TRUSTTRACE: CODE GENERATION AND REVISION WITH HIERARCHICAL PROGRAM KNOWLEDGE

Anonymous authors
Paper under double-blind review

ABSTRACT

Code generation requires algorithmic knowledge about a task and the computations that implement a solution. A central challenge is to preserve the relation between a solution's global strategy, the responsibilities of its components, and their executable behavior. We present TRUSTTRACE, a framework for code generation and revision based on hierarchical program knowledge. We organize a solution into task strategies, module descriptions, and code implementations. Module descriptions specify what each component computes, while interfaces and dependencies determine how components work together. We use this structure to learn from complete solutions and to examine the behavior of individual modules when revising a program. During learning, we assign each structured response a quality weight that determines its contribution to training. During revision, we compare observed module inputs and outputs with the responsibilities stated in the descriptions. This connects the organization of algorithmic knowledge with the execution of the resulting program. Experiments show that quality weighting raises Test whole-problem accuracy over ordinary modular SFT from 62.05% to 63.05% with supplied plans and from 17.93% to 20.92% without plans, while module traces with corresponding guidance raise revision success over score-and-error feedback from 66.50% to 67.51% with five candidates.

1 INTRODUCTION

Code generation connects a problem specification to an executable computation. Code language models learn patterns from natural language and programs (Chen et al., 2021; Rozière et al., 2024; Guo et al., 2024), while a working solution must also coordinate interfaces, dependencies, and state transfers. A locally plausible computation can fail when composed with the rest of the program.

Reasoning-guided generation and planning expose intermediate solution steps (Wei et al., 2022; Jiang et al., 2024; Li et al., 2025a). Yet a correct description of an interval algorithm may accompany code that mishandles boundaries or transfers inconsistent state between helpers. Pairing each module's responsibility with its implementation makes this correspondence inspectable. Execution then supplies observations of how the modules behave together.

We represent a solution as a sequence of description–code pairs, with each description specifying a module's responsibility and its relations to other modules. We use these pairs to train the generator, score complete responses, and associate runtime observations with the relevant code. We study two uses of this representation: weighting SFT responses by learned quality scores and revising programs with module execution feedback.

Prior work explores modular instruction tuning in MoTCoder (Li et al., 2025b), execution-informed step scoring in CodePRM (Li et al., 2025c), and runtime debugging in LDB (Zhong et al., 2024). Our scorer aggregates structural and module-correspondence features into a response score, computed once before generator training and used as a fixed SFT weight. At inference, the scorer selects a candidate, and module inputs, outputs, and execution status supply evidence for revision. We thus use one response scorer to set training weights and to select programs for revision.

We evaluate TRUSTTRACE on 10,034 modular programming problems with 1,148,459 test-case pairs. In Plan, we provide an algorithmic rationale with the problem; in Direct, we provide the problem alone. With the same targets and loss normalization, response weighting raises Test StrictAcc

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

over ordinary modular SFT from 62.05% to 63.05% for Plan and from 17.93% to 20.92% for Direct. We also compare revision feedback on 397 sampled Plan problems. With five candidates, module traces and corresponding guidance achieve 67.51% success, compared with 66.50% for score-and-error feedback.

Our contributions are:

- A description–code representation that pairs module responsibilities and interfaces with implementations and reuses these pairs for supervision, response scoring, and execution feedback. Supervision-format and cross-backbone comparisons evaluate its generation behavior in both input conditions.
- Response-weighted modular SFT with fixed quality scores and shared response-level loss normalization. Compared with ordinary modular SFT, it raises Test StrictAcc from 62.05% to 63.05% for Plan and from 17.93% to 20.92% for Direct.
- A revision scheme combining observed module inputs, outputs, and status with corresponding guidance. We obtain higher success than score-and-error feedback at each evaluated candidate prefix, reaching 67.51% versus 66.50% with five candidates.

2 Related Work

Supervision representation. Code pretraining (Wang et al., 2021; Rozière et al., 2024; Guo et al., 2024), chain-of-thought (Wei et al., 2022), and self-planning (Jiang et al., 2024) provide foundations for reasoning-guided generation. Structured CoT organizes sequence, branch, and loop structures (Li et al., 2025a), and supervision format affects generation (Liu et al., 2025a). MoTCoder (Li et al., 2025b) constructs function headers and purpose descriptions, implements and composes modules, and uses the resulting targets for instruction tuning and self-reflection. We use description–code pairs both as supervision targets and to associate response features and execution records with individual modules. To evaluate response weighting, we keep the targets fixed and change their contribution to the SFT loss.

Scoring unit and training use. Synthetic instruction data (Luo et al., 2024; Wei et al., 2024) and context distillation (Bai et al., 2022; Snell et al., 2022) inform how we construct module descriptions and implementations. CodePRM (Li et al., 2025c) labels thought steps with the pass rates of derived code and learns step scores to guide reasoning refinement during code search. We instead score complete responses using structural and module-correspondence features. We use each response score to set a fixed SFT weight during training and to rank candidate programs at inference.

Execution evidence and revision. Execution outcomes guide reward-based training (Le et al., 2022; Liu et al., 2023; Gehring et al., 2025), debugging (Chen et al., 2023), reflection (Shinn et al., 2023), and tool use (Yao et al., 2023; Wang et al., 2024). Reflexion uses reflective text from task feedback in subsequent trials, while Self-Refine iteratively improves outputs using model-generated feedback (Madaan et al., 2023). LDB (Zhong et al., 2024) inspects basic blocks and intermediate runtime variable values to guide debugging. We record inputs, outputs, and execution status at module boundaries and compare them with each module's description and implementation. We evaluate this feedback together with its revision guidance against score-only and score-and-error feedback.

3 Hierarchical Program Knowledge in TrustTrace

We organize each solution into a task-level strategy, module responsibilities, and code implementations (Figure 1). The generator produces module descriptions and code blocks in sequence. We score the resulting response as a whole and record function inputs, outputs, and execution status when the assembled program runs.

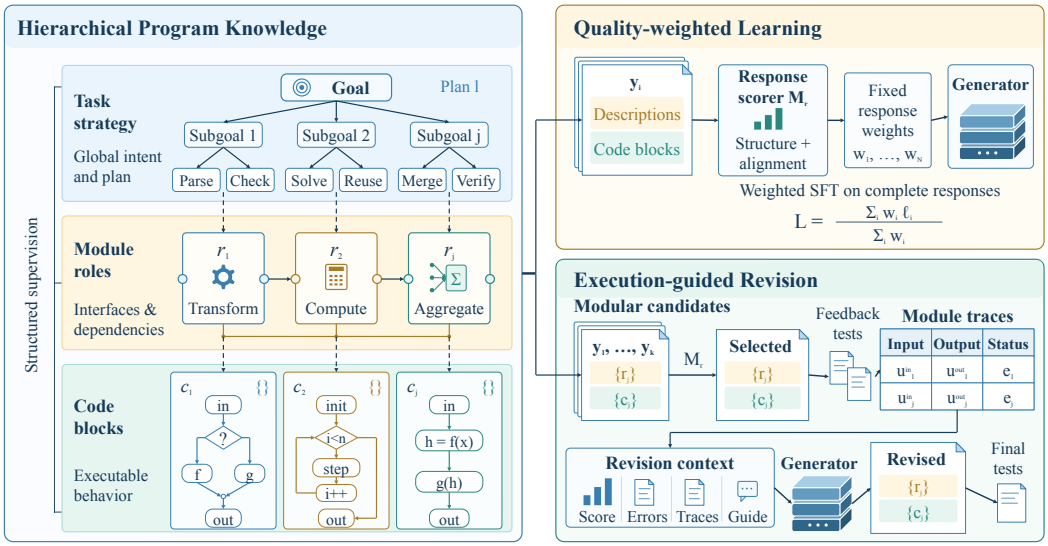


Figure 1: Overview of TRUSTTRACE. Structured supervision aligns task strategies, module descriptions, and code. We use a response-quality scorer to set fixed SFT weights and rank generated candidates. Module-boundary inputs, outputs, and status enter the revision context with scores, errors, and guidance.

3.1 KNOWLEDGE AT THREE LEVELS

Task-level knowledge. A problem x specifies the desired input-output behavior, and a rationale l describes an algorithmic strategy. We use the rationale to identify the main algorithmic steps and their required order.

Module-level knowledge. The strategy is organized into responsibilities, such as input transformation, a reusable computation, or result aggregation. Each module description r_j states its purpose, required inputs, outputs, and dependencies. These descriptions connect global intent to computations that can be inspected separately.

Implementation-level knowledge. A code block c_j realizes each module, and the assembled program determines how data move between them. During execution, observed module inputs, outputs, and failures provide concrete evidence about these computations. A response is serialized as $y = \text{serialize}((r_1, c_1), \ldots, (r_J, c_J))$. The hierarchy is encoded through responsibilities and dependencies within this sequence; scoring and training weights operate on the complete response.

For example, Appendix C counts set bits over a signed integer interval. The global strategy separates negative and nonnegative inputs. The interval helper `sum_bits(A,B)` calls a cumulative-counting function at the two boundaries and subtracts the results. Its description specifies the interval-sum operation, and its arguments and return value identify the quantities exchanged with the caller. A wrong boundary argument can therefore be examined at this interface, separately from the cumulative-counting implementation.

Response and module indexing. We keep module identifiers with their descriptions and code. The generator is trained on the full serialized response; all supervised tokens in that response share one weight under Equation 3. During revision, we use the module identifier to attach each execution record to the corresponding function. We then generate and evaluate a complete revised program.

3.2 Constructing Structured Supervision

We use the `seed_sft` and `seed_testcase` splits of rStar-Coder (Liu et al., 2025b). A seed sample is $S_i = (x_i, l_i, y_i^{\text{seed}}, \mathcal{T}_i)$, containing a problem, a long rationale, reference code, and tests. Following teacher-to-student context distillation (Bai et al., 2022; Snell et al., 2022), we use Qwen3.5-Plus to extract major steps and substeps from the existing rationale, group them into module descriptions, and generate corresponding code. Descriptions specify responsibilities and dependencies; code blocks expose signatures and return values and can be assembled with an entry function. Appendix B provides the construction and revision prompts.

We retain outputs whose assembled code passes the available tests and whose fields can be parsed into the required format. Table 1 describes the benchmark used in the reported comparisons.

Table 1: Benchmark statistics for the reported experiments. Sample counts refer to problems; test-case pairs are the associated input-output tests.

Statistic	Train	Valid	Test	Overall
Samples	8,045	985	1,004	10,034
Test-case pairs	917,582	117,852	113,025	1,148,459
Avg. input tokens	1,364.33	1,373.30	1,360.00	1,364.78
Avg. output tokens	1,688.91	1,688.64	1,680.30	1,688.02
Avg. code tokens	677.91	682.83	676.89	678.29
Avg. modules	4.34	4.34	4.38	4.35

3.3 Learning the Representation

Two input conditions share the modular output format. In $\mathcal{D}_{\text{direct}}$, $a_i = x_i$ contains only the problem. In $\mathcal{D}_{\text{plan}}$, $a_i = (x_i, l_i)$ also contains the seed's algorithmic rationale, including at evaluation time. Plan therefore measures implementation conditioned on supplied solution information; Direct measures generation from the problem alone. Comparisons between methods are made within each condition. Ordinary and weighted SFT use the same response-level normalization: token losses are first averaged within each response, ignoring instruction and padding labels. For a loss-reduction group of N responses, with L_i supervised tokens in response i, ordinary SFT uses their mean:

$$\mathcal{L}_{\text{SFT}}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\frac{1}{L_i}\sum_{t=1}^{L_i}\log p_\theta(y_{i,t} \mid a_i, y_{i,<t}). \tag{1}$$

We call this unweighted modular baseline ModuCoder. We supervise both the module descriptions and their code as tokens in the same response.

3.4 Assessing Knowledge for Weighted Learning

Scorer training uses positive modular responses from $\mathcal{D}_{\text{direct}}$ and negative responses collected during synthesis evaluation, including failed programs and incomplete outputs. Before the independent test holdout, the source data have a positive-to-negative ratio of approximately 8:1. The scoring test set is held out by problem, so responses to a problem cannot cross the training–test boundary. The remaining scorer input data are stratified by class into 95% training and 5% validation. In $\mathcal{D}_{\text{score}} = \{(a_i, y_i, u_i)\}_{i=1}^{N_r}$, $u_i \in \{0, 1\}$ is the binary class label: 1 for a response from the positive set and 0 for a response from the negative set. We encode structural and description–code statistics as input features and use the binary label as the classification target. Scorer training excludes generator validation and test problems (Appendix A.13). We select the classification threshold η by scoring-validation Macro-F1 and weight configurations on generator Valid. We extract a feature vector $z_i = \phi(y_i)$ from each response:

$$z_i = \left[z_i^{\text{stat}}, \bar{e}_i^{\text{nlp}}, \bar{e}_i^{\text{code}}, |\bar{e}_i^{\text{nlp}} - \bar{e}_i^{\text{code}}|, \bar{e}_i^{\text{nlp}} \odot \bar{e}_i^{\text{code}}\right]. \tag{2}$$

Here $\bar{e}_i^{\text{nlp}}$ and $\bar{e}_i^{\text{code}}$ average module token-distribution vectors; z_i^{stat} contains structural and similarity statistics. Appendix A.6 specifies the hashing and aggregation. We train the MLP members

of an EasyEnsemble (Liu et al., 2009) with binary cross entropy, $-u_i \log p_i - (1-u_i)\log(1-p_i)$, where p_i is a member's predicted positive-class probability. We aggregate their probabilities into $s_i = M_r(\phi(y_i))$. Thus u_i is the supervision target, z_i is the model input, and s_i is the response score used for weighting and selection.

Before generator training, we score each target response once and keep its weight fixed throughout SFT. Gradients update only the generator. We use $w_{\min} = 0.3$ and $w_{\max} = 1.0$, with upper threshold $\tau = 0.5$. The same response features and ensemble are used later to score generated candidates.

Weighted SFT first averages token losses within each response and then takes the normalized weighted mean of response losses:

$$\mathcal{L}_{\text{W-SFT}}(\theta) = \frac{\sum_i w_i \ell_i(\theta)}{\sum_i w_i}, \qquad \ell_i(\theta) = -\frac{1}{L_i}\sum_{t=1}^{L_i} \log p_\theta(y_{i,t} \mid a_i, y_{i,<t}). \tag{3}$$

The weight controls each complete response's relative contribution to the training loss. Setting every $w_i = 1$ recovers ordinary SFT in Equation 1; both objectives share response-level loss normalization.

We map scores to weights using effective thresholds satisfying $\tau_{\text{low}} < \tau_{\text{high}}$:

$$w_i = \begin{cases} w_{\min}, & s_i \le \tau_{\text{low}}, \\ w_{\min} + (w_{\max} - w_{\min}) \left(\dfrac{s_i - \tau_{\text{low}}}{\tau_{\text{high}} - \tau_{\text{low}}} \right)^{\gamma}, & \tau_{\text{low}} < s_i < \tau_{\text{high}}, \\ w_{\max}, & s_i \ge \tau_{\text{high}}. \end{cases} \tag{4}$$

For the main configuration, $\tau_{\text{high}} = \tau = 0.5$, $\tau_{\text{low}} = \tau - \epsilon$, $\epsilon = 10^{-6}$, and $\gamma = 2$. A score at $\tau - \epsilon$ receives 0.3; a score at τ receives 1.0; scores strictly between them use the interpolation branch. The implementation enforces this minimum threshold separation before computing weights. Appendix A.8 gives the threshold comparisons and an algebraic decomposition for two-level weights.

3.5 Using Execution Knowledge for Revision

For each problem x, we use $\mathcal{T}_x^{\text{fb}}$ to construct revision feedback and $\mathcal{T}_x^{\text{eval}}$ to score the final program, with $\mathcal{T}_x^{\text{fb}} \cap \mathcal{T}_x^{\text{eval}} = \varnothing$. For test $t \in \mathcal{T}_x^{\text{fb}}$, let $\mathcal{J}_t$ denote the observed modules. We record their input summaries $u_{j,t}^{\text{in}}$, output summaries $u_{j,t}^{\text{out}}$, and execution status $e_{j,t}$:

$$\text{Trace}_t(y) = \{(j, u_{j,t}^{\text{in}}, u_{j,t}^{\text{out}}, e_{j,t}) : j \in \mathcal{J}_t\}. \tag{5}$$

For a test set, $\text{Trace}(y, \mathcal{T}) = \{(t, \text{Trace}_t(y)) : t \in \mathcal{T}\}$. The wrapper captures outermost calls, storing arguments, returns, and errors while suppressing nested recursion. The module I/O extraction step selects the first captured call per function per program test. For a candidate prefix $\mathcal{C}_k$, the scorer selects $\hat{y} \in \arg\max_{y \in \mathcal{C}_k} M_r(\phi(y))$. Reranking returns it directly. The contextual input a is supplied to the generator and to feedback construction. The scorer operates on $\phi(y)$, computed from the candidate's module descriptions and code using the same tokenizer and feature aggregation as during scorer training. Reflection combines this starting program with the problem, score, and execution evidence:

$$f = \Gamma\big(a, \hat{y}, M_r(\phi(\hat{y})), \text{Trace}(\hat{y}, \mathcal{T}_x^{\text{fb}})\big), \qquad \widetilde{y} \sim p_\theta(\cdot \mid a, f). \tag{6}$$

We summarize representative failing tests and the associated module states in the revision prompt. We ask the generator to inspect transformations, interfaces, and state transfers and to preserve computations consistent with the observed traces. The selected or revised program is scored on the held-out tests $\mathcal{T}_x^{\text{eval}}$.

4 Experiments

We evaluate response-weighted learning, supervision formats, and backbone models under shared benchmark partitions. After specifying the training and evaluation settings, we compare ordinary and weighted modular SFT, alternative supervision targets, and two code-generation backbones.

4.1 Evaluation Settings

The main comparisons use the same benchmark partitions for both $\mathcal{D}_{\text{plan}}$ and $\mathcal{D}_{\text{direct}}$. Hyperparameters and scoring thresholds use validation data, with Test reserved for final evaluation. We use Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) as the primary generator and train DeepSeek-Coder-6.7B-Instruct (Guo et al., 2024) with the same modular SFT settings. Training uses PyTorch and DeepSpeed ZeRO-3 on four NVIDIA A100-PCIE-40GB GPUs, with BF16, FlashAttention-2, gradient checkpointing, and AdamW. Ordinary and weighted SFT both use a per-GPU batch size of 2 and 16 gradient accumulation steps, giving an effective batch size of 128 across four GPUs, with training seed 3407. The maximum sequence length is 4096, the learning rate is 10^{-5}, and training runs for 3 epochs with warmup ratio 0.1 and cosine decay. Deterministic generation uses temperature 0.0. Main-experiment candidate generation and reflection revisions each allow at most 2048 output tokens. Sampling uses seed 3407 and `top_p`=0.95. Appendix A gives scorer settings and full baselines.

For M problems with nonempty final scoring sets $\mathcal{T}_j$, let $P(c_j, t) \in \{0, 1\}$ indicate test success. AvgAcc first computes the fraction of tests passed for each problem and then averages these fractions over problems; StrictAcc measures the fraction of problems passing every test:

$$\text{AvgAcc} = \frac{1}{M}\sum_{j=1}^{M}\frac{1}{|\mathcal{T}_j|}\sum_{t\in\mathcal{T}_j}P(c_j,t), \qquad \text{StrictAcc} = \frac{1}{M}\sum_{j=1}^{M}\prod_{t\in\mathcal{T}_j}P(c_j,t). \tag{7}$$

AvgAcc uses problem-level macro averaging: each problem receives equal weight, regardless of the number of associated tests. On the same nonempty scoring population, $0 \le \text{StrictAcc} \le \text{AvgAcc} \le 1$. Tables report percentages separately for Valid and Test, with Test StrictAcc used for primary comparisons. Appendix A.12 specifies the metric aggregation and reporting units. In reflection, $\mathcal{T}_j = \mathcal{T}_j^{\text{eval}}$ is withheld from feedback. Section 5.1 specifies the sampled-generation protocol.

4.2 Response-Weighted Learning

We compare ordinary and weighted modular SFT using the same backbone, targets, and response-level loss normalization (Table 2). This keeps the supervision format fixed while changing how much each response contributes to the training loss.

Table 2: Training-configuration results on Qwen2.5-Coder-7B-Instruct (%). Base denotes the model before task-specific SFT; ModuCoder and TrustTrace use ordinary and score-weighted modular SFT. Bold marks column maxima within each input condition.

		Valid		Test	
Setting	Model	AvgAcc	StrictAcc	AvgAcc	StrictAcc
Plan	Base	65.00	56.75	66.12	57.67
	ModuCoder	67.56	64.06	69.70	62.05
	TrustTrace	**70.01**	**66.90**	**70.29**	**63.05**
Direct	Base	37.96	17.16	**41.47**	17.33
	ModuCoder	38.82	23.96	36.59	17.93
	TrustTrace	**43.09**	**28.53**	40.65	**20.92**

Response weighting improves AvgAcc and StrictAcc over ordinary modular SFT on both Valid and Test, under both Plan and Direct inputs. Test StrictAcc rises from 62.05% to 63.05% for Plan and from 17.93% to 20.92% for Direct. On Direct Test, weighted SFT reaches 40.65% AvgAcc, above ordinary modular SFT's 36.59% but below the base model's 41.47%. Its StrictAcc is 20.92%, compared with 17.93% and 17.33%, respectively. We report the changes between successive training configurations in Appendix Table 5.

For Plan, weighted SFT also exceeds the base model on Test AvgAcc (70.29% versus 66.12%) and StrictAcc (63.05% versus 57.67%). Thus, the comparison with the base model differs between Plan and Direct, while the comparison with ordinary modular SFT favors weighting in both conditions.

4.3 Supervision Format

We examine how the supervision format affects generation under shared training problems and filtering. Figure 2 compares Direct-code SFT, CoT-code SFT, and modular SFT on Qwen2.5-Coder-7B-Instruct, using code, reasoning followed by code, and description–code pairs as targets.

