Effects of Self-Explanation and Reading Questions and Answers on Learning Computer Programming Language*. Phd dissertation, University of Nevada, Las Vegas, Las Vegas, NV, USA. https://doi.org/10.34917/5363915.

Lee, N., Hong, E. (2017). Examining the Effects of Two Computer Programming Learning Strategies: Self-Explanation versus Reading Questions and Answers. *IAFOR Journal of Education*, 5(Special Issue), 69–88.

Lehtinen, T., Santos, A.L., Sorva, J. (2021). Let's Ask Students About Their Programs, Automatically. In: *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pp. 467–475. https://doi.org/10.1109/ICPC52881.2021.00054.

Lekshmi-Narayanan, A.-B., Brusilovsky, P. (2024). Evaluating Correctness of Student Code Explanations: Challenges and Solutions. In: *Proceedings of the 8th Educational Data Mining in Computer Science Education (CSEDM) Workshop at EDM 2024*. CEUR Workshop Proceedings*: Vol.* 3796, Atlanta, GA, USA.

Lombrozo, T. (2006). The structure and function of explanations. *Trends in Cognitive Sciences*, 10(10), 464–470. https://doi.org/10.1016/j.tics.2006.08.004.

Margulieux, L., Catrambone, R. (2017). Using Learners' Self-Explanations of Subgoals to Guide Initial Problem Solving in App Inventor. In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ICER '17. Association for Computing Machinery, New York, NY, USA, pp. 21–29. https://doi.org/10.1145/3105726.3106168.

Navarro-Cota, C., Molina, A.I., Redondo, M.A., Lacave, C. (2025). Individual differences in computer programming: a systematic review. *Behaviour & Information Technology*, 44(2), 357–375. https://doi.org/10.1080/0144929X.2024.2317377.

Niazi, T., Das, T., Ahmed, G., Waqas, S.M., Khan, S., Khan, S., Abdelatif, A.A., Wasi, S. (2023). Investigating Novice Developers' Code Commenting Trends Using Machine Learning Techniques. *Algorithms*, 16(1). https://doi.org/10.3390/a16010053.

Oli, P., Banjade, R., Lekshmi Narayanan, A., Chapagain, J., Tamang, L.J., Brusilovsky, P., Rus, V. (2023). Improving Code Comprehension Through Scaffolded Self-Explanations. In: Wang, N., Rebolledo-Mendez, G., Dimitrova, V., Matsuda, N., Santos, O.C. (Eds.), *Artificial Intelligence in Education. AIED 2023*. Communications in Computer and Information Science*: Vol.* 1831. Springer, pp. 478–483. https://doi.org/10.1007/978-3-031-36336-8_74.

Page, M.J., McKenzie, J.E., Bossuyt, P.M., Boutron, I., Hoffmann, T.C., Mulrow, C.D., Shamseer, L., Tetzlaff, J.M., Akl, E.A., Brennan, S.E., Chou, R., Glanville, J., Grimshaw, J.M., Hróbjartsson, A., Lalu, M.M., Li, T., Loder, E.W., Mayo-Wilson, E., McDonald, S., McGuinness, L.A., Stewart, L.A., Thomas, J., Tricco, A.C., Welch, V.A., Whiting, P., Moher, D. (2021). The PRISMA 2020 Statement: An Updated Guideline for Reporting Systematic Reviews. *BMJ*, 372, n71. https://doi.org/10.1136/bmj.n71.

Rani, P., Blasi, A., Stulova, N., Panichella, S., Gorla, A., Nierstrasz, O. (2023). A decade of code comment quality assessment: A systematic literature review. *Journal of Systems and Software*, 195, 111515. https://doi.org/10.1016/j.jss.2022.111515.

Recker, M.M., Pirolli, P. (1992). Student strategies for learning programming from a computational environment. In: Frasson, C., Gauthier, G., McCalla, G.I. (Eds.), *Intelligent Tutoring Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 382–394.

Rus, V., Brusilovsky, P., Tamang, L.J., Akhuseyinoglu, K., Fleming, S. (2022). DeepCode: An Annotated Set of Instructional Code Examples to Foster Deep Code Comprehension and Learning. In: Crossley, S., Popescu, E. (Eds.), *Intelligent Tutoring Systems*. Springer International Publishing, Cham, pp. 36–50.

Sandoval-Medina, C., Arévalo-Mercado, C., Muñoz-Andrade, E., Muñoz-Arteaga, J. (2024). Self-Explanation Effect of Cognitive Load Theory in Teaching Basic Programming. *Journal of Information Systems Education*, 35(3), 303–312. https://doi.org/10.62273/GMIV1698.

