

# Small Language Models for Generating Code Comments

Kimia Abedini<sup>1</sup>, Charles Koutcheme<sup>1</sup>, Jing Fan<sup>1</sup>, Arto Hellas<sup>1</sup> and Juho Leinonen<sup>1,\*</sup>

<sup>1</sup>Aalto University, Espoo, Finland

## Abstract

Generating line-level code explanations at scale highlights the potential of Large Language Models (LLMs) in computing education. Most prior work relies on large proprietary models accessed through closed APIs, which raises concerns about API cost, student data privacy, and the inability to deploy models on student devices. This paper investigates whether Small Language Models (SLMs) under 4B parameters in size can produce high-quality line-level explanations for CS1 students. We evaluate six models, namely Qwen3.5 at 0.8B, 2B, and 4B scales, SmoLLM3-3B, Llama-3.2-1B and Llama-3.1-8B as a larger reference point, on 104 annotated introductory Python programs from the ACOS learning platform. We compare three prompting strategies varying along two axes: whether the model sees the full program (Context-Aware) or only the target line (Line-Only), and within the Context-Aware setting, whether the prompt contains a single worked demonstration (one-shot) or none (zero-shot). Quality is assessed with G-Eval framework on five dimensions: Correctness, Clarity, Conciseness, Pedagogical Completeness, and Abstraction Level, complemented by automated reference-based metrics. Our results show that model scale improves explanation quality, and one-shot demonstrations consistently improve performance, driven mostly by gains in conciseness. However, demonstrations trade pedagogical completeness for brevity, and the improvement shrinks as scale increases. Clarity and Abstraction Level are largely insensitive to model size and prompting strategy, suggesting these dimensions are already well handled by even the smallest models. These findings indicate that SLMs under 4B parameters in size, combined with context-aware one-shot prompting, are a promising privacy-preserving approach for generating line-level code explanations.

## Keywords

Small Language Models, Code Comments, Computing Education, Code Explanations

## 1. Introduction

Over the past few years, Large Language Models (LLMs) have emerged as a central tool in computing education [1, 2]. They have been studied in many areas of computer science learning, including generating code for novice programming problems [3, 1], debugging assistance and the enhancement of programming error messages [4, 5, 6], next-step hint generation [7, 8], natural language feedback generation for beginner programmers [9, 10, 11, 12] and source code explanation for learners [13, 14]. Among these applications, generating code comments and line-level explanations holds a particular place in CS1, where reading and tracing skills are foundational and well-formed comments scaffold program comprehension for novices [15, 16].

Several approaches have been used to produce and assess code comments and most of them rely on large proprietary models accessed through closed commercial APIs. Their deployment in education raises three concerns. First, API costs scale linearly with class size and can be prohibitive for under-resourced institutions. Second, sending student code to third-party servers raises privacy issues [17]. Third, on-device deployment is not possible, which prevents offline use on student laptops and limits reproducibility, since closed-weight models can change silently. These constraints motivate a closer look at Small Language Models (SLMs) as a substitute for the larger, higher-parameter models that many current pipelines depend on, and lead to our main Research Question (RQ): *How accurately do SLMs under 4B parameters in size generate line-level code explanations compared to reference annotations?* We decompose this question into three sub-RQs:

CSEDM'26: 10th Educational Data Mining in Computer Science Education (CSEDM) Workshop, June 28, 2026, Seoul, South Korea

\*Corresponding author.

✉ kimia.abedini@aalto.fi (K. Abedini); charles.koutcheme@aalto.fi (C. Koutcheme); jing.fan@aalto.fi (J. Fan); arto.hellas@aalto.fi (A. Hellas); juho.2.leinonen@aalto.fi (J. Leinonen)

🆔 0009-0007-2586-4920 (K. Abedini); 0000-0002-2272-2763 (C. Koutcheme); 0009-0002-1170-6693 (J. Fan); 0000-0001-6502-209X (A. Hellas); 0000-0001-6829-9449 (J. Leinonen)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

**RQ1.** How does model scale (parameter count) affect explanation quality?

**RQ2.** How does providing the surrounding program context, rather than the target line in isolation, affect explanation quality?

**RQ3.** How does augmenting the prompt with a single worked demonstration affect explanation quality, relative to a zero-shot baseline?

RQ1 aims to characterize how rubric scores evolve from the 0.8B scale up to the 8B reference, and to test whether a small but recent model can match a larger model from an earlier generation. It targets a recent debate on whether SLMs can serve as viable substitutes for the larger, higher-parameter models in specialized code-related tasks, in particular when deployment constraints such as on-device execution, student data privacy, and bounded compute budgets are central. We do not aim to identify the single best checkpoint for the task. Rather, we characterize how the five rubric dimensions and the lexical metrics evolve across a coherent size sweep, and we test whether a recent SLM under 4B parameters in size can match an earlier-generation 8B reference. RQ2 targets the role of program context. Most prior work on code explanation operates on isolated snippets without surrounding program context [18, 19]. We test whether exposing the model to the full program is necessary in the CS1 setting, where a line rarely makes pedagogical sense without its context. RQ3 targets the effectiveness of augmenting the prompt with a single worked demonstration to elicit line-level explanations, contributing to ongoing discussions around in-context learning for code explanation. While one-shot prompting has been reported to consistently improve over zero-shot prompting in code explanation [20], with gains being more pronounced for medium-sized models [19], these findings come from software engineering settings and are typically reported on aggregate metrics. Whether a single demonstration helps along the multiple, sometimes competing, dimensions of pedagogical quality remains unexplored.

**Contributions.** This paper makes the following contributions:

- (1) We present a systematic study of SLMs under 4B parameters in size for generating line-level code explanations in CS1. We evaluate five recent open-weight models: Qwen3.5 (0.8B, 2B, and 4B), SmolLM3-3B, and Llama-3.2-1B, against Llama-3.1-8B as a larger reference point. The evaluation is conducted on 104 annotated introductory Python programs from the ACOS<sup>1</sup> learning platform.
  - (2) We adopt a pedagogically grounded, multi-dimensional evaluation framework. Building on G-Eval [21], we assess each generated explanation along five rubric dimensions: Correctness, Clarity, Conciseness, Pedagogical Completeness, and Abstraction Level. We complement these dimensions with three automated metrics, namely ROUGE-L, BLEU-1, and CodeBERTScore-F1.
  - (3) We provide a controlled comparison of two prompt-design decisions, namely whether the model sees the surrounding program (Line-Only vs. Context-Aware) and whether the prompt contains a worked demonstration (zero-shot vs. one-shot). The Context-Aware framing improves all six checkpoints over the Line-Only baseline, with the gain concentrated on Pedagogical Completeness, which is the dimension that explicitly rewards connection to the surrounding program. The addition of a single demonstration further improves the aggregate judge mean across all six checkpoints, but this aggregate gain is driven almost entirely by Conciseness, comes at the cost of Pedagogical Completeness, and shrinks monotonically with model scale within the Qwen3.5 family.
- All code, prompts, and evaluation scripts used in this study are available at here<sup>2</sup>.

**Outline.** The rest of the paper is organized as follows. In Section 2, we review related work. In Section 3, we introduce the experimental setup, covering the curated ACOS corpus (Section 3.1), the SLMs under evaluation (Section 3.2), the prompting strategies (Section 3.3), and the evaluation procedures (Section 3.4). Section 4 presents the main empirical results, organized as a series of findings. Section 5 outlines the limitations of the study. Finally, Section 6 provides concluding discussions

---

<sup>1</sup><https://acos.cs.vt.edu/>

<sup>2</sup><https://kimia-abedini.github.io/slm-commentator/>

on the implications of our findings for educational deployment. Appendix A provides additional implementation details regarding the experimental setup and the evaluation rubric.

## **2. Related Work**

### **2.1. Code Explanation in Computing Education**

Reading, tracing, and explaining code are foundational skills that strongly correlate with the ability to write code [22], and line-level comments scaffold this comprehension by linking each syntactic element to its role in the surrounding program, in line with the Block Model of program understanding [23] and recent empirical work confirms that the presence of well-formed comments increases the perceived helpfulness of code for novices [15, 16]. Several tools and instructional designs build on these findings. Hosseini et al. [24] introduced PCEX, an interactive tool that combines explorable worked examples with embedded challenges, and demonstrated through a semester-long classroom study that interactive examples lead to significantly higher time-on-task and better coding performance compared to static textbook-style alternatives. Oli et al. [25] further showed that scaffolded self-explanations improve code comprehension beyond passive reading, motivating the development of automatically generated line-level explanations that can serve as scaffolds for such activities.

### **2.2. Generating Code Comments and Explanations**

Approaches to producing code comments range from human annotation to fully automated generation. Human annotation by instructors or crowd workers provides high quality but does not scale, and the resulting datasets remain small or rely on noisy proxies such as scraped docstrings [26, 18]. Rule-based and template-based generators [27, 28] are deterministic and cheap, yet they depend on well-named identifiers, which novice student code rarely satisfies, and they produce a single rigid style that cannot adapt to learner level.

LLM-based generation addresses many of these limitations. Leinonen et al. [14] conducted a large-scale study comparing GPT-3-generated code explanations with student-written ones in a first-year programming course of approximately 1000 students, finding that LLM-generated explanations were rated as more accurate and easier to understand. Bhattacharya and Gupta [19] investigated selective shot learning strategies for open-source Code-LLMs and found that few-shot prompting consistently improves over zero-shot prompting, with gains being more pronounced in medium-sized models. Their work is directly relevant to ours, though it focuses on software engineering use cases rather than pedagogical quality for novice learners, and it does not address educational deployment constraints such as privacy or offline use.

### **2.3. Small Language Models for Code Tasks**

Recent work on SLMs has begun to assess whether compact open-weight models can replace proprietary ones in code-related tasks. Mahade Hasan et al. [29] conducted a comprehensive empirical study of 20 open-source SLMs ranging from 0.4B to 10B parameters across five code generation benchmarks. Their results show that model size significantly affects functional correctness, but several compact models achieve competitive performance against much larger ones. Importantly, VRAM usage scales sharply with parameter count while inference time does not vary significantly, meaning memory is the primary practical constraint for SLM deployment. These findings motivate our focus on models between 1B and 4B parameters, which are feasible for deployment on student laptops or institutional servers without GPU clusters.

Within computing education specifically, studies [30, 31] have investigated whether compact open-weight models, paired with retrieval-augmented generation, can replace larger models when supporting computer science learning. Their work argues that small models can reach usable quality for educational support once the surrounding pipeline is designed appropriately, and reports practical deployment

considerations for such setups. Our study is complementary. We isolate the effect of model scale and prompt design on intrinsic explanation quality under a controlled rubric, without retrieval augmentation, which lets us attribute the observed differences to the model and the prompt rather than to a retrieval component.

### 3. Experiments

In this section, we present the entire experimental setup used to answer our research questions on the quality of the explanations of the line-level code produced by SLMs under 4B parameters in size. First, in §3.1, we detail the curated corpus of introductory Python programs from ACOS. Next, in §3.2 and §3.3, we describe the SLMs under evaluation and the prompting strategies used to elicit line-level explanations. Finally, the automated assessment procedures used in our experiments (LLM-as-a-judge and lexical overlap measures) are presented in §3.4.

#### 3.1. Data

To answer our research questions, we analyzed a curated corpus of 104 introductory Python programs from ACOS. The reference explanations in the ACOS corpus were expert-authored by instructors to ensure high pedagogical quality for CS1 students.

To better understand the dataset and its characteristics, we then annotated each problem with a knowledge component (KC) label from a 14-category taxonomy; this annotation is purely for descriptive purposes and does not feed into the evaluation pipeline. Under this scheme, each program carries exactly one primary KC and 0-3 secondary KCs. The resulting dataset, on average, has 1.48 KCs (mean 0.48 secondary KCs), and 40.4% of the data are explicitly multi-KC.

The dataset is small, but the KC distribution mirrors the pedagogical emphasis in introductory courses: there are loops ( $n=19$ ) and conditions ( $n=16$ ), which together account for one-third of all primary labels. At the pair level, the most frequent KC co-occurrences are operators+variables ( $n=8$ ), conditionals+operators ( $n=6$ ), and loops+operators ( $n=5$ ), which reflect the composition of expressions, control flow, and state updates on this dataset. The full intersection structure and 14-category taxonomy are visualized as an UpSet plot in Figure 1 in the Appendix.

Beyond the KC label statistics, as the dataset itself is for an introductory course, the programs are short and shallow; the median length is 7 lines of code (mean 7.60, SD 4.38, IQR 4–10, range 1–26). However, this length is far from uniform across KCs: OOP programs are by far the longest and most deeply nested, while I/O, tuples, strings, dictionaries, list\_ops, and variables are short and flat exercises. For evaluation purposes, we only use the annotated lines, which on average cover 83.0% of the lines of each program. Across the 104 programs, we evaluated a total of 656 individual annotated lines, representing the core logic of the exercises.

#### 3.2. Models

We focus on recent state-of-the-art SLMs. In our evaluations, we used the Qwen-3.5 [32] family models (0.8B, 2B, and 4B scales), Llama-3.2-1B [33], and SmolLM-3B [34]. This selection gives us a coherent size evolution from  $\leq 1$ B to 4B parameters and, at the same time, allows us to compare the effect of increasing model size within a single family. All SLMs under evaluation are run at full precision.

For completeness, we compare the results for SLMs with a popular medium-sized model, Llama-3.1-8B. We selected Llama because of its widespread adoption in AI-in-education research and its competitive performance on the 8B scale. Unlike the SLMs, we quantize Llama3.1-8B to 4 bits [35] in order to imitate realistic deployment setups, since medium-sized models can also run on consumer laptops once quantized.

### 3.3. Prompting Approach

We compare three prompting strategies that vary along two axes. The first axis is whether the model sees the surrounding program (Context-Aware) or only the target line (Line-Only). The second axis, applied only within the Context-Aware setting, is whether the prompt contains a single worked demonstration (one-shot) or none (zero-shot). The three resulting strategies are LO-0, CA-0, and CA-1. Full prompt templates can be found in Figure 2 in the Appendix. The system message is identical across all three strategies. It casts the model as a Python instructor writing line-by-line explanations for students in their first programming course. The LO-0 strategy is a zero-shot strategy that provides only the target line in the user message, without any surrounding program. This setup mirrors the input format used in much of the code summarization and code explanation literature, where models operate on isolated snippets [18, 19]. We use it as a controlled baseline to isolate the effect of program context. The CA-0 strategy is also zero-shot, but the user message provides the full Python program together with the target line, since the surrounding program is the information a CS1 student has when reading a line. The CA-1 strategy shares the Context-Aware framing of CA-0 and adds a single demonstration program before the target query. The demonstration is serialized in conversational format, with each annotated line of the program rendered as a (user, assistant) turn that follows the same template. The demonstration is the program whose number of lines is closest to the dataset median, and it is reused for every query, which makes the strategy topic-agnostic. This serialization makes the expected output format and answer style explicit through an example, rather than only through the system message, and steers the model toward the answer style that we wish to obtain.

### 3.4. Evaluation

To assess the effectiveness of SLMs across different prompting approaches for generating comments on code lines, we focus on two key measures: LLM-as-a-judge and automated reference-based metrics.

**LLM-as-a-judge.** We define a grading rubric for this LLM-as-a-judge evaluation using five criteria: Correctness (*Cor*), Clarity (*Cla*), Conciseness (*Con*), Pedagogical Completeness (*Ped*), and Abstraction Level (*Abs*). To score each criterion effectively, we rely on the G-Eval framework [21]. Rather than prompting a judge model for a direct integer score, G-Eval guides the model through a structured, step-by-step evaluation process before eliciting a score refined using probability-weighted aggregation, which has been shown to yield higher correlation with human judgments than direct scoring. We refine this framework for our purpose. Here, we refer to our reference-guided variant as G-Eval-Ref.

Each criterion is evaluated independently using a separate prompt. Each prompt provides the judge with a set of evaluation steps specific to that criterion. Although G-Eval supports automatic generation of evaluation steps from a high-level criterion description, we define ours manually to ensure consistency and make sure they are appropriate for our educational context. Each criterion was scored on a three-point scale (0, 1, or 2). As an addition to the original framework, our prompt also specifies, alongside the evaluation steps, which score to assign based on the observed quality level. Similarly, unlike the original G-Eval formulation, which is reference-free, we adopt a reference-guided grading scheme: the judge receives the human-annotated reference explanation alongside the generated output, as reference access consistently improves scoring reliability [36]. Finally, rather than treating the model’s discrete output as the final score, G-Eval-Ref uses the log-probabilities of each candidate score token (0, 1, 2) to compute a probability-weighted sum, yielding a continuous value in  $[0, 2]$ . We normalize this to  $[0, 1]$  by dividing by 2, obtaining a final continuous quality score for each criterion. We note that reference-guided grading introduces a known tension. By showing the judge the human annotation, the scheme improves scoring reliability, but it can also reward outputs that converge on the surface form and length of the reference rather than on independent pedagogical value. We therefore interpret per-dimension movements that are tightly coupled to reference style, in particular Conciseness, with additional caution. Table 2 (Appendix A.4) summarizes the metrics used as well as the three-point Likert-like scale adapted to this. Appendix A.4 also details each individual prompt for each criterion.

We use 4-bit quantized Gemma-4-31B<sup>3</sup> as our LLM-judge. We chose a judge from a different model family than the SLMs under evaluation in order to avoid the same-family bias and to reduce the risk of overlapping pretraining-data effects.

**Automated reference-based metrics.** For completeness, we also report two traditional lexical metrics (i.e., ROUGE-L [37] and BLEU-1 [38]) that quantify the surface-level overlap between the generated text and the reference. Both are standard measures in code summarization and natural language generation. We specifically use BLEU-1 rather than higher-order n-gram variants because our references are single sentences, where higher-order n-gram statistics become sparse and unreliable, and we use the standard ROUGE-L formulation based on the longest common subsequence. We additionally report CodeBERTScore-F1 [39], a contextual embedding-based metric tailored to code that compares the generated and reference texts in the representation space of a pretrained code model. We acknowledge that all three metrics have known limitations for evaluating natural language descriptions of code, since they reward surface or embedding similarity rather than pedagogical value, and we therefore treat them as complementary signals to the multi-dimensional G-Eval-Ref rubric described above.

## 4. Results

Table 1 reports the LLM-judge scores and automated reference-based metrics for all six checkpoints under zero-shot and one-shot prompting. We organize the discussion as a series of observations on how the judge mean and the per-dimension scores respond to model scale and to the addition of a single demonstration, and we close with the relationship between the lexical metrics and the judge mean. We start with the role of program context. Across all six checkpoints, the move from LO-0 to CA-0 improves the judge mean. The gap is largest at the smallest model, at +.142 on Qwen3.5-0.8B, and remains substantial at the larger scales, with +.083 on Qwen3.5-4B and +.080 on Llama-3.1-8B. The benefit of providing context therefore does not vanish with scale. Even an 8B model, which can plausibly infer the role of a typical Python statement from its priors, gains close to one tenth of the 0–1 judge scale once the rest of the program is shown. The gain concentrates on Pedagogical Completeness. For the four models at 2B and above, removing the context lowers Pedagogical Completeness by at least .138, with the largest drop on Qwen3.5-4B (.788 to .595, a loss of .193). This pattern matches the rubric. Pedagogical Completeness rewards explanations that connect the line to its role in the surrounding program, and this connection cannot be recovered from the line in isolation. Correctness and Abstraction Level also drop under LO-0, but the magnitude is smaller. Clarity and Conciseness are largely insensitive to context, which reflects a shorter average generation length under LO-0 rather than better calibrated phrasing, and does not translate into a better judge mean.

**Finding 1:** Providing the surrounding program context improves all six checkpoints, with the gain concentrated on Pedagogical Completeness.

Within the Qwen3.5 family, the judge mean under zero-shot prompting rises from .550 at 0.8B to .727 at 2B and .812 at 4B, a total gain of .262 across the scale. The same trend holds under one-shot prompting, where the mean moves from .690 to .781 to .829. The cross-family checkpoints are consistent with this picture. SmolLM3-3B reaches .745 zero-shot and .788 one-shot, sitting between Qwen3.5-2B and Qwen3.5-4B, while Llama-3.1-8B reaches .830 one-shot, essentially tied with Qwen3.5-4B at .829. The fact that a 4B checkpoint and an 8B checkpoint are very similar on the aggregate judge mean is the central scaling observation in our results. In addition, all six checkpoints gain on the LLM-judge mean when we move from CA-0 to CA-1, that is, when a single worked demonstration is added to the context-aware prompt. The gain ranges from +.017 on Qwen3.5-4B to +.151 on Llama-3.2-1B, with a median improvement of +.070 across models. One-shot prompting generally appears to be a safe choice on the aggregate metric for this task, though some gains remain marginal.

<sup>3</sup><https://huggingface.co/QuantTrio/gemma-4-31B-it-AWQ>

**Table 1**

Full results: models under the three prompting strategies. LO-0: line-only zero-shot. CA-0: context-aware zero-shot. CA-1: context-aware one-shot. Judge dimensions: Correctness (Cor.), Clarity (Cla.), Conciseness (Con.), Pedagogical Completeness (Ped.), Abstraction Level (Abs.). Automated metrics: CodeBERTScore-F1 (CB), BLEU-1 (B-1), ROUGE-L (R-L). Best per model per column in bold. All scores in [0, 1].

| Model                | Strategy | LLM Judge   |             |             |             |             |             | Auto. Metrics |             |             |
|----------------------|----------|-------------|-------------|-------------|-------------|-------------|-------------|---------------|-------------|-------------|
|                      |          | Cor.        | Cla.        | Con.        | Ped.        | Abs.        | Mean        | CB            | B-1         | R-L         |
| Qwen3.5-0.8B         | LO-0     | .377        | .653        | .081        | .494        | .435        | .408        | .741          | .135        | .192        |
|                      | CA-0     | .517        | .780        | .181        | <b>.670</b> | .604        | .550        | .747          | .135        | .206        |
|                      | CA-1     | <b>.653</b> | <b>.854</b> | <b>.598</b> | .614        | <b>.732</b> | <b>.690</b> | <b>.766</b>   | <b>.243</b> | <b>.296</b> |
| Llama-3.2-1B         | LO-0     | .543        | .776        | .014        | .488        | .597        | .484        | .743          | .102        | .178        |
|                      | CA-0     | .594        | .800        | .039        | .630        | .695        | .551        | .745          | .101        | .184        |
|                      | CA-1     | <b>.666</b> | <b>.836</b> | <b>.623</b> | <b>.647</b> | <b>.737</b> | <b>.702</b> | <b>.774</b>   | <b>.266</b> | <b>.319</b> |
| Qwen3.5-2B           | LO-0     | .626        | .871        | .326        | .553        | .729        | .621        | .750          | .163        | .229        |
|                      | CA-0     | .756        | <b>.946</b> | .373        | <b>.691</b> | <b>.868</b> | .727        | .759          | .175        | .247        |
|                      | CA-1     | <b>.770</b> | .943        | <b>.807</b> | .554        | .831        | <b>.781</b> | <b>.776</b>   | <b>.267</b> | <b>.335</b> |
| SmolLM3-3B           | LO-0     | .798        | .942        | .021        | .716        | .874        | .670        | .757          | .129        | .212        |
|                      | CA-0     | <b>.890</b> | <b>.979</b> | .041        | <b>.860</b> | <b>.956</b> | .745        | .764          | .148        | .234        |
|                      | CA-1     | .803        | .943        | <b>.827</b> | .556        | .812        | <b>.788</b> | <b>.783</b>   | <b>.282</b> | <b>.363</b> |
| Qwen3.5-4B           | LO-0     | .726        | .945        | .588        | .595        | .791        | .729        | .749          | .191        | .232        |
|                      | CA-0     | .785        | <b>.990</b> | .602        | <b>.788</b> | <b>.898</b> | .812        | .751          | .199        | .229        |
|                      | CA-1     | <b>.797</b> | .973        | <b>.799</b> | .698        | .878        | <b>.829</b> | <b>.768</b>   | <b>.246</b> | <b>.293</b> |
| Llama-3.1-8B (4-bit) | LO-0     | .747        | .933        | .254        | .597        | .793        | .665        | .751          | .146        | .222        |
|                      | CA-0     | <b>.857</b> | <b>.988</b> | .145        | <b>.802</b> | <b>.931</b> | .745        | .762          | .152        | .235        |
|                      | CA-1     | .836        | .979        | <b>.717</b> | .711        | .906        | <b>.830</b> | <b>.777</b>   | <b>.278</b> | <b>.324</b> |

**Finding 2:** Performance improves as model size increases, yet smaller models remain competitive with the 8B reference model. Moreover, one-shot demonstrations consistently lead to better overall quality.

Of the five rubric dimensions, Conciseness is the only one that improves consistently and substantially when a demonstration is added. Within the Qwen3.5 family, Conciseness rises from .181 to .598 at 0.8B, from .373 to .807 at 2B, and from .602 to .799 at 4B. The same pattern holds for the cross-family checkpoints. Across all six models, the absolute Conciseness gain ranges from +.197 to +.786, while no other rubric dimension shows a comparable shift in either direction. The judge-mean improvement reported above is therefore almost entirely a Conciseness improvement.

On the other hand, the Conciseness gain comes at the expense of Pedagogical Completeness. Five of the six checkpoints lose on this dimension when a demonstration is added, and the only exception is Llama-3.2-1B, which gains a marginal +.017. The drops are largest for the strongest zero-shot baselines: SmolLM3-3B falls from .860 to .556 (−.304), and Qwen3.5-2B falls from .691 to .554 (−.137). The rubric in Table 2 distinguishes between Score 2 (covers both operation and purpose in context) and Score 1 (covers operation but omits a key educational detail), and the magnitude of the drop is consistent with a shift from Score 2 to Score 1 rather than to Score 0. Demonstrations therefore make outputs shorter without removing the explanation entirely, but the brevity comes at the cost of educational connection to the surrounding program.

**Finding 3:** The demonstration benefit is concentrated in Conciseness, and this improvement is traded off against Pedagogical Completeness.

Correctness moves in opposite directions depending on the model’s zero-shot performance. The two strongest zero-shot baselines on this dimension, SmolLM3-3B (.890) and Llama-3.1-8B (.857), both lose Correctness when a demonstration is added (−.087 and −.021, respectively). The four weaker baselines instead gain on Correctness, with the largest gain at the smallest scale: +.136 for Qwen3.5-0.8B and +.072 for Llama-3.2-1B. This pattern is consistent with the interpretation that the demonstration mostly compensates for output-format and content-coverage issues that the larger and stronger models already handle implicitly, and that introducing a fixed demonstration nudges these strong baselines toward the answer style of the demonstration even when it is less accurate than what the model would produce on its own. In contrast to Correctness, Clarity, and Abstraction Level are largely insensitive to the prompting strategy. For Clarity, five of six models show an absolute change  $|\Delta| \leq .036$  between zero-shot and one-shot, and the only larger change is +.074 on Qwen3.5-0.8B. Abstraction Level shows a similar but slightly noisier picture: four of six models stay within  $|\Delta| \leq .042$ , with two outliers at the extremes of the scale, namely +.128 on Qwen3.5-0.8B and −.144 on SmolLM3-3B. The SmolLM3-3B drop mirrors the same model’s drop in Pedagogical Completeness, which suggests a strategy-specific style shift on this checkpoint rather than a general trend. Both dimensions are also close to ceiling under zero-shot prompting, with Clarity  $\geq .780$  and Abstraction Level  $\geq .604$  across all checkpoints, which leaves little headroom for the demonstration to improve them. We therefore treat these two dimensions as broadly saturated for this dataset, and the per-dimension trade-offs we discuss in the rest of the paper concentrate on Correctness, Conciseness, and Pedagogical Completeness.

**Finding 4:** In correctness, the demonstration helps the weaker baselines but hurts the two strongest zero-shot models, while Clarity and Abstraction Level remain near ceiling and are largely insensitive to the prompting strategy.

Although the direction of the demonstration effect is uniform, its magnitude is not. Within the Qwen3.5 family the one-shot gain decreases from +.140 at 0.8B to +.054 at 2B and +.017 at 4B. The 8B reference checkpoint follows a similar pattern with a gain of +.085, larger than the 4B gain because its zero-shot mean is unexpectedly low. The pattern is consistent with the interpretation that the demonstration mostly compensates for output-format issues that larger models already handle implicitly.

**Finding 5:** The aggregate benefit of demonstrations shrinks with scale.

CodeBERTScore-F1, BLEU-1, and ROUGE-L all improve from zero-shot to one-shot prompting on every checkpoint, mirroring the direction of the judge mean. However, none of the three lexical metrics reflects the negative movement on Pedagogical Completeness that the rubric isolates, and none captures the differential behavior of Correctness across model sizes. The lexical metrics are therefore a reasonable proxy for the aggregate ranking of strategies, but they are insufficient as a standalone evaluation tool when the goal is to reason about pedagogical quality. This finding supports our methodological choice of complementing lexical overlap with a multi-dimensional G-Eval-Ref rubric.

**Finding 6:** Automated metrics track the judge mean but mask per-dimension trade-offs.

## 5. Limitations

**Dataset size and scope.** Our evaluation rests on a curated corpus of 104 annotated introductory Python programs from the ACOS platform. We frame this work as a focused feasibility study rather than a large-scale benchmark, and we therefore treat the corpus as sufficient to establish a reproducible methodology that can scale to larger collections. Nonetheless, the corpus is restricted to short and shallow CS1 programs (median 7 LOC), to a single programming language, and to a single learning platform. Generalization to longer programs, to other languages, and to authentic novice code that contains syntactic or semantic errors remains an open question that a larger and more diverse corpus would be required to answer.

**Reliability of LLM-as-a-judge and reference quality.** We rely on a single judge model, Gemma-4-31B, with reference-guided grading anchored to a single human annotation per line. Although prior work shows that strong LLM judges agree with human annotators in more than 80% of pairwise comparisons on open-ended tasks [36], we do not characterize the variance across judge model choices or across decoding seeds, and inter-annotator agreement on the ACOS line annotations is not reported in this work. A small-scale human validation study on a stratified sample of the outputs, together with a second independent annotation pass, is the natural next step.

**Quantization of the 8B reference.** We quantize Llama-3.1-8B to 4 bits to mirror realistic deployment on consumer hardware. This choice is consistent with our deployment framing, but it likely lowers the absolute scores of the 8B reference model relative to a full-precision version. The observation that Qwen3.5-4B is essentially tied with Llama-3.1-8B on the aggregate judge mean should therefore be considered with this caveat.

**No learner-side validation.** Our evaluation is purely intrinsic. We assess whether the generated explanations match a pedagogically motivated rubric and a human-annotated reference, but we do not measure whether students who receive these explanations show better code comprehension, higher engagement, or improved transfer to code writing or code explanation tasks. The case for SLMs as a viable alternative to proprietary models for this task is, at this stage, an offline-quality argument rather than an evidence-based educational outcome argument. A classroom study with the strongest configurations identified here would be the most direct way to validate any practical claim.

## 6. Concluding Discussions

This study examined whether small, open-weight language models can generate pedagogically meaningful line-level explanations for introductory Python programs. Overall, the results suggest that SLMs are a promising option for this task: while explanation quality improved with model scale, the strongest sub-4B model performed comparably to the larger 8B reference model on the aggregate judge score. One-shot prompting consistently improved overall quality, but the improvement was driven mainly by conciseness rather than broad gains across all pedagogical dimensions. This is important because the same prompting strategy also tended to reduce pedagogical completeness, showing that shorter explanations are not automatically better for novice learners.

For computing education, these findings suggest that SLMs could provide scalable, privacy-preserving, and potentially on-device generation of code explanations in CS1 settings. Such models could be used to provide students with just-in-time explanations of their own code, to support reflection on completed solutions, or to generate explanation-based activities such as Parsons-style tasks and code comprehension questions. However, instructors and system designers should treat prompt design as a pedagogical choice rather than a purely technical optimization: prompts that maximize brevity may make explanations easier to read, but they may also omit connections to the surrounding program that are valuable for learning. Future classroom studies are needed to determine whether these offline quality gains translate into improved comprehension, engagement, and learning for students.

## Acknowledgments

This work was supported by Research Council of Finland grants #356114 and #367787. We acknowledge the computational resources provided by the Aalto Science-IT project.

## Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT and Grammarly for grammar and spelling check. After using these tools, the authors reviewed and edited the content as needed and take

full responsibility for the publication's content.

## References

- [1] J. Prather, P. Denny, J. Leinonen, B. A. Becker, I. Albluwi, M. Craig, H. Keuning, N. Kiesler, T. Kohn, A. Luxton-Reilly, et al., The robots are here: Navigating the generative ai revolution in computing education, in: *Proceedings of the 2023 working group reports on innovation and technology in computer science education*, 2023, pp. 108–159.
- [2] P. Denny, J. Prather, B. A. Becker, J. Finnie-Ansley, A. Hellas, J. Leinonen, A. Luxton-Reilly, B. N. Reeves, E. A. Santos, S. Sarsa, Computing education in the era of generative ai, *Communications of the ACM* 67 (2024) 56–67.
- [3] J. Finnie-Ansley, P. Denny, B. A. Becker, A. Luxton-Reilly, J. Prather, The robots are coming: Exploring the implications of openai codex on introductory programming, in: *Proceedings of the 24th Australasian Computing Education Conference, ACE '22*, Association for Computing Machinery, New York, NY, USA, 2022, p. 10–19. URL: <https://doi.org/10.1145/3511861.3511863>. doi:10.1145/3511861.3511863.
- [4] J. Leinonen, A. Hellas, S. Sarsa, B. Reeves, P. Denny, J. Prather, B. A. Becker, Using large language models to enhance programming error messages, in: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 563–569.
- [5] T. Phung, J. Cambrero, S. Gulwani, T. Kohn, R. Majumdar, A. Singla, G. Soares, Generating high-precision feedback for programming syntax errors using large language models., *Proceedings of the 16th International Conference on Educational Data Mining* (2023).
- [6] L. Lee Solano, C. Koutchme, J. Leinonen, A. Vassar, J. Renzella, Fine-tuning open-source models as a viable alternative to proprietary llms for explaining compiler messages, in: *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1, SIGCSE TS 2026*, Association for Computing Machinery, New York, NY, USA, 2026, p. 617–623. URL: <https://doi.org/10.1145/3770762.3772576>. doi:10.1145/3770762.3772576.
- [7] L. Roest, H. Keuning, J. Jeuring, Next-step hint generation for introductory programming using large language models, in: *Proceedings of the 26th Australasian Computing Education Conference*, 2024, pp. 144–153.
- [8] N. C. C. Brown, P. Weill-Tessier, J. Leinonen, P. Denny, M. Kölling, Howzat? appealing to expert judgement for evaluating human and ai next-step hints for novice programmers, *ACM Trans. Comput. Educ.* 25 (2025). URL: <https://doi.org/10.1145/3737885>. doi:10.1145/3737885.
- [9] C. Koutchme, Towards open natural language feedback generation for novice programmers using large language models, in: *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research*, 2022, pp. 1–2.
- [10] C. Koutchme, N. Dainese, S. Sarsa, A. Hellas, J. Leinonen, P. Denny, Open source language models can provide feedback: Evaluating llms' ability to help students using gpt-4-as-a-judge, in: *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1, ITiCSE 2024*, Association for Computing Machinery, New York, NY, USA, 2024, p. 52–58. URL: <https://doi.org/10.1145/3649217.3653612>. doi:10.1145/3649217.3653612.
- [11] C. Koutchme, N. Dainese, S. Sarsa, A. Hellas, J. Leinonen, S. Ashraf, P. Denny, Evaluating language models for generating and judging programming feedback, in: *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1, SIGCSETS 2025*, Association for Computing Machinery, New York, NY, USA, 2025, p. 624–630. URL: <https://doi.org/10.1145/3641554.3701791>. doi:10.1145/3641554.3701791.
- [12] N. Kiesler, D. Lohr, H. Keuning, Exploring the potential of large language models to generate formative programming feedback, in: *2023 IEEE Frontiers in Education Conference (FIE)*, 2023, pp. 1–5. doi:10.1109/FIE58773.2023.10343457.
- [13] S. Sarsa, P. Denny, A. Hellas, J. Leinonen, Automatic generation of programming exercises and

- code explanations using large language models, in: Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1, 2022, pp. 27–43.
- [14] J. Leinonen, P. Denny, S. MacNeil, S. Sarsa, S. Bernstein, J. Kim, A. Tran, A. Hellas, Comparing code explanations created by students and large language models, in: Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, ITiCSE '23, ACM, 2023. doi:10.1145/3587102.3588785.
  - [15] Y. Abdelsalam, N. Peitek, A. Bergum, S. Apel, The effect of comments on program comprehension: An eye-tracking study, *Empirical Software Engineering* 31 (2026) 94.
  - [16] K. Figl, M. Kirchner, S. Baltes, M. Felderer, The influence of code comments on the perceived helpfulness of stack overflow posts, *Empirical Software Engineering* 30 (2025) 178.
  - [17] C. Koutchme, J. Woodrow, C. Piech, Aligning small language models for programming feedback: Towards scalable coding support in a massive global course, in: Proceedings of the 57th ACM Technical Symposium on Computer Science Education V. 1, 2026, pp. 610–616.
  - [18] A. LeClair, C. McMillan, Recommendations for datasets for source code summarization, in: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), 2019, pp. 3931–3937.
  - [19] P. Bhattacharya, R. Gupta, Selective shot learning for code explanation, in: O. Arviv, M. Clinciu, K. Dhole, R. Dror, S. Gehrmann, E. Habba, I. Itzhak, S. Mille, Y. Perlitz, E. Santus, J. Sedoc, M. Shmueli Scheuer, G. Stanovsky, O. Tafjord (Eds.), Proceedings of the Fourth Workshop on Generation, Evaluation and Metrics (GEM<sup>2</sup>), Association for Computational Linguistics, Vienna, Austria and virtual meeting, 2025, pp. 151–160. URL: <https://aclanthology.org/2025.gem-1.12/>.
  - [20] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, X. Liao, Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–13.
  - [21] Y. Liu, D. Iter, Y. Xu, S. Wang, R. Xu, C. Zhu, G-Eval: NLG evaluation using GPT-4 with better human alignment, in: Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Singapore, 2023, pp. 2511–2522. doi:10.18653/v1/2023.emnlp-main.153.
  - [22] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, et al., A multi-national study of reading and tracing skills in novice programmers, *Acm sigcse bulletin* 36 (2004) 119–150.
  - [23] C. Schulte, Block model: an educational model of program comprehension as a tool for a scholarly approach to teaching, in: Proceedings of the fourth international workshop on computing education research, 2008, pp. 149–160.
  - [24] R. Hosseini, K. Akhuseyinoglu, P. Brusilovsky, L. Malmi, K. Pollari-Malmi, C. Schunn, T. Sirkiä, Improving engagement in program construction examples for learning Python programming, *International Journal of Artificial Intelligence in Education* 30 (2020) 299–336. doi:10.1007/s40593-020-00197-0.
  - [25] P. Oli, R. Banjade, A. B. Lekshmi Narayanan, J. Chapagain, L. J. Tamang, P. Brusilovsky, V. Rus, Improving code comprehension through scaffolded self-explanations, in: Artificial Intelligence in Education. Posters and Late Breaking Results, Workshops and Tutorials, Industry and Innovation Tracks, Practitioners, Doctoral Consortium and Blue Sky, volume 1831 of *Communications in Computer and Information Science*, Springer, Cham, 2023, pp. 478–483. doi:10.1007/978-3-031-36336-8\_74.
  - [26] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, M. Brockschmidt, Codesearchnet challenge: Evaluating the state of semantic code search, arXiv preprint arXiv:1909.09436 (2019).
  - [27] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, K. Vijay-Shanker, Automatic generation of natural language summaries for java classes, in: 2013 21st International conference on program comprehension (ICPC), IEEE, 2013, pp. 23–32.
  - [28] P. W. McBurney, C. McMillan, Automatic source code summarization of context for java methods, *IEEE Transactions on Software Engineering* 42 (2015) 103–119.

- [29] M. Mahade Hasan, M. Waseem, K.-K. Kemell, J. Rasku, J. Ala-Rantala, P. Abrahamsson, Assessing small language models for code generation: An empirical study with benchmarks, *Journal of Systems and Software* 236 (2026) 112815. URL: <https://www.sciencedirect.com/science/article/pii/S016412122600049X>. doi:<https://doi.org/10.1016/j.jss.2026.112815>.
- [30] S. Liu, Z. Yu, F. Huang, Y. Bulbulia, A. Bergen, M. Liut, Can small language models with retrieval-augmented generation replace large language models when learning computer science?, in: *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1, ITiCSE 2024*, Association for Computing Machinery, New York, NY, USA, 2024, p. 388–393. URL: <https://doi.org/10.1145/3649217.3653554>. doi:10.1145/3649217.3653554.
- [31] Z. Yu, S. Liu, P. Denny, A. Bergen, M. Liut, Integrating small language models with retrieval-augmented generation in computing education: Key takeaways, setup, and practical insights, in: *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1, SIGCSETS 2025*, Association for Computing Machinery, New York, NY, USA, 2025, p. 1302–1308. URL: <https://doi.org/10.1145/3641554.3701844>. doi:10.1145/3641554.3701844.
- [32] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, et al., Qwen3 technical report, arXiv preprint arXiv:2505.09388 (2025).
- [33] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, et al., The llama 3 herd of models, arXiv preprint arXiv:2407.21783 (2024).
- [34] E. Bakouch, L. Ben Allal, A. Lozhkov, N. Tazi, L. Tunstall, C. M. Patiño, E. Beeching, A. Roucher, A. J. Reedi, Q. Gallouédec, K. Rasul, N. Habib, C. Fourrier, H. Kydlicek, G. Penedo, H. Larcher, M. Morlon, V. Srivastav, J. Lochner, X.-S. Nguyen, C. Raffel, L. von Werra, T. Wolf, SmolLM3: smol, multilingual, long-context reasoner, <https://huggingface.co/blog/smollm3>, 2025.
- [35] T. Dettmers, A. Pagnoni, A. Holtzman, L. Zettlemoyer, Qlora: efficient finetuning of quantized llms, in: *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23*, Curran Associates Inc., Red Hook, NY, USA, 2023.
- [36] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, I. Stoica, Judging llm-as-a-judge with mt-bench and chatbot arena, in: *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23*, Curran Associates Inc., Red Hook, NY, USA, 2023.
- [37] C.-Y. Lin, ROUGE: A package for automatic evaluation of summaries, in: *Text Summarization Branches Out*, Association for Computational Linguistics, Barcelona, Spain, 2004, pp. 74–81. URL: <https://aclanthology.org/W04-1013>.
- [38] K. Papineni, S. Roukos, T. Ward, W.-J. Zhu, Bleu: a method for automatic evaluation of machine translation, in: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.
- [39] S. Zhou, U. Alon, S. Agarwal, G. Neubig, Codebertscore: Evaluating code generation with pretrained models of code, in: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 13921–13937.

## A. Implementation Details

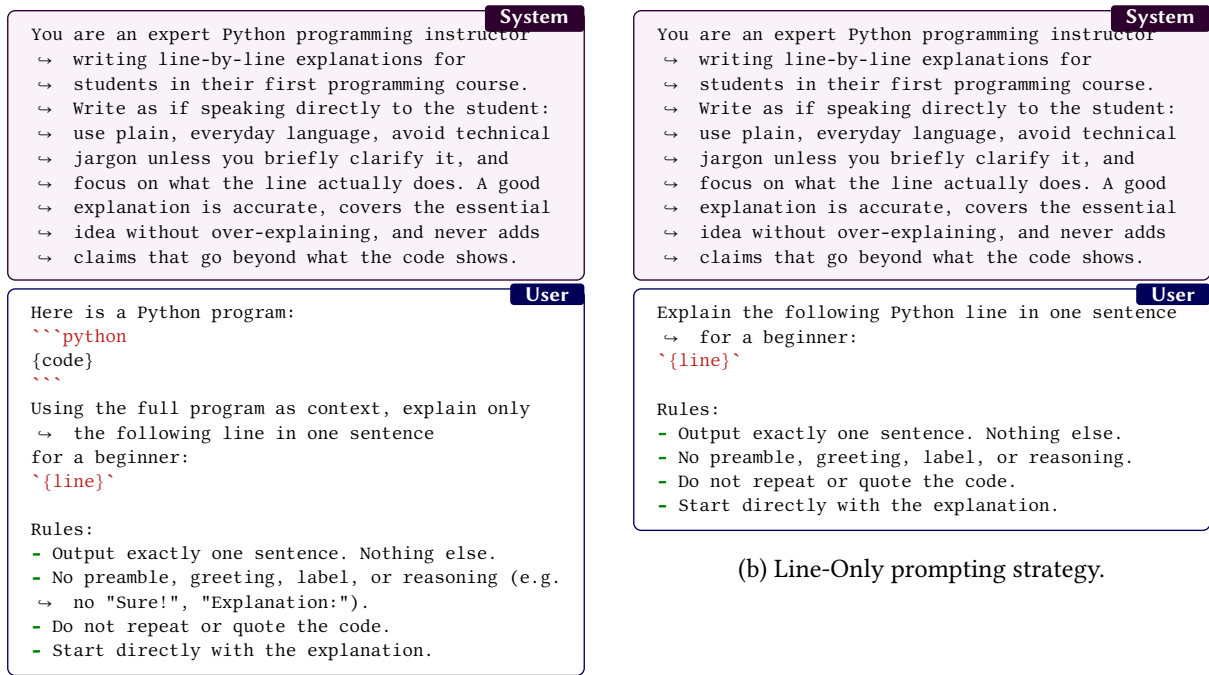
### A.1. Knowledge Component Distribution in the Data

Figure 1 shows the distribution of knowledge components per exercise.

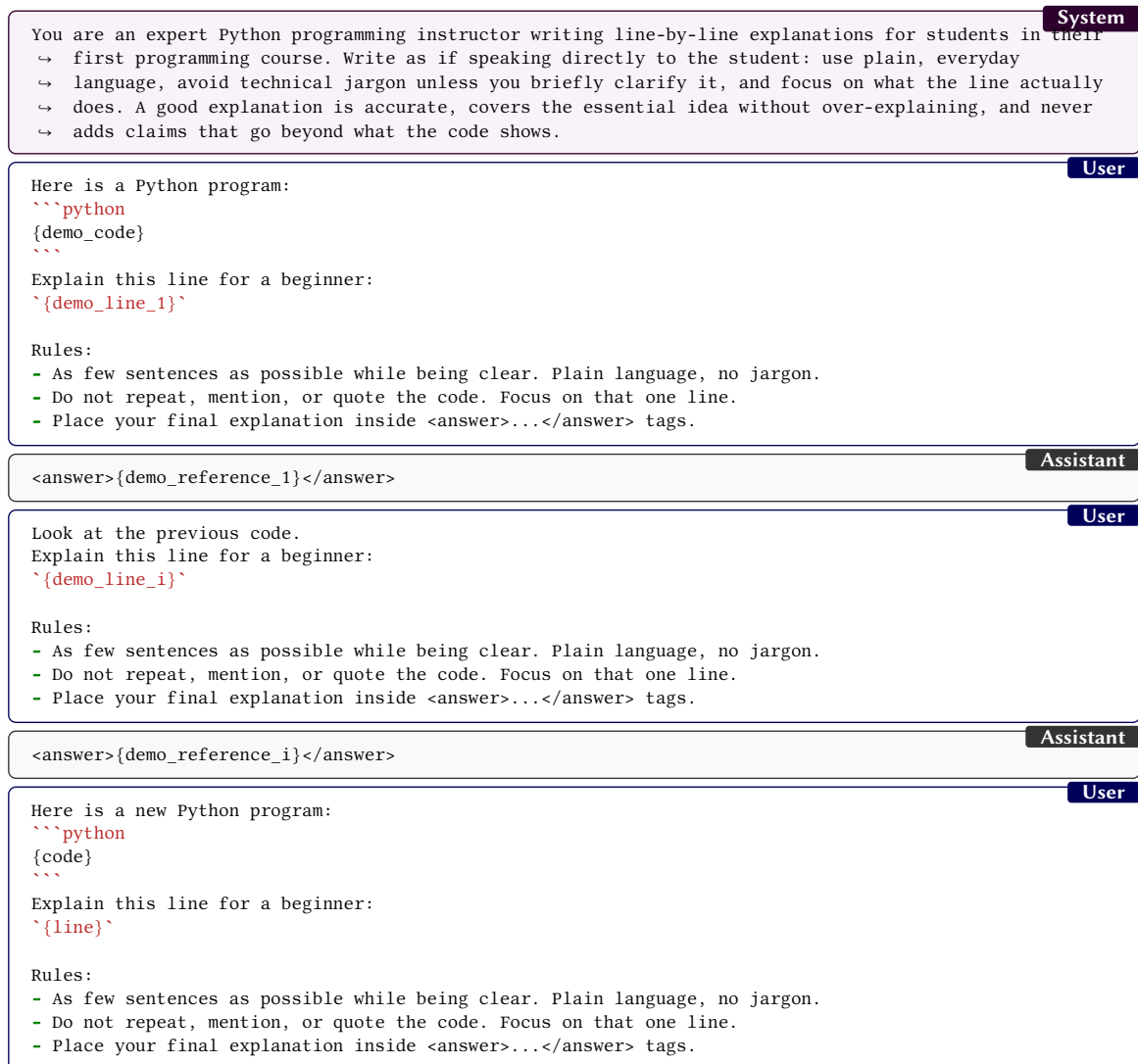
### A.2. Example Data and Annotation

Sample `pyt1_4` below illustrates how an example program is encoded for our experiments.





(a) Context-Aware (zero-shot) prompting strategy.



(c) Context-Aware (one-shot) prompting strategy.

Figure 2: Prompting strategies used in our experiments.

#### A.4. G-Eval custom metrics, criteria, and evaluation steps.

**Table 2**

Evaluation rubric for GEVAL explanation quality using a 3-point scale.

| Metric                   | Abbrev.    | Score 0 – Fail                                                                                                   | Score 1 – Partial                                                                     | Score 2 – Pass                                                                             |
|--------------------------|------------|------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| Correctness              | <i>Cor</i> | Contains a hallucination (i.e., a fabricated or unsupported claim like a wrong variable or incorrect behaviour). | Factually accurate but omits a key detail present in the reference.                   | Fully accurate; captures all essential information. Minor phrasing differences acceptable. |
| Clarity                  | <i>Cla</i> | Contains unexplained jargon or ambiguity that would actively mislead a CS1 student.                              | Understandable but requires effort; one unclear term or minor doubt remains.          | Immediately clear and unambiguous in plain English on the first read.                      |
| Conciseness              | <i>Con</i> | Actively harmed by length: key point buried in verbosity, or critical meaning lost to truncation.                | Acceptable length but with noticeable filler, hedging, or slight over-truncation.     | Single focused sentence; no padding, no loss of meaning.                                   |
| Pedagogical Completeness | <i>Ped</i> | Describes syntax only; no explanation of what the line does or why it exists.                                    | Explains the mechanical operation but does not connect it to its role in the program. | Covers both operation and purpose in context; complete educational value.                  |
| Abstraction Level        | <i>Abs</i> | Wrong level: unexplained advanced concepts, or too abstract to convey actionable understanding.                  | Mostly appropriate for CS1 but contains one term a first-year student might not know. | Perfectly calibrated for CS1: concrete, instructional, no unexplained advanced concepts.   |

#### Correctness (*Cor*) – evaluation prompt

##### 1 Task introduction

You are evaluating line-level Python code explanations generated by a language model. You will be given a target line of code, an automatically generated explanation, and a reference explanation written by a human annotator.

##### 2 Evaluation criterion

**Correctness (*Cor*), scored 0–2:** Does the explanation convey the same meaning as the reference without introducing unsupported claims?

##### EVALUATION STEPS

1. Check whether the explanation introduces any claims that cannot be inferred from the target line or its surrounding code (hallucination check first).
2. If a fabricated or unsupported claim is present — wrong variable name, invented behaviour, misidentified construct — assign score 0 regardless of other qualities.
3. If no fabricated claims exist, compare the explanation to the reference to determine whether the core meaning is accurately captured.
4. Penalise omissions of key details (e.g., missing the assignment target, omitting the loop variable) as score 1.

- Award score 2 only if the explanation is both accurate and complete relative to the reference.

#### INPUT

Target line: {line}  
Reference: {reference}  
Explanation: {explanation}

#### SCORING

- 0** Fabricated or unsupported claim — actively misleading.
- 1** Accurate but incomplete — omits a key detail.
- 2** Fully accurate and captures all essential information.

### Clarity (*Cla*) — evaluation prompt

#### 1 Task introduction

You are evaluating line-level Python code explanations generated by a language model. You will be given a target line of code, an automatically generated explanation, and a reference explanation written by a human annotator.

#### 2 Evaluation criterion

**Clarity (*Cla*), scored 0–2:** Is the explanation written in plain, beginner-friendly English?

#### EVALUATION STEPS

- Read the explanation and assess whether it uses plain, beginner-friendly English.
- Penalise unexplained jargon or technical terms a first-year student would not recognise.
- Penalise vague or ambiguous phrasing that could leave the student confused about what the line does.
- Reward explanations that state the concept directly and concretely.

#### INPUT

Target line: {line}  
Reference: {reference}  
Explanation: {explanation}

#### SCORING

- 0** Confusing or uses unexplained jargon — a beginner would not understand.
- 1** Mostly clear but one phrase is ambiguous or mildly technical.
- 2** Plain, concrete, and immediately understandable by a first-year student.

### Conciseness (*Con*) — evaluation prompt

#### 1 Task introduction

You are evaluating line-level Python code explanations generated by a language model. You will be given a target line of code, an automatically generated explanation, and a reference explanation written by a human annotator.

#### 2 Evaluation criterion

**Conciseness (*Con*), scored 0–2:** Is the explanation appropriately brief without losing meaning?

### EVALUATION STEPS

1. Check that the explanation is a single sentence or short phrase, matching the expected one-line format.
2. Penalise unnecessary repetition, hedging phrases, or filler that adds no informational value.
3. Penalise over-truncation: if critical meaning is sacrificed for brevity, score lower.
4. The ideal length closely matches the reference explanation.

### INPUT

Target line: {line}  
Reference: {reference}  
Explanation: {explanation}

### SCORING

- 0** Padded with filler or redundant phrasing that dilutes the message.
- 1** Acceptable length but contains minor unnecessary words.
- 2** Tight single sentence — no wasted words, nothing essential missing.

## Pedagogical Completeness (*Ped*) — evaluation prompt

### 1 Task introduction

You are evaluating line-level Python code explanations generated by a language model. You will be given a target line of code, an automatically generated explanation, and a reference explanation written by a human annotator.

### 2 Evaluation criterion

**Pedagogical Completeness (*Ped*), scored 0–2:** Does the explanation cover all concepts a CS1 student needs, including both what the line does and its role in the program?

### EVALUATION STEPS

1. Check whether the explanation conveys both *what* the line does and *why* it matters in the surrounding code.
2. Penalise explanations that only describe syntax without explaining the line's purpose or effect in context.
3. Reward explanations that connect the line to broader program logic (e.g., accumulating a sum, checking a condition, iterating a collection).
4. Compare with the reference to identify any critical educational detail that is missing.

### INPUT

Target line: {line}  
Reference: {reference}  
Explanation: {explanation}

### SCORING

- 0** Only describes syntax with no sense of purpose or program-level effect.
- 1** Covers the operation but omits a key educational detail present in the reference.
- 2** Explains both what the line does and its role in the program — everything a CS1 student needs.

## Abstraction Level (*Abs*) — evaluation prompt

### 1 Task introduction

You are evaluating line-level Python code explanations generated by a language model. You will be given a target line of code, an automatically generated explanation, and a reference explanation written by a human annotator.

### 2 Evaluation criterion

**Abstraction Level (*Abs*), scored 0–2:** Is the explanation pitched at the right conceptual level for a first-year programming student?

#### EVALUATION STEPS

1. Assess whether the explanation is appropriate for a student in their first programming course.
2. Penalise explanations that use advanced terminology (e.g., *lambda*, *list comprehension*, *recursion*) without explanation.
3. Penalise explanations so generic or abstract that they provide no actionable understanding of the specific line.
4. The ideal explanation is conceptual enough to build understanding and concrete enough to be instructional.

#### INPUT

Target line: {line}

Reference: {reference}

Explanation: {explanation}

#### SCORING

- 0** Pitched too high (unexplained advanced terms) or too low (trivially obvious) for a CS1 audience.
- 1** Mostly appropriate but slightly off in abstraction — one term or phrase misjudges the audience.
- 2** Perfectly calibrated for an introductory programming student — conceptual yet concrete.