Beyond Output Correctness: Benchmarking and Evaluating Large Language Model Reasoning in Coding Tasks

Anonymous ACL submission

Abstract

Large language models (LLMs) increasingly rely on explicit reasoning to solve coding tasks, yet evaluating the quality of this reasoning remains challenging. Existing reasoning evaluators are not designed for coding, and current benchmarks focus primarily on code generation, leaving other coding tasks largely unexplored. We introduce CodeRQ-Bench, the first benchmark for evaluating LLM reasoning quality across three coding task categories: generation, summarization, and classification. Using this benchmark, we analyze 1,069 mismatch cases from existing evaluators, identify five recurring limitations, and derive four design insights for reasoning evaluation in coding tasks. Guided by these insights, we propose VERA, a two-stage evaluator that combines evidence-grounded verification with ambiguity-aware score correction. Experiments on CodeRQ-Bench show that VERA consistently outperforms strong baselines across four datasets, improving AUCROC by up to 0.26 and AUPRC by up to 0.21. We release CodeRQ-Bench at `https://anonymous.4open.science/r/CodeRQ-Bench-F12D`, supporting future investigations.

1 Introduction

Large Language models (LLMs) are widely used for coding tasks (Roziere et al., 2023; Hou et al., 2024; Zhang et al., 2024; Sun et al., 2025; Bouzenia et al., 2025). With the rise of reasoning-enhanced models such as OpenAI's o-series (Jaech et al., 2024; El-Kishky et al., 2025), DeepSeek-R1 (Guo et al., 2025), and Chain-of-Thought (CoT) prompting (Wei et al., 2022) and its variants (Yao et al., 2023; Besta et al., 2024; Li et al., 2025), eliciting step-by-step reasoning has become a key strategy for improving performance on coding tasks.

However, the quality of such reasoning is often unreliable (Turpin et al., 2023; Lanham et al., 2023; Tanneru et al., 2024), and flawed reasoning can

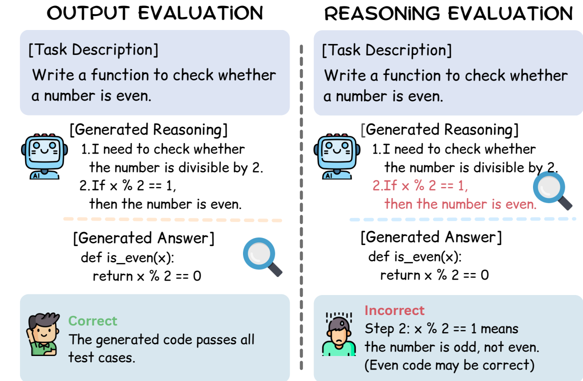

Figure 1: Output-based vs. Reasoning-based evaluation in a code generation task.

degrade the final output (Zhang et al., 2025). Despite this, existing evaluation primarily focuses on final output correctness. Benchmarks such as SWE-bench (Jimenez et al., 2024) and HumanEval (Chen et al., 2021) assess task success using outcome-based metrics (e.g., test passing, pass@k), leaving the reasoning process largely unexamined. As a result, LLMs may produce correct code while relying on faulty intermediate reasoning (Zhang et al., 2025) (Fig. 1 shows an example). Ignoring the reasoning process limits our understanding of model behavior and hinders progress toward reliable reasoning in coding tasks.

Recent work has begun to move beyond output correctness and evaluate the reasoning processes generated by LLMs. RECEVAL (Prasad et al., 2023) assesses reasoning along correctness and informativeness, SOCREVAL (He et al., 2024) uses Socratic-style prompting to evaluate reasoning, and CaSE (Do et al., 2025) applies step-level causal scoring to assess relevance and coherence. However, these methods were developed and validated on general NLP tasks, such as arithmetic reasoning and commonsense question answering. Coding tasks differ from NLP tasks substantially, as they involve program structure, semantics, and execution behavior (Liu et al., 2024; Yang et al., 2025).

Therefore, it is necessary to evaluate these methods in the coding domain to assess their effectiveness.

Another key challenge is the lack of comprehensive benchmarks for evaluating reasoning quality in coding tasks. Existing efforts focus mainly on code generation, with limited coverage of other task categories. Coding tasks are commonly grouped by output modality into generation, summarization, and classification (Lu et al., 2021; Zhang et al., 2024). These categories require different forms of reasoning (Gu et al., 2024a; Liu et al., 2024; Yuan and Zhang, 2025; Xie et al., 2025): generation involves constructive planning, summarization requires semantic abstraction and compression, and classification relies on diagnostic deduction. Restricting evaluation to generation therefore provides only a partial view of evaluator performance, obscuring task-specific limitations and hindering the development of more reliable reasoning evaluators.

To address these limitations, *(1)* we introduce **Code R**easoning **Q**uality **Bench**mark (CodeRQ-Bench), to our knowledge the first benchmark for evaluating the quality of LLM-generated reasoning across three coding task categories. Each instance is annotated for reasoning quality through a three-expert consensus-based annotation process, following established practices in dataset annotation and annotator agreement assessment (Pustejovsky and Stubbs, 2012; Teruel et al., 2018; Oortwijn et al., 2021). *(2)* Built on CodeRQ-Bench, we show that existing reasoning evaluators have substantial limitations on coding tasks. We analyze 1,069 cases where evaluators misjudge reasoning correctness, identify five recurring limitations, and derive four key insights that highlight new directions for reasoning-quality evaluation in coding tasks (see §4). *(3)* Motivated by these findings, we propose **V**erified **E**valuation of **R**easoning with **A**mbiguity-awareness (VERA), a two-stage evaluator that first verifies reasoning correctness against external evidence and then calibrates scores based on task ambiguity and how well the reasoning addresses it. On CodeRQ-Bench, VERA achieves state-of-the-art performance, improving over existing baselines by up to 0.26 in AUCROC and 0.21 in AUPRC.

2 Related Work

Outcome-Based Evaluation of Coding Tasks. Existing evaluation of coding tasks is largely outcome-based, relying on metrics such as execution correctness, exact match, and classification accuracy across tasks like code generation, repair, summarization, and bug detection (Chen et al., 2021; Du et al., 2023; Liu et al., 2023; Tian et al., 2024; Jimenez et al., 2024). While effective for measuring task success, these metrics cannot assess the reliability of the reasoning process: correct outputs may arise from flawed reasoning, and vice versa (Zhang et al., 2025). Recent benchmarks such as CRUXEval (Gu et al., 2024a), CoRe (Xie et al., 2025), CodeMind, and EquiBench (Liu et al., 2024; Wei et al., 2025) evaluate models on reasoning-intensive coding tasks but still focus on task performance as the primary metric. In contrast, our work evaluates the reasoning chain itself, enabling direct assessment of reasoning reliability and more fine-grained diagnosis of reasoning failures.

Existing Reasoning Evaluation Methods. Early work evaluates model-generated reasoning by comparing it with human-written explanations using similarity-based metrics (Clinciu et al., 2021; Welleck et al., 2022). ROSCOE extends this approach by assessing reasoning along multiple dimensions, including factuality, coherence, and redundancy (Golovneva et al., 2022). However, reference-based methods rely on predefined reasoning chains, limiting their ability to capture the diversity of valid reasoning paths.

Recent work moves toward reference-free reasoning evaluation. ReCEval (Prasad et al., 2023) evaluates reasoning in terms of correctness and informativeness; SocREval (He et al., 2024) uses Socratic-style prompting to guide LLMs through probing questions before producing an evaluation; and CaSE (Do et al., 2025) applies causal stepwise scoring to assess relevance and coherence. However, these methods were developed for general reasoning tasks, such as arithmetic and commonsense question answering, rather than coding tasks. In coding settings, reasoning quality depends not only on linguistic coherence but also on code semantics, execution behavior, and technical constraints. We address this gap by introducing a coding-oriented benchmark and evaluator.

3 CodeRQ-Bench: Code Reasoning Quality Benchmark

3.1 Preliminaries and Problem Definition

Code Reasoning Quality (CodeRQ) evaluation focuses on assessing the correctness of LLM-generated reasoning for solving a coding task. In

CodeRQ evaluation, the input consists of a coding-task description q, an LLM-generated reasoning chain $C = (s_1, \ldots, s_n)$, and the corresponding final output o. Given (q, C, o), the goal is to return a score in $[0, 1]$ indicating how likely the reasoning process is correct, where larger values indicate higher estimated reasoning quality.

3.2 Dataset Construction

To date, the only reasoning-annotated datasets for coding tasks are CoderEval-**R**easoning **E**valuation (CoderEval-RE) and SWE-bench-**R**easoning **E**valuation (SWEbench-RE), both introduced by Zhang et al. (Zhang et al., 2025). We adopt the DeepSeek-R1(Guo et al., 2025)-generated traces provided in that work (see Appx. A.1 for details). A comprehensive benchmark for CodeRQ evaluation should cover the three categories of coding tasks under the output-modality taxonomy: generation, summarization, and classification (Lu et al., 2021; Zhang et al., 2024). However, CoderEval-RE and SWEbench-RE cover only the generation category, limiting evaluation to a single modality. To address this limitation, CodeRQ-Bench builds on these two datasets and further introduces Modified-ClassEval-**R**easoning **E**valuation (ClassEval-RE) for summarization and Bug Detection-**R**easoning **E**valuation (DebugBench-RE) for classification, each annotated by three experts.

ClassEval-RE is constructed based on Modified-ClassEval (Makharev and Ivanov, 2025), a dataset obtained by reformulating ClassEval (Du et al., 2023) for code summarization. DebugBench-RE is constructed based on DebugBench (Tian et al., 2024), a code debugging benchmark in which models determine whether a code snippet contains a defect. For both datasets, we sample 139 and 252 instances for ClassEval-RE and DebugBench-RE, respectively, at a confidence level of 90% and a margin of error of 5%. For each sampled instance, we generate a reasoning trace with GPT-4o (OpenAI, 2024a), selected for its strong overall capability and favorable cost (OpenAI, 2025a). We then label its correctness with three expert annotators. More details about the construction of ClassEval-RE and DebugBench-RE, including the used source datasets and sampling procedure, see Appx. A.2 and A.3. CodeRQ-Bench's data distribution is shown in Table 1.

3.3 Data Annotation Process

To ensure annotation consistency, three expert annotators follow a consensus-based workflow,

Table 1: CodeRQ-Bench Dataset Distribution

Output Modality	Dataset	Size	# Correct / Incorrect Reasoning
Code Generation	CoderEval-RE	230	70 / 160
	SWEBench-RE	111	21 / 90
Code Summarization	ClassEval-RE	139	112 / 27
Bug Detection	DebugBench-RE	252	153 / 99

consistent with common dataset annotation practices (Pustejovsky and Stubbs, 2012; Artstein, 2017; Teruel et al., 2018; Oortwijn et al., 2021). The annotators first jointly establish guidelines for judging reasoning correctness and then independently annotate each instance. To assess reliability before adjudication, we measure inter-annotator agreement during the initial annotation phase using Fleiss' κ (Fleiss, 1971), which is suitable for three annotators. For the two newly constructed datasets, Fleiss' κ is 0.91 for ClassEval-RE and 0.95 for DebugBench-RE, with adjudication rates of 4.32% and 3.57%, respectively, indicating almost perfect agreement (Landis and Koch, 1977).

After independent annotation, annotators review cases with inconsistent labels and assign a consensus label through discussion. If disagreements reveal ambiguities in the guidelines, the rules are refined and the affected instances are re-annotated. This cycle of independent annotation, disagreement resolution, and guideline refinement continues until consensus labels are obtained for all instances.

The annotation guideline is grounded in theories of human problem solving and logical reasoning. Following Pólya's framework of understanding, planning, execution, and verification (Schoenfeld, 1987), we operationalize reasoning quality into three stage-specific dimensions: *Comprehension*, *Analysis*, and *Conclusion*. In addition, inspired by Johnson-Laird et al. (Johnson-Laird et al., 2004), which shows that inconsistency detection relies on constructing a coherent mental model, we introduce *Consistency* as a cross-cutting dimension. For each dimension, we define concrete annotation rules for each dataset. The full guidelines for ClassEval-RE and DebugBench-RE are provided in Appx. A.2.3 and Appx. A.3.3.

4 Analyzing the Limitations of Existing Reasoning Evaluators on Coding Tasks

Existing reasoning quality evaluators, including RECEVAL, SOCREVAL, and CaSE (Prasad et al., 2023; He et al., 2024; Do et al., 2025), perform poorly on coding tasks, often failing to distinguish

correct from incorrect reasoning and achieving near-random performance (see Table 3). To diagnose these limitations and inform the design of improved evaluators, we analyze cases where evaluator scores misalign with the actual reasoning correctness.

4.1 Analysis Setup and Protocol

We apply existing reasoning evaluators to CodeRQ-Bench and analyze cases where their scores misalign with the ground-truth reasoning correctness. We categorize mismatches into two types: *missed errors*, where incorrect reasoning receives a high score, and *false alarms*, where correct reasoning receives a low score. Since the existing evaluators produce continuous scores, we use the midpoint of each evaluator's score range as a threshold, treating scores above or equal to the midpoint as high and those below as low. Using this criterion, we identify 1,069 mismatch cases, including 709 *missed errors* and 360 *false alarms*.

We analyze these mismatch cases using a three-step protocol. First, for each case, we use GPT-5.2 (OpenAI, 2025b), which supports long-context, to generate a diagnostic annotation explaining why the evaluator's score diverges from the ground-truth reasoning correctness. The annotation considers the evaluator's method description, the task prompt, the LLM-generated reasoning chain and final output, and the evaluator's detailed assessment outputs. Second, GPT-5.2 extracts open-coded limitation labels from these diagnostics. Third, we consolidate these labels into higher-level categories through a two-stage process: embedding-based pre-clustering using `text-embedding-3-large` (OpenAI, 2024b), followed by taxonomy consolidation with GPT-5.2, yielding a taxonomy of evaluator limitations. All prompts are provided in Appx. B.1. To validate the LLM-based analysis, we randomly sample 65 cases (90% confidence level, 10% margin of error) for manual review; 61 cases (93.8%) contain valid diagnoses and limitation labels.

4.2 Taxonomy of Limitations in Existing Reasoning Evaluators and Design Insights

Based on the mismatch-case analysis above, we organize the observed evaluator limitations into a taxonomy of five limitation categories that cover all mismatch cases. These categories are non-exclusive, and a single mismatch case may exhibit more than one limitation. Table 2 summarizes their definitions, along with how each limitation is distributed across evaluators and mismatch types. Fig. 2 visualizes the distribution of limitation annotations across the 1,069 mismatch cases, including a breakdown by evaluation method.

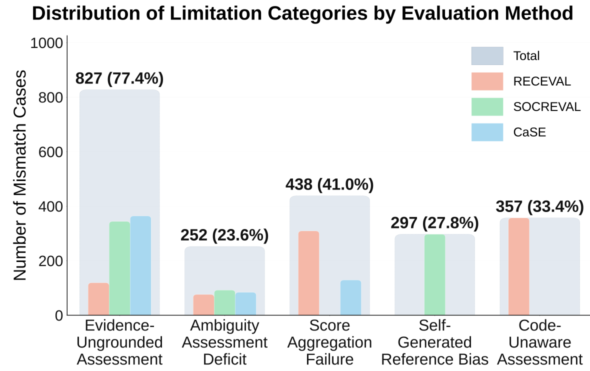

Figure 2: Distribution of five limitation categories across 1,069 mismatch cases from three evaluation methods. Background bars show aggregated totals (percentages relative to all cases; categories are non-exclusive). Foreground bars decompose each category by method.

Evidence-Ungrounded Assessment arises when evaluators score reasoning without verifying claims or assumptions that require external evidence (e.g., repository context, API documentation, or task specifications). Existing evaluators do not support such evidence retrieval and verification, and therefore tend to accept plausible but unverified reasoning as correct, which results primarily in missed errors (85.4% of the cases under this limitation).

- *Insight 1: Evidence-grounded verification.* CodeRQ evaluators should retrieve and incorporate supporting evidence during assessment (e.g., repository context, API documentation, tests/specifications) and verify whether the reasoning is consistent with that evidence, rather than scoring based on plausibility alone.

Ambiguity Assessment Deficit arises when the underlying coding problem specification is ambiguous or underspecified, in which case correct reasoning should remain conditional or explicitly uncertain rather than overly certain. Existing evaluators cannot recognize such ambiguity or assess whether the reasoning responds appropriately to it, and thus often accept overconfident reasoning, resulting primarily in missed errors (77.4%).

- *Insight 2: Ambiguity-aware assessment under underspecification.* CodeRQ evaluators should detect ambiguity or underspecification in the task specification and penalize overconfident reasoning when correctness cannot be established.

Table 2: Taxonomy of existing reasoning-quality evaluator limitations on coding tasks. For each limitation, we report its definition, the distribution of annotated mismatch cases across evaluators, and the breakdown by error type. *REC.*, *SOCR.*, and *CaSE* denote ReCEval, SocREval, and CaSE, respectively; *Missed* and *FA* denote missed errors and false alarms.

Limitation	**Definition**	**Evaluator (*REC. / SOCR. / CaSE*)**	**Error Type (*Missed / FA*)**
Evidence-Ungrounded Assessment	The evaluator does not ground its assessment in the evidence needed to verify reasoning correctness in coding tasks, where many claims must be searched and checked against repository context, API documents, or task specifications.	14.4% / 41.6% / 44.0%	85.4% / 14.6%
Ambiguity Assessment Deficit	The evaluator cannot determine whether a task is ambiguous or underspecified, nor assess whether the reasoning handles such ambiguity.	30.2% / 36.5% / 33.3%	77.4% / 22.6%
Score Aggregation Failure	The evaluator produces an unreliable final score because its scoring rule combines step-level judgments in a way that either lets one misjudged step override an otherwise correct trace or allows fatal correctness errors to be diluted by other steps.	70.5% / – / 29.5%	24.7% / 75.3%
Self-Generated Reference Bias	The evaluator judges reasoning against self-generated references or standards, creating circular validation in which shared errors or biases are mistaken for correctness.	– / 100% / –	97.3% / 2.7%
Code-Unaware Assessment	The evaluator fails to reliably assess code-related reasoning because it cannot correctly represent or judge code-relevant semantics in the reasoning process.	100% / – / –	9.0% / 91.0%

Score Aggregation Failure occurs in RECEVAL and CaSE, where the aggregation of step-level scores distorts the final judgment. In RECEVAL, min aggregation makes the score overly sensitive to a single misjudged step, whereas in CaSE, averaging can dilute correctness errors across steps.

Self-Generated Reference Bias occurs in SocREVAL, where the evaluator judges reasoning against self-generated references, creating a form of circular validation in which shared errors or biases may be mistaken for correctness. As a result, agreement with these references can reflect shared mistakes rather than true correctness, leading primarily to missed errors (97.3%).

- *Insight 3: Avoid self-referential standards.* CodeRQ evaluators should avoid using self-generated references as the primary standard for correctness; instead, they should prefer evidence-grounded checks or independently sourced verification signals to reduce circular validation and shared-error amplification.

Code-Unaware Assessment occurs in RECEVAL, whose pipeline converts reasoning steps into premise–conclusion pairs via semantic role labeling (SRL) (Shi and Lin, 2019) and evaluates entailment using a natural language inference (NLI) model (Laurer et al., 2024). Because these models are trained on general natural-language corpora and are brittle on code-mixed reasoning, RECEVAL often misinterprets code-related statements or entailment relations. As a result, many correct reasoning traces are incorrectly flagged, leading predominantly to false alarms (91.0%).

- *Insight 4: Code-aware semantic assessment.* CodeRQ evaluators should incorporate code-aware understanding when assessing reasoning, so that statements about code are evaluated based on the actual semantics and behavior of the code involved.

5 Methods

5.1 Verified Evaluation of Reasoning with Ambiguity-awareness (VERA)

Guided by the four design insights derived in §4.2, we propose VERA, a two-stage automated evaluation method for assessing the quality of reasoning in coding tasks. Given a task description q, a reasoning chain $C = (s_1, s_2, \ldots, s_n)$, and a generated output o, VERA produces a quality score in $[0, 1]$ through two stages (Fig. 3).

The first stage performs *verified reasoning evaluation*: an LLM judge scores the correctness of the reasoning trace, with access to a web search tool that it invokes autonomously when encountering technical claims requiring factual verification, and yields a base score $p \in [0, 1]$. The second stage performs *ambiguity-aware score correction*: a second LLM judge call assesses the degree of task ambiguity and how appropriately the reasoning addresses it, producing a penalty $\delta \leq 0$. The final score is $\mathrm{VERA}(q, C, o) = \max(p + \delta,\ 0)$, where the $\max$ operator ensures that the final score remains in $[0, 1]$.

5.2 Verified Reasoning Evaluation

Motivated by prior LLM-as-a-Judge studies (Zheng et al., 2023; Gu et al., 2024b), we use an LLM judge as a starting point for automated reasoning evaluation in this setting. Given a sample (q, C, o), the first stage prompts an LLM judge to assess the

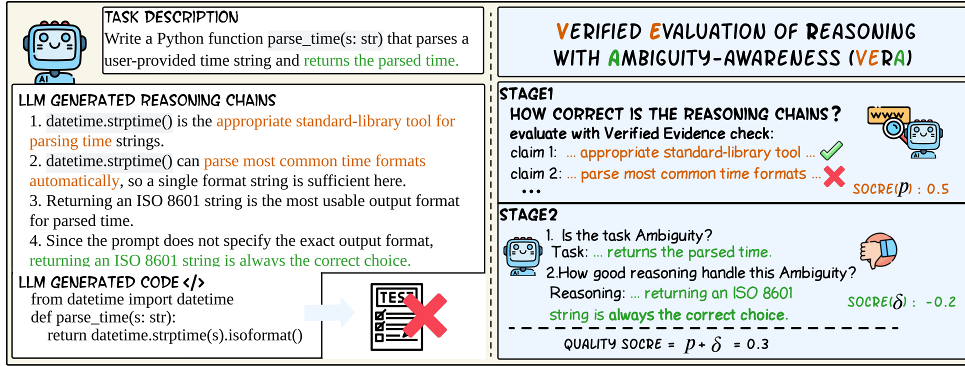

Figure 3: Overview of VERA. The left panel shows an input sample with the task description, the LLM-generated reasoning chain, and the generated code output; the right panel shows the two-stage evaluation process. Orange text marks Stage 1 evidence-verified claims, producing a base score p, and green text marks Stage 2 ambiguity-related content, producing a correction term δ. The final score combines p and δ.

overall correctness of the reasoning chain C, focusing on whether the reasoning captures the technical requirements of the task, makes technically sound claims, and supports an output that correctly addresses the problem, in line with Insight 4 (§4.2). The full evaluation prompt and rubric are provided in Appx. C.1.

To reduce reliance on the judge's parametric knowledge, the judge is given access to a search tool that invokes autonomously during evaluation, in line with Insights 1 and 3 (§4.2). When the reasoning chain contains verifiable technical claims, such as the behavior of a standard library function or the documented behavior of a framework component, the judge issues a search query to retrieve external evidence before rendering its judgment. The retrieved evidence is then used in the subsequent evaluation, allowing the judge to compare the claim against external sources before producing its final score. This mechanism shifts evaluation from pure plausibility-based assessment toward evidence-grounded verification; its empirical effect is measured via an ablation study in §6.2.1.

The judge produces an integer score $r \in [1, 10]$ according to a rubric with anchored descriptions ranging from fundamentally incorrect reasoning (r=1) to fully correct reasoning, with verifiable claims correctly supported when applicable (r=10). The base score is obtained by normalizing to the unit interval: $p = \frac{r-1}{9}$.

5.3 Ambiguity-Aware Score Correction

The base score p from Stage 1 evaluates whether the reasoning is correct and, when needed, grounds this judgment in verifiable evidence. However, like existing reasoning evaluators, it does not assess whether the task itself is ambiguous, nor whether the reasoning appropriately responds to such ambiguity. Stage 2 addresses this limitation by evaluating both the degree of task ambiguity and the quality of ambiguity handling, and applies a penalty when ambiguity is handled poorly, in line with Insight 2 (§4.2).

In Stage 2, we also use an LLM judge to produce two judgments for each sample (q, C, o): an *ambiguity level* $a \in [0, 1]$, measuring how underspecified the task q is, and a *handling quality* score $h \in [0, 1]$, measuring how appropriately the reasoning chain C handles that ambiguity. Both are elicited in a single call to ensure that they are assessed under the same interpretation of the instance. The full prompt and calibrated rubrics are provided in Appx. C.2.

The two scores are combined into a penalty δ:

$$\delta = \begin{cases} a \cdot \min(2h - 1, 0), & \text{if } a \geq \tau, \\ 0, & \text{otherwise,} \end{cases}$$

where τ is a threshold on the ambiguity level. The transformation $2h-1$ recenters h so that h=0.5 serves as the neutral point, while h<0.5 maps to a negative value. The $\min(\cdot, 0)$ operator ensures that $\delta \leq 0$: good ambiguity handling prevents penalty but does not inflate the score beyond what the correctness assessment already warrants. Scaling by a makes the penalty magnitude proportional to the degree of ambiguity. τ is set to 0.4; a sensitivity analysis is provided in Appx. D.4.

6 Experiments

6.1 Experimental Setup

Methods Compared. We compare VERA against six representative reasoning evaluation

Table 3: Performance of VERA and six baselines on the four CodeRQ-Bench datasets, evaluated using Somers' D, Spearman's ρ, AUCROC, and AUPRC (↑ higher is better). Best results highlighted in bold.

Method	**CoderEval-RE** (Generation)				**SWEbench-RE** (Generation)			
	Somers' D ↑	Spearman's ρ ↑	AUCROC ↑	AUPRC ↑	Somers' D ↑	Spearman's ρ ↑	AUCROC ↑	AUPRC ↑
RECEVAl	0.1255	0.1000 (p=0.1308)	0.5627	0.3516	0.0878	0.0596 (p=0.5344)	0.5439	0.2125
SOCREVAl	0.1399	0.1248 (p=0.0588)	0.5700	0.3410	0.1556	0.1830 (p=0.0545)	0.5778	0.2165
CaSE	0.0368	0.0392 (p=0.5543)	0.5184	0.3131	-0.0243	-0.0588 (p=0.5400)	0.4878	0.1855
MAD*	-0.0408	-0.0480 (p=0.4684)	0.4796	0.2963	-0.0746	-0.0644 (p=0.5019)	0.4627	0.1785
LLM-as-Judge	0.0864	0.1077 (p=0.1032)	0.5432	0.3252	0.0222	0.0280 (p=0.7707)	0.5111	0.1924
AutoRace*	0.0178	0.0146 (p=0.8253)	0.5089	0.3359	0.0344	0.0247 (p=0.7972)	0.5172	0.2114
VERA (Ours)	**0.3811**	**0.3076** (p<0.0001)	**0.6905**	**0.4615**	**0.2799**	**0.1947** (p=0.0406)	**0.6399**	**0.3058**

Method	**ClassEval-RE** (Summarization)				**DebugBench-RE** (Classification)			
	Somers' D ↑	Spearman's ρ ↑	AUCROC ↑	AUPRC ↑	Somers' D ↑	Spearman's ρ ↑	AUCROC ↑	AUPRC ↑
RECEVAl	-0.0218	-0.0150 (p=0.8613)	0.4891	0.8239	-0.0896	-0.0758 (p=0.2306)	0.4552	0.5844
SOCREVAl	0.1680	0.1814 (p=0.0330)	0.5840	0.8345	0.1819	0.1958 (p=0.0018)	0.5909	0.6566
CaSE	0.0397	0.0582 (p=0.4960)	0.5198	0.8120	0.3538	0.3402 (p<0.0001)	0.6769	0.7431
MAD*	0.2209	0.1588 (p=0.0619)	0.6104	0.8494	-0.0166	-0.0197 (p=0.7557)	0.4917	0.6192
LLM-as-Judge	-0.0089	-0.0418 (p=0.6252)	0.4955	0.8044	0.0530	0.0471 (p=0.4569)	0.5265	0.6262
AutoRace*	0.2500	0.1905 (p=0.0247)	0.6250	0.8502	0.2063	0.1772 (p=0.0048)	0.6032	0.7142
VERA (Ours)	**0.4180**	**0.3085** (p=0.0002)	**0.7090**	**0.8869**	**0.4351**	**0.3845** (p<0.0001)	**0.7176**	**0.7939**

baselines. **RECEVAL** (Prasad et al., 2023) is a step-level evaluator that measures intra-step correctness, inter-step correctness, and informativeness, and aggregates them into chain-level scores. **SOCREVAL** (He et al., 2024) is a Socratic-prompting-based LLM judge that assigns a discrete overall score to the full reasoning trace. **CaSE** (Do et al., 2025) is a causal stepwise evaluator that scores each reasoning step in terms of relevance and coherence using only its preceding context. **LLM-as-Judge** (Zheng et al., 2023) is a vanilla LLM-based evaluator that directly assigns an overall quality score to the reasoning trace. **MAD*** (Zhang et al., 2025) is adapted from MAD, a debate-based framework for detecting low-quality reasoning, with its original binary output converted into a continuous score in $[0, 1]$ for comparison. **AutoRace*** (Hao et al., 2024) is adapted from AutoRace, a task-adaptive automated reasoning evaluation framework that induces task-specific criteria from example incorrect reasoning chains and then uses an LLM judge for assessment, with its original binary output converted into a continuous score in $[0, 1]$ for comparison. Since MAD* relies on role-based debate and AutoRace* depends on induced task-specific criteria rather than fixed evaluator formulations, we do not include them in the limitation analysis in §4.2. Additional details of the baseline methods are provided in Appx. D.1.

Evaluation Metrics. We evaluate each reasoning evaluator using rank-based and discriminative metrics: Somers' D (Somers, 1962), Spearman's ρ (Spearman, 1961), AUCROC (Green et al., 1966), and AUPRC (Davis and Goadrich, 2006). Together, these metrics assess both ranking quality and discriminative ability. Full details are provided in Appx. D.2.

Implementation Details. RECEVAL is implemented following its original configuration with three dimensions averaged into a unified score. SOCREVAL and CaSE use GPT-4 by default and GPT-4.1 on SWEbench-RE due to context length limits. VERA, MAD*, LLM-as-Judge, and AutoRace* use GPT-4o-mini for cost efficiency. VERA uses $\tau = 0.4$ based on the sensitivity study in Appx. D.4, and all outputs are normalized to $[0, 1]$. To ensure the robustness of our findings, we repeat each experiment three times and report the average performance. Full details are in Appx. D.3.

6.2 Results

Cross-dataset comparison and the value of CodeRQ-Bench. Table 3 reports Somers' D and Spearman's ρ measuring the association between predicted scores and ground-truth correctness labels, respectively. AUCROC measures the probability that a correct reasoning trace receives a higher score than an incorrect one, while AUPRC is particularly informative under class imbalance.

The results show that baseline performance varies across coding settings. SOCREVAL is the strongest baseline on the two generation-oriented datasets, but this advantage does not generalize to

summarization and classification tasks. CaSE becomes more competitive on bug detection, whereas MAD* and AutoRace* perform relatively better on code summarization than on generation. These shifts indicate that conclusions drawn from generation-only benchmarks provide an incomplete view of evaluator performance.

Notably, even VERA does not achieve uniform performance across settings, underscoring the need for evaluation beyond generation-only benchmarks. Overall, these results highlight the importance of CodeRQ-Bench's broad task coverage for assessing reasoning evaluators in coding tasks.

Overall performance of VERA. Table 3 shows that VERA consistently achieves the best performance across all four datasets and all evaluation metrics, indicating substantially stronger alignment with reasoning correctness in both rank consistency and discriminative ability. The gains are particularly pronounced on the generation-oriented datasets. On CoderEval-RE, Somers' D increases from 0.1399 to 0.3811 and AUCROC from 0.5700 to 0.6905; on SWEbench-RE, Somers' D improves from 0.1556 to 0.2799 and AUPRC from 0.2165 to 0.3058. These results suggest that existing evaluators are particularly limited when reasoning makes factual claims about code behavior, API usage, or library semantics that cannot be verified from the reasoning text alone.

A similar advantage holds on ClassEval-RE (0.4180 Somers' D, 0.7090 AUCROC) and DebugBench-RE (0.4351 Somers' D, 0.7176 AUCROC), where CaSE is the strongest baseline yet still falls below VERA on all metrics, indicating that step-level relevance and coherence alone are insufficient for accurate reasoning evaluation.

LLM-as-Judge yields modest and sometimes unstable performance, including negative rank correlations on some datasets. MAD* is relatively competitive on ClassEval-RE (0.2209 Somers' D, 0.6104 AUCROC) but performs poorly on the generation and bug detection datasets. Together, these results show that strong reasoning evaluation does not emerge from a general-purpose LLM judge or debate-based procedure alone; it requires a design that explicitly accounts for evidence grounding and task ambiguity. Notably, despite using the more economical `GPT-4o-mini`, VERA outperforms GPT-4-based baselines by a large margin, confirming that the gains stem from the evaluation design rather than model size.

6.2.1 Ablation Study

Table 4: Ablation study of VERA on four CodeRQ-Bench datasets, including two ablated variants: *w/o* verif. (without evidence verification) and *w/o* ambig. (without ambiguity-aware correction).

Dataset	Method	Somers' D ↑	Spearman's ρ ↑	AUCROC ↑	AUPRC ↑
CoderEval-RE (Generation)	**VERA** (Ours)	**0.3811**	**0.3076** (p<0.0001)	**0.6905**	**0.4615**
	w/o verif.	0.2945	0.2369 (p=0.0003)	0.6472	0.4356
	w/o ambig.	0.3216	0.2658 (p<0.0001)	0.6608	0.4364
SWEbench-RE (Generation)	**VERA** (Ours)	**0.2799**	**0.1947** (p=0.0406)	**0.6399**	**0.3058**
	w/o verif.	-0.0376	-0.0261 (p=0.7855)	0.4812	0.1842
	w/o ambig.	0.2571	0.1796 (p=0.0593)	0.6286	0.3012
ClassEval-RE (Summarization)	**VERA** (Ours)	**0.4180**	**0.3085** (p=0.0002)	**0.7090**	**0.8869**
	w/o verif.	0.2622	0.1984 (p=0.0192)	0.6311	0.8561
	w/o ambig.	0.4134	0.3051 (p=0.0003)	0.7067	0.8864
DebugBench-RE (Classification)	**VERA** (Ours)	**0.4351**	**0.3845** (p<0.0001)	**0.7176**	**0.7939**
	w/o verif.	0.3440	0.3094 (p<0.0001)	0.6720	0.7575
	w/o ambig.	0.4184	0.3725 (p<0.0001)	0.7092	0.7887

Table 4 reports two ablated variants: *w/o* verif., in which the Stage 1 evaluator judges reasoning correctness without externally verifying verifiable technical claims, and *w/o* ambig., which removes the ambiguity-aware correction. Both consistently underperform the full model, confirming that each component is necessary. Removing evidence verification causes the largest degradation, particularly on the generation-oriented datasets: Somers' D drops from 0.3811 to 0.2945 on CoderEval-RE and from 0.2799 to -0.0376 on SWEbench-RE, with similar trends on ClassEval-RE and DebugBench-RE. This identifies evidence verification as the dominant driver of improvement, especially when reasoning makes factual claims about code behavior, API usage, or library semantics that require external grounding to assess. Removing ambiguity-aware correction yields a smaller but consistent drop, most visible on the generation-oriented datasets where task requirements are more often underspecified. Its impact on ClassEval-RE and DebugBench-RE is marginal, confirming that this component provides complementary benefits primarily under ambiguous task conditions.

7 Conclusion

We introduce CodeRQ-Bench and analyze 1,069 mismatches from existing reasoning-quality evaluators, deriving four design insights. Next, we propose VERA, a two-stage evaluator combining evidence-grounded verification with ambiguity-aware score correction. Our results show that VERA outperforms baselines across four datasets, highlighting the need for both broader benchmarks and evaluation methods beyond plausibility-based judgment to reliably assess reasoning in coding.

Limitations

Despite its contributions, this work has certain limitations. Although CodeRQ-Bench provides broad coverage across generation, summarization, and classification tasks, the current benchmark still focuses on relatively controlled evaluation settings and does not yet fully capture more realistic coding workflows, such as tool-integrated problem solving, interactive debugging, and richer long-context repository reasoning. Future work can extend the benchmark to these broader settings, enabling more comprehensive evaluation of reasoning quality in coding tasks.

Ethical considerations

This work adheres to ethical standards emphasizing transparency, reliability, and privacy in reasoning-quality evaluation for coding tasks. By openly releasing CodeRQ-Bench, reporting our empirical analysis of existing evaluator limitations, and introducing VERA, this work provides a standardized foundation for advancing more reliable assessment of LLM reasoning in coding tasks. All benchmark instances are derived from publicly available datasets and contain no personally identifiable information. External models and APIs are used in accordance with their respective terms of service. Additionally, we used ChatGPT exclusively to improve minor grammar in the final manuscript text.

Broader Impacts

The benchmark and evaluator proposed in this paper provide a benchmark foundation for reasoning-quality evaluation in coding tasks. By standardizing the evaluation setting and moving beyond final-output correctness, this work supports progress in reliable coding assistants, reasoning evaluation methods, and trustworthy LLM research for software engineering. The benchmark promotes transparency, reproducibility, and further innovation, ultimately contributing to more reliable and responsible use of LLMs in coding-related applications.

References

Ron Artstein. 2017. *Inter-annotator Agreement*, pages 297–313. Springer Netherlands, Dordrecht.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and 1 others. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 17682–17690.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. Repairagent: An autonomous, llm-based agent for program repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2188–2200. IEEE.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Miruna Clinciu, Arash Eshghi, and Helen Hastie. 2021. A study of automatic metrics for the evaluation of natural language explanations. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 2376–2387.

Jesse Davis and Mark Goadrich. 2006. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240.

Heejin Do, Jaehui Hwang, Dongyoon Han, Seong Joon Oh, and Sangdoo Yun. 2025. What defines good reasoning in llms? dissecting reasoning steps with multi-aspect evaluation. *arXiv preprint arXiv:2510.20603*.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*.

Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaiev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, and 1 others. 2025. Competitive programming with large reasoning models. *arXiv preprint arXiv:2502.06807*.

Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378.

Olga Golovneva, Moya Chen, Spencer Poff, Martin Corredor, Luke Zettlemoyer, Maryam Fazel-Zarandi, and Asli Celikyilmaz. 2022. Roscoe: A suite of metrics for scoring step-by-step reasoning. *arXiv preprint arXiv:2212.07919*.

David Marvin Green, John A Swets, and 1 others. 1966. *Signal detection theory and psychophysics*, volume 1. Wiley New York.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. 2024a. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, and 1 others. 2024b. A survey on llm-as-a-judge. *The Innovation*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Shibo Hao, Yi Gu, Haotian Luo, Tianyang Liu, Xiyan Shao, Xinyuan Wang, Shuhua Xie, Haodi Ma, Adithya Samavedhi, Qiyue Gao, and 1 others. 2024. New evaluation, library, and analysis of step-by-step reasoning with large language models. *arXiv preprint arXiv:2404.05221*.

Hangfeng He, Hongming Zhang, and Dan Roth. 2024. Socreval: Large language models with the socratic method for reference-free reasoning evaluation. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 2736–2764.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, and 1 others. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? *Preprint*, arXiv:2310.06770.

Philip N Johnson-Laird, Vittorio Girotto, and Paolo Legrenzi. 2004. Reasoning from inconsistency to consistency. *Psychological Review*, 111(3):640.

J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174.

Tamera Lanham, Anna Chen, Ansh Radhakrishnan, Benoit Steiner, Carson Denison, Danny Hernandez, Dustin Li, Esin Durmus, Evan Hubinger, Jackson Kernion, and 1 others. 2023. Measuring faithfulness in chain-of-thought reasoning. *arXiv preprint arXiv:2307.13702*.

Moritz Laurer, Wouter Van Atteveldt, Andreu Casas, and Kasper Welbers. 2024. Less annotating, more classifying: Addressing the data scarcity issue of supervised machine learning with deep transfer learning and bert-nli. *Political Analysis*, 32(1):84–100.

Yuangang Li, Yiqing Shen, Yi Nian, Jiechao Gao, Ziyi Wang, Chenxiao Yu, Shawn Li, Jie Wang, Xiyang Hu, and Yue Zhao. 2025. Mitigating hallucinations in large language models via causal reasoning. *arXiv preprint arXiv:2508.12495*.

Changshu Liu, Yang Chen, and Reyhaneh Jabbarvand. 2024. Codemind: Evaluating large language models for code reasoning. *arXiv preprint arXiv:2402.09664*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems*, 36:21558–21572.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, and 1 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Vladimir Makharev and Vladimir Ivanov. 2025. Code summarization beyond function level. *Preprint*, arXiv:2502.16704.

Yvette Oortwijn, Thijs Ossenkoppele, and Arianna Betti. 2021. Interrater disagreement resolution: A systematic procedure to reach consensus in annotation tasks. In *Proceedings of the workshop on human evaluation of NLP systems (HumEval)*, pages 131–141.

OpenAI. 2024a. Gpt-4o system card.

OpenAI. 2024b. New embedding models and api updates.

OpenAI. 2025a. Api pricing. `https://developers.openai.com/api/docs/pricing?legacy-pricing=standard`.

OpenAI. 2025b. Introducing gpt-5.2. Accessed: 2026-03-06.

Archiki Prasad, Swarnadeep Saha, Xiang Zhou, and Mohit Bansal. 2023. Receval: Evaluating reasoning chains via correctness and informativeness. *arXiv preprint arXiv:2304.10703*.

James Pustejovsky and Amber Stubbs. 2012. *Natural Language Annotation for Machine Learning: A guide to corpus-building for applications*. " O'Reilly Media, Inc.".

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Alan H Schoenfeld. 1987. Pólya, problem solving, and education. *Mathematics magazine*, 60(5):283–291.

Peng Shi and Jimmy Lin. 2019. Simple bert models for relation extraction and semantic role labeling. *arXiv preprint arXiv:1904.05255*.

Robert H Somers. 1962. A new asymmetric measure of association for ordinal variables. *American sociological review*, pages 799–811.

Charles Spearman. 1961. The proof and measurement of association between two things.

Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2025. Source code summarization in the era of large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1882–1894. IEEE.

Sree Harsha Tanneru, Dan Ley, Chirag Agarwal, and Himabindu Lakkaraju. 2024. On the hardness of faithful chain-of-thought reasoning in large language models. *arXiv preprint arXiv:2406.10625*.

Milagro Teruel, Cristian Cardellino, Fernando Cardellino, Laura Alonso Alemany, and Serena Villata. 2018. Increasing argument annotation reproducibility by using inter-annotator agreement to improve guidelines. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Hui Haotian, Liu Weichuan, Zhiyuan Liu, and 1 others. 2024. Debugbench: Evaluating debugging capability of large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 4173–4198.

Miles Turpin, Julian Michael, Ethan Perez, and Samuel Bowman. 2023. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. *Advances in Neural Information Processing Systems*, 36:74952–74965.

Anjiang Wei, Jiannan Cao, Ran Li, Hongyu Chen, Yuhui Zhang, Ziheng Wang, Yaofeng Sun, Yuan Liu, Thiago SFX Teixeira, Diyi Yang, and 1 others. 2025. Equibench: Benchmarking code reasoning capabilities of large language models via equivalence checking. *arXiv e-prints*, pages arXiv–2502.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Sean Welleck, Jiacheng Liu, Ximing Lu, Hannaneh Hajishirzi, and Yejin Choi. 2022. Naturalprover: Grounded mathematical proof generation with language models. *Advances in Neural Information Processing Systems*, 35:4913–4927.

Danning Xie, Mingwei Zheng, Xuwei Liu, Jiannan Wang, Chengpeng Wang, Lin Tan, and Xiangyu Zhang. 2025. Core: Benchmarking llms code reasoning capabilities through static analysis tasks. *arXiv preprint arXiv:2507.05269*.

Dayu Yang, Tianyang Liu, Daoan Zhang, Antoine Simoulin, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, Xin Qian, Grey Yang, Jiebo Luo, and 1 others. 2025. Code to think, think to code: A survey on code-enhanced reasoning and reasoning-driven code intelligence in llms. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 2586–2616.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12.

Haohan Yuan and Haopeng Zhang. 2025. Understanding llm reasoning for abstractive summarization. *arXiv preprint arXiv:2512.03503*.

Binquan Zhang, Li Zhang, Zhiwen Luo, Yuxin Du, Fang Liu, Song Wang, and Lin Shi. 2025. Are they all good? evaluating the quality of cots in llm-based code generation. *arXiv preprint arXiv:2507.06980*.

Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2024. A survey on large language models for software engineering. *Preprint*, arXiv:2312.15223.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, and 1 others. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623.

Supplementary Material

A CodeRQ-Bench's Dataset Details

A.1 CoderEval-RE and SWEbench-RE

Zhang et al. (Zhang et al., 2025) evaluate three LLMs on CoderEval (Yu et al., 2024) and SWE-bench (Jimenez et al., 2024), providing expert annotations that verify the correctness of LLM-generated reasoning traces.

Due to the partial release of the original experimental data, we use the expert-annotated reasoning traces generated by DeepSeek-R1 (Guo et al., 2025) to ensure the completeness and reproducibility of our baseline.

CoderEval-RE CoderEval (Yu et al., 2024) is a benchmark designed for pragmatic code generation, containing real-world coding tasks extracted from open-source projects. CoderEval-RE extends this benchmark with reasoning trace annotations, enabling evaluation of the logical correctness of LLM reasoning processes during code generation.

SWEbench-RE SWE-bench (Jimenez et al., 2024) consists of real GitHub issues and their corresponding pull requests from popular Python repositories. SWEbench-RE augments this benchmark with expert-annotated reasoning traces, allowing assessment of whether LLMs reason correctly when resolving software engineering tasks.

A.2 ClassEval-RE Dataset Details

A.2.1 Source Dataset - Modified-ClassEval

ClassEval (Du et al., 2023) is a class-level Python code generation benchmark consisting of 100 manually constructed class-level programming tasks, covering 100 classes and 410 methods. Modified-ClassEval adapts this benchmark for the code summarization setting by extracting the contextual information, function implementation, and corresponding natural language summary for each method.

A.2.2 Construction Details

To construct ClassEval-RE, we first filtered the 410 methods to remove cases containing fewer than five lines of code, as such trivial implementations typically do not require substantial reasoning for summarization. This filtering step resulted in a pool of 282 candidate methods. From this pool, we randomly sampled instances using a confidence level of 90% and a margin of error of 5%, yielding a final dataset of 139 methods. For the sampled instances, we use GPT-4o to generate reasoning traces. The correctness of these reasoning traces is then annotated by three expert annotators following the annotation process described in § 3.3.

A.2.3 Annotation Rules

Table A1 summarizes the dimension-specific annotation rules for ClassEval-RE.

Table A1: Annotation rules for ClassEval-RE (Code Summarization) across four evaluation dimensions.

Dimension: Comprehension	
R1	The reasoning must correctly identify the input parameters and return values of the code.
R2	The reasoning must accurately trace the control flow (loops, conditionals, branches) of the code.
R3	The reasoning must correctly interpret the operations performed on data structures.
R4	The reasoning must not attribute functionality that does not exist in the code.
Dimension: Analysis	
R1	The reasoning must correctly map variable names and function calls to their semantic roles.
R2	The reasoning must abstract implementation details to appropriate conceptual descriptions.
R3	The reasoning must correctly identify the algorithmic or design pattern employed in the code.
Dimension: Conclusion	
R1	The reasoning must address the primary functionality of the code.
R2	The reasoning must cover all significant code branches and edge cases that affect the summary.
R3	The reasoning must not include irrelevant analysis that does not contribute to understanding the code's purpose.
Dimension: Consistency	
R1	The reasoning must not contain self-contradictory statements about code behavior.
R2	Each reasoning step must logically follow from the previous step or from the code itself.
R3	The final summary must be logically derivable from the reasoning steps presented.

A.3 DebugBench-RE Dataset Details

A.3.1 Source Dataset - DebugBench

DebugBench (Tian et al., 2024) is a bug detection benchmark composed of LeetCode-style code snippets in which bugs are injected using GPT-4 and subsequently verified by both human annotators and LLMs. The dataset contains both correct and buggy programs spanning four bug categories: syn-

tax errors, reference errors, logic errors, and multiple errors (a combination of the previous three categories).

A.3.2 Construction Details

To construct DebugBench-RE, we filtered the dataset to remove instances labeled solely as syntax errors, as detecting syntax errors is largely trivial and does not require substantive reasoning. After filtering, 3,492 instances remained. From this pool, we randomly sampled examples using a confidence level of 90% and a margin of error of 5%, resulting in a final dataset of 252 methods, consisting of 126 bug-free and 126 buggy programs. For the sampled instances, we use GPT-4o to generate reasoning traces. The correctness of these reasoning traces is then annotated by three expert annotators following the annotation process described in § 3.3.

A.3.3 Annotation Rules

Table A2 summarizes the dimension-specific annotation rules for DebugBench-RE.

Table A2: Annotation rules for DebugBench-RE (Bug Detection) across four evaluation dimensions.

Dimension: Comprehension	
R1	The reasoning must correctly identify parameters, their types (if inferable), and their roles where bugs are identified.
R2	The reasoning must correctly identify the return value(s), including affected conditionals where bugs are identified.
R3	The reasoning must correctly describe how bug-related data structures (e.g., lists, maps, trees) are read, modified, or traversed.
Dimension: Analysis	
R1	All bugs that affect observable behavior must be acknowledged.
R2	The reasoning must correctly describe the execution order of conditionals, loops, and function calls.
R3	The reasoning must not describe conditionals that are unreachable or nonexistent in the code.
R4	The reasoning must not misattribute algorithmic behavior (e.g., sorting, filtering) where it does not occur.
Dimension: Conclusion	
R1	The reasoning must make a single, unambiguous verdict and bind it to the analyzed defect site(s).
R2	The reasoning must not rely on unstated assumptions about specifications, libraries, or inputs.
Dimension: Consistency	
R1	The reasoning must not contain statements that contradict earlier claims.
R2	Terminology and abstraction levels must remain consistent throughout the explanation.
R3	The same code element must not be assigned conflicting roles.
R4	No conclusions may be drawn without sufficient grounding in the code.

B Additional Details of Limitation Analysis

B.1 Implementation Details of the Limitation Analysis Pipeline

B.1.1 Diagnostic Annotation

We use GPT-5.2 to generate diagnostic annotations for mismatch cases. The model is provided with the original task, the LLM-generated reasoning chain and final output, and the evaluator's detailed assessment outputs for that reasoning. The diagnostic annotation stage uses two prompt variants corresponding to the two mismatch types defined in § 4.1: *missed errors* and *false alarms*.

Prompt for Missed Errors This prompt is used for mismatch cases in which the evaluator assigns a high score to reasoning that is actually incorrect. The full prompt is shown in Table A3.

Prompt for False Alarms This prompt is used for mismatch cases in which the evaluator assigns a low score to reasoning that is actually correct. The full prompt is shown in Table A4.

B.1.2 Open-Coded Limitation Label Extraction

In the second stage of the pipeline, we use GPT-5.2 to extract open-coded limitation labels from the diagnostic annotations generated in the previous stage. These labels are intended to capture the specific failure modes reflected in each mismatch case before taxonomy consolidation into higher-level categories. The full prompt is shown in Table A5.

B.1.3 Taxonomy Consolidation

In the third stage of the pipeline, we consolidate the open-coded limitation labels into higher-level limitation categories. This stage follows a two-step process: embedding-based pre-clustering using `text-embedding-3-large`, followed by taxonomy consolidation using GPT-5.2. The pre-clustering step groups semantically similar labels before GPT-based consolidation into the final taxonomy. The full prompt for taxonomy consolidation is shown in Table A6.

PROMPT TEMPLATE: DIAGNOSTIC ANNOTATION FOR MISSED ERRORS

```
You are an expert analyzing why the reasoning evaluation method {method_name} produces
scores/conclusions that significantly differ from human evaluation when assessing LLM-generated
reasoning.

## Background
This is a case where {method_name} gave a high score, but the reasoning is actually incorrect
(human label = 0). Your task is NOT to judge whether the LLM's reasoning is correct (this is
already determined by human annotation). Instead, analyze what content the evaluation method
{method_name} failed to assess in this LLM-generated reasoning, causing it to give a high
score while human annotators marked it as incorrect.

## {method_name} Evaluation Method Details
{method_description}

## Case Content
{case_content}

## Your Analysis Task
The "Execution Log" section above shows the actual intermediate outputs from the {method_name}
pipeline for this case. Use this evidence to pinpoint exactly where and why the evaluation
failed. Specifically:
1. Identify which specific component/step in the pipeline produced the erroneous signal (e.g.,
which SRL parse was wrong, which NLI judgment was incorrect, which dimension scored too high,
or how the self-generated reference was flawed).
2. Explain the root cause: why did that component fail on this particular input?
3. What capability is missing from {method_name} that would be needed to correctly evaluate
this case?
```

Table A3: Prompt template used to generate diagnostic annotations for missed-error cases, where the evaluator assigns a high score to reasoning that is actually incorrect. Here, {case_content} includes the original task, the LLM-generated reasoning chain and final output, and, the execution log containing the evaluator's intermediate assessment outputs. Text in blue braces denotes dynamic variables.

C Additional Details of VERA

C.1 Prompt for Verified Reasoning Evaluation

Table A7 presents the prompt template used in the first stage of VERA. Given a coding-task description, an LLM-generated reasoning chain, and the corresponding output, this prompt instructs the evaluator to assess the overall correctness of the reasoning process. When technical claims require external verification, the evaluator is allowed to consult web search for evidence grounding. The returned score is an integer from 1 to 10, which is later normalized to $[0, 1]$ as described in § 5.

C.2 Prompt for Ambiguity-Aware Score Correction

Table A8 presents the prompt template used in the second stage of VERA. Given the task description, the reasoning chain, and the generated output, the evaluator estimates two quantities: the ambiguity level of the task and the quality of the model's uncertainty handling. These two scores are then used to compute the ambiguity-aware correction term described in § 5.

D Additional Details of Experiment

D.1 Baseline Methods

RECEVAL RECEVAL (Reasoning Chain Evaluation) (Prasad et al., 2023) is a reference-free framework for evaluating multi-step reasoning chains, originally developed for tasks such as deductive reasoning, mathematical word problems, and discrete reading comprehension. It conceptualizes a reasoning chain as an informal proof and decomposes each step into fine-grained Reasoning Content Units (RCUs) using semantic role labeling. The method evaluates reasoning along two primary dimensions, correctness and informativeness: correctness is measured through both intra-step and inter-step logical consistency using natural language inference, while informativeness is quan-

PROMPT TEMPLATE: DIAGNOSTIC ANNOTATION FOR FALSE ALARMS

```
You are an expert analyzing why the reasoning evaluation method {method_name} produces
scores/conclusions that significantly differ from human evaluation when assessing LLM-generated
reasoning.

## Background
This is a case where {method_name} gave a low score, but the reasoning is actually correct
(human label = 1). Your task is NOT to judge whether the LLM's reasoning is correct (this is
already determined by human annotation). Instead, analyze what content the evaluation method
{method_name} may have incorrectly penalized in this LLM-generated reasoning, causing it to
give a low score while human annotators marked it as correct.

## {method_name} Evaluation Method Details
{method_description}

## Case Content
{case_content}

## Your Analysis Task
The "Execution Log" section above shows the actual intermediate outputs from the {method_name}
pipeline for this case. Use this evidence to pinpoint exactly where and why the evaluation
failed. Specifically:
1. Identify which specific component/step in the pipeline produced the erroneous signal (e.g.,
which step was the bottleneck, which SRL parse was malformed, which NLI score was unreasonably
low, or which dimension was incorrectly penalized).
2. Explain the root cause: why did that component fail on this particular input?
3. What are the characteristics of this correct reasoning that confused the {method_name}
evaluator?
```

Table A4: Prompt template used to generate diagnostic annotations for false-alarm cases, where the evaluator assigns a low score to reasoning that is actually correct. Here, **{case_content}** includes the original task, the LLM-generated reasoning chain and final output, and, when available, the execution log containing the evaluator's intermediate assessment outputs. Text in blue braces denotes dynamic variables.

tified by pointwise V-information to capture each step's information gain toward the final answer. In its original form, RECEVAL outputs continuous chain-level scores for these dimensions by applying minimum-pooling aggregation over step-level evaluations, where correctness is a probability in $(0, 1)$ and informativeness is an unbounded real-valued score.

SOCREVAL SOCREVAL (Socratic Method-Inspired Reasoning Evaluation) (He et al., 2024) is a reference-free framework for assessing machine-generated reasoning chains without model fine-tuning or human-written references. It uses large language models prompted with Socratic principles, including Definition, Maieutics, and Dialectic, to guide qualitative evaluation of reasoning. The evaluator first generates its own response to the input prompt, then analyzes the candidate reasoning chain, and finally assigns an overall reasoning-quality judgment. The original method outputs a discrete score from 1 to 5 and was developed and validated on a range of general reasoning tasks, including arithmetic reasoning, deductive and commonsense inference, and discrete reasoning over paragraphs.

CaSE CaSE (Causal Stepwise Evaluation) (Do et al., 2025) is a step-level framework for reasoning evaluation, originally developed for mathematical problem-solving tasks. To reduce hindsight bias and better reflect autoregressive reasoning generation, it evaluates each intermediate step using only the original question and the preceding context available before that step. The method assesses reasoning mainly along two dimensions, relevance and coherence, where relevance measures whether a step is grounded in the problem and coherence measures whether it logically follows from prior steps. In its original form, CaSE outputs explicit binary judgments (0 or 1) for each reasoning step on these dimensions, providing localized fine-grained

```
PROMPT TEMPLATE: OPEN-CODED LIMITATION LABEL EXTRACTION

You are analyzing a case in which {method_name}, an evaluation method for assessing
LLM-generated reasoning on coding tasks, produced an assessment that is misaligned with the
actual correctness of the reasoning.

## Mismatch Type: {mismatch_type}
{mismatch_description}

## Diagnostic Annotation
The following diagnostic annotation explains why {method_name} failed on this case:
{annotation}

## Task
Based on the diagnostic annotation above, extract 2-4 limitation labels that describe why
{method_name} failed in this case.

Key question: What limitation caused {method_name} to make this misjudgment?

## Requirements
- Derive the labels directly from the diagnostic annotation
- Each label should contain 2-5 words, written in lowercase with underscores
- Focus on limitations in evaluating LLM-generated reasoning for coding tasks
- Do not use overly generic labels that could apply to any evaluation error

Return JSON:
{{"limitations": ["label1", "label2", ...]}}
```

Table A5: Unified prompt template used to extract open-coded limitation labels from diagnostic annotations in the second stage of the pipeline. Here, **{method_name}** denotes the evaluation method being analyzed; **{mismatch_type}** denotes the mismatch type of the case, whose value is either *missed errors* or *false alarms*; **{mismatch_description}** provides the corresponding textual description of that mismatch type; and **{annotation}** denotes the diagnostic annotation generated in the previous stage for the case. The prompt returns 2–4 limitation labels in JSON format. Text in blue braces denotes dynamic variables.

evaluations rather than a single holistic score.

MAD MAD (Multi-Agent Debate) is adapted by Zhang et al. (Zhang et al., 2025) for detecting low-quality reasoning through structured multi-agent debate. In this setup, a verifier raises potential logical flaws in a candidate reasoning chain, a defender responds with counterarguments and supporting evidence, and an arbiter evaluates the rationality of the debate outcome. The original method produces a final binary decision after a fixed number of debate rounds.

LLM-as-Judge LLM-as-Judge (Zheng et al., 2023) is a general evaluation paradigm that uses strong large language models to assess the quality of candidate responses across diverse tasks, including coding and mathematical reasoning. It supports both pairwise comparison and direct scoring; in the latter setting, the judge assigns a score, typically with a textual explanation. In our experiments, we use it as a vanilla LLM-based evaluator for overall reasoning quality assessment.

AutoRace AutoRace (Hao et al., 2024) is a fully automated framework for evaluating step-by-step reasoning chains, originally developed for mathematical, commonsense, and logical reasoning tasks. It operates by first inducing task-specific evaluation criteria from collected incorrect reasoning chains, using a LLM to identify and summarize common error patterns. The resulting criteria are then used to guide detailed evaluation of a candidate reasoning chain. Since these criteria are generated dynamically for each task, the evaluation dimensions are task-adaptive rather than fixed, and the original method outputs a binary judgment for the overall reasoning process.

D.2 Evaluation Metrics

Somers' D and Spearman's ρ, both defined in $[-1, 1]$, measure the ordinal and monotonic asso-

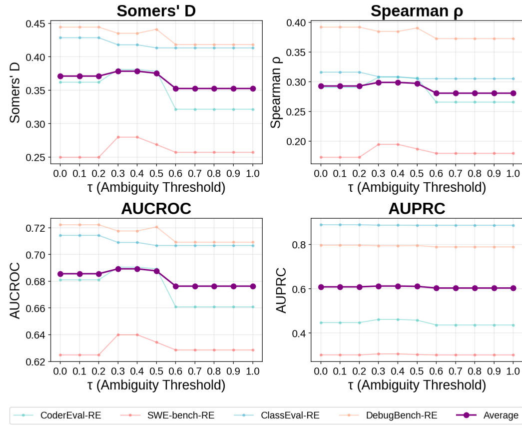

Figure A1: Threshold sensitivity analysis for the ambiguity threshold τ in VERA.

ciation between predicted scores and ground-truth correctness labels, respectively. AUCROC measures the probability that a correct reasoning trace receives a higher score than an incorrect one, while AUPRC is particularly informative under class imbalance.

D.3 Implementation Details

We follow the original configurations of all baselines whenever possible. RECEVAL retains its original correctness scores, with informativeness normalized to $[0, 1]$ before averaging the three reported dimensions into a unified score. SOCREVAL and CaSE use GPT-4 by default and GPT-4.1 on SWEbench-RE due to context length limits.

VERA, MAD*, LLM-as-Judge, and AutoRace* are all instantiated with GPT-4o-mini for fair comparison. Each MAD debate runs for three rounds. For AutoRace*, we first collect incorrect reasoning examples using Qwen-2.5 with manual verification, and then use GPT-4o-mini for both criteria induction and final evaluation.

VERA uses $\tau = 0.4$, selected based on the threshold sensitivity study in Appx. D.4. All baseline outputs are normalized to $[0, 1]$ for comparability.

D.4 Threshold Sensitivity of the Ambiguity Correction

We analyze the sensitivity of VERA to the ambiguity threshold τ used in the ambiguity-aware correction module. To do so, we vary τ from 0.0 to 1.0 and evaluate the resulting performance on all four datasets using the same metrics as in the main experiments.

Fig. A1 summarizes the results. Overall, VERA achieves the strongest average performance around $\tau = 0.4$, which we therefore use in all experiments. Smaller values make ambiguity-aware correction overly aggressive, whereas larger values make it too conservative. The effect is more pronounced on the generation-oriented datasets, while the summarization and bug detection datasets are relatively less sensitive to the threshold.

PROMPT TEMPLATE: TAXONOMY CONSOLIDATION

```
You are given {num_pcs} label clusters (called "preclusters") that describe mismatch patterns
of automated methods for evaluating Chain-of-Thought reasoning in coding tasks.

These preclusters were extracted from 1,069 mismatch cases across three coding tasks
(code generation, code summarization, bug detection) and three evaluation methods (CaSE,
ReCEval, SocREval). Each precluster name describes a specific way an evaluation method
mismatches—either by missing an error in reasoning or by falsely flagging correct reasoning.

## All {num_pcs} Preclusters (sorted by frequency):
{formatted_preclusters}

## Your Task
Group these {num_pcs} preclusters into categories that are strictly MECE (Mutually Exclusive
and Collectively Exhaustive).

### MECE Requirements
Mutually Exclusive means:
- For ANY given precluster, there must be exactly ONE category it belongs to—no ambiguity.
- Two categories X and Y are mutually exclusive if and only if: there is NO conceivable
mismatch instance that could be equally well described by both X and Y.
- Conceptual test: If someone describes a mismatch and you cannot decide whether it is X or
Y, then X and Y are NOT mutually exclusive and must be merged or re-divided along a clearer
fault line.

Collectively Exhaustive means:
- Every single one of the {num_pcs} preclusters must fit into exactly one category.
- No residual "other" or "miscellaneous" category.

### What NOT to do
- Do NOT create categories based on evaluation method names (CaSE, ReCEval, SocREval)—categories
should describe the nature of the mismatch, not which tool has it.
- Do NOT create categories that are subsets of other categories.
- Do NOT create categories that have causal relationships (e.g., "shallow analysis" causing
"logic errors")—if A commonly causes B, they are not independent dimensions and need rethinking.

### Instructions
1. First, read through all {num_pcs} preclusters and identify the natural fault lines.
2. Propose categories. For each category, provide:
- A clear name (2-5 words, lowercase_with_underscores)
- A precise definition (1-2 sentences)
- A boundary rule: how to decide if a precluster belongs here vs. in another category
3. After proposing all categories, run a pairwise MECE check: for each pair (X, Y), state in
one sentence why a mismatch described by X cannot also be described by Y. If you find overlap,
revise.
4. Finally, assign every precluster to exactly one category.

### Output Format
Return a JSON object:
{ "reasoning": "Your analysis of the natural fault lines you identified",
"categories": [ {
"name": "category_name", "definition": "Precise definition of this category",
"boundary_rule": "When a precluster is X vs Y, it belongs here if...",
"assigned_preclusters": ["precluster1", "precluster2", ...] } ],
"pairwise_checks": [ { "pair": ["category_a", "category_b"],
"why_exclusive": "One sentence explaining mutual exclusivity" } ] }
```

Table A6: Prompt template used for GPT-based taxonomy consolidation in the third stage of the pipeline. Here, **{num_pcs}** denotes the number of preclusters produced after embedding-based pre-clustering, and **{formatted_preclusters}** denotes the formatted list of those preclusters together with their frequency, size, and task-distribution information. The prompt asks the model to consolidate these preclusters into a final taxonomy and return category names, definitions, boundary rules, assigned preclusters, and consistency checks in JSON format. Text in blue braces denotes dynamic variables.

Prompt for Verified Reasoning Evaluation

```
You are a strict evaluator of AI reasoning chains in coding tasks, with access to web search.
Most AI-generated reasoning contains errors — your job is to find them.

## Task / Question:
{question}

## AI's Reasoning Chain:
{chain}

## AI's Generated Output:
{output}

Use web search to verify specific technical claims when needed (API usage, library
functions, language features, algorithm correctness).

Evaluate the CORRECTNESS of the reasoning chain and whether it leads to a correct
solution:
- Does the reasoning correctly identify the key technical aspects of the task?
- Are claims about algorithms, data structures, edge cases, and syntax correct?
- Does the reasoning lead to an output that correctly solves the problem?
- Are there factual errors about language features, APIs, or libraries in the reasoning?

Rate the reasoning correctness from 1 to 10 (most samples should fall in 3-7 range;
reserve 9-10 only for truly flawless reasoning):
10 = Flawless reasoning; all claims correct, leads to correct solution
8-9 = Strong; minor issues that do not affect the solution's correctness
6-7 = Acceptable; one moderate technical error or oversight in reasoning
4-5 = Problematic; significant error that would affect solution correctness
2-3 = Poor; multiple reasoning errors that would produce wrong results
1 = Fundamentally wrong; core reasoning approach is incorrect

Respond with ONLY a JSON object:
{"score": <1-10>, "reason": "brief justification"}
```

Table A7: Prompt template used in the verified reasoning evaluation stage of VERA. The placeholders {question}, {chain}, and {output} are instantiated with the task description, the candidate reasoning chain, and the corresponding generated output, respectively.

Prompt for Ambiguity-Aware Score Correction

```
You are evaluating how well an AI's reasoning chain handles uncertainty in a coding task.

## Task / Question:
{question}

## AI's Reasoning Chain:
{chain}

## AI's Generated Output:
{output}

Assess two things:

### 1. Task Ambiguity Level (0.0 - 1.0)
How ambiguous or underspecified is this task for a software developer?

Use this calibrated scale — note that scores above 0.5 should be common for real-world tasks:
- 0.9–1.0 = Fundamentally underspecified: no single correct answer; any experienced developer
would need to ask clarifying questions before starting. E.g., "improve this function," "fix
the bug," open-ended refactoring, or missing success criteria.
- 0.6–0.8 = Significantly ambiguous: the task has a plausible main interpretation, but key
design decisions (error handling, edge cases, performance/readability trade-offs, API style)
are left unstated and reasonable developers would make different choices.
- 0.3–0.5 = Mildly ambiguous: mostly clear, but one or two specific behaviors are underspecified
(e.g., what to return on empty input, whether to mutate or copy, which exception type to
raise).
- 0.1–0.2 = Essentially clear: a well-specified algorithmic task; at most trivial naming
choices remain.
- 0.0 = Fully deterministic: one unambiguous correct answer, no design decisions required.

### 2. Uncertainty Handling Quality (0.0 - 1.0)
Only matters when ambiguity_level > 0.2. How appropriately does the AI's reasoning respond to
the actual ambiguity?

Strict scale — most reasoning chains will score low because they ignore ambiguity:
- 0.8–1.0 = Excellent: explicitly names the ambiguity, states which interpretation was chosen
and why, preserves conditional conclusions ("if X, then A; otherwise B"), or reasonably
abstains on genuinely unknowable aspects.
- 0.5–0.7 = Partial: acknowledges uncertainty somewhere in the reasoning but does not fully
address it (e.g., mentions "assuming X" once without justification).
- 0.2–0.4 = Poor: proceeds confidently with one interpretation, makes large implicit
assumptions, and gives no acknowledgment of alternatives.
- 0.0–0.1 = Actively misleading: presents a highly contested design choice as the only correct
approach, or expresses false certainty about unknowable behavior.

Default: if ambiguity_level < 0.2, set handling_quality = 0.5 (neutral — the task is
clear, so uncertainty handling is not relevant).

Respond with ONLY a JSON object:
{"ambiguity_level": <0.0-1.0>, "handling_quality": <0.0-1.0>, "handling_issues": "brief
explanation"}
```

Table A8: Prompt template used in the ambiguity-aware score correction stage of VERA. The placeholders {question}, {chain}, and {output} are instantiated with the task description, the candidate reasoning chain, and the corresponding generated output, respectively.