Santos, A., Soares, T., Garrido, N., Lehtinen, T. (2022). Jask: Generation of Questions About Learners' Code in Java. In: *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education, Vol. 1*. ITiCSE '22. Association for Computing Machinery, New York, NY, USA, pp. 117–123. https://doi.org/10.1145/3502718.3524761.

Sengupta, S. (2020). Learning to Code in a Virtual World: A Preliminary Comparative Analysis of Discourse and Learning in Two Online Programming Communities. In: *Companion Publication of the 2020 Conference on Computer Supported Cooperative Work and Social Computing*. ACM, Virtual Event USA, pp. 389–394. https://doi.org/10.1145/3406865.3418319.

Shi, N. (2021). Improving Undergraduate Novice Programmer Comprehension through Case-Based Teaching with Roles of Variables to Provide Scaffolding. *Information*, 12(10). https://doi.org/10.3390/info12100424. https://www.mdpi.com/2078-2489/12/10/424.

Song, X., Sun, H., Wang, X., Yan, J. (2019). A Survey of Automatic Generation of Source Code Comments: Algorithms and Techniques. *IEEE Access*, 7, 111411–111428. https://doi.org/10.1109/ACCESS.2019.2931579.

Sorva, J. (2013). Notional Machines and Introductory Programming Education. *ACM Transactions on Computing Education*, 13(2), Article 8, 1–31. https://doi.org/10.1145/2483710.2483713.

Sudol-DeLyser, L.A. (2015). Expression of Abstraction: Self Explanation in Code Production. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. Association for

Computing Machinery, New York, NY, USA, pp. 272–277. https://doi.org/10.1145/2676723.2677222.

Sukamto, R.A., Rischa, M.N.F., Piantari, E., Wibisono, Y., Megasari, R. (2023). Auto Code Comment Assessment for Online Judge Using Word Embedding and Word Mover's Distance. In: *Proceedings of the 2022 International Conference on Computer, Control, Informatics and Its Applications*. ACM, pp. 345–349. https://doi.org/10.1145/3575882.3575949.

Tamang, L.J., Alshaikh, Z., Khayi, N.A., Oli, P., Rus, V. (2021). A Comparative Study of Free Self-Explanations and Socratic Tutoring Explanations for Source Code Comprehension. In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. ACM, pp. 219–225. https://doi.org/10.1145/3408877.3432423.

Tan, L. (2009). *Leveraging code comments to improve software reliability*. Ph.D., University of Illinois at Urbana-Champaign, United States – Illinois. https://www.proquest.com/docview/205446726/abstract/101B817DFE1A4CB6PQ/1.

Tsai, M.-J., Chien, F.P., Sun, W.-T., Jha, N.K. (2025). How block-based programming supports novice learners' coding comprehension: Evidence from eye-tracking lag-sequential analysis. *Computers & Education*, 239, 105430. https://doi.org/10.1016/j.compedu.2025.105430.

van der Werf, V., Aivaloglou, E., Hermans, F., Specht, M. (2022). What does this Python code do? An exploratory analysis of novice students' code explanations. In: *Proceedings of the 10th Computer Science Education Research Conference*. CSERC'21. Association for Computing Machinery, New York, NY, USA, pp. 94–107. https://doi.org/10.1145/3507923.3507956.

Vieira, C., Roy, A., Magana, A.J., Falk, M.L., Reese, M.J. (2016). In-Code Comments as a Self-Explanation Strategy for Computational Science Education. In: *Proceedings of the 123rd ASEE Annual Conference and Exposition*. ASEE Conferences, New Orleans, Louisiana. https://doi.org/10.18260/p.25642.

Vieira, C., Magana, A.J., Falk, M.L., Garcia, R.E. (2017). Writing In-Code Comments to Self-Explain in Computational Science and Engineering Education. *ACM Transactions on Computing Education*, 17(4). https://doi.org/10.1145/3058751.

Vieira, C., Magana, A.J., Roy, A., Falk, M.L. (2019). Student Explanations in the Context of Computational Science and Engineering Education. *Cognition and Instruction*, 37(2), 201–231. https://doi.org/10.1080/07370008.2018.1539738.

Vihavainen, A., Miller, C.S., Settle, A. (2015). Benefits of Self-explanation in Introductory Programming. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. Association for Computing Machinery, New York, NY, USA, pp. 284–289. https://doi.org/10.1145/2676723.2677260.

Yang, S., Baird, M., O'Rourke, E., Brennan, K., Schneider, B. (2024). Decoding Debugging Instruction: A Systematic Literature Review of Debugging Interventions. *ACM Transactions on Computing Education*, 24(4), Article 45, 1–44. https://doi.org/10.1145/3690652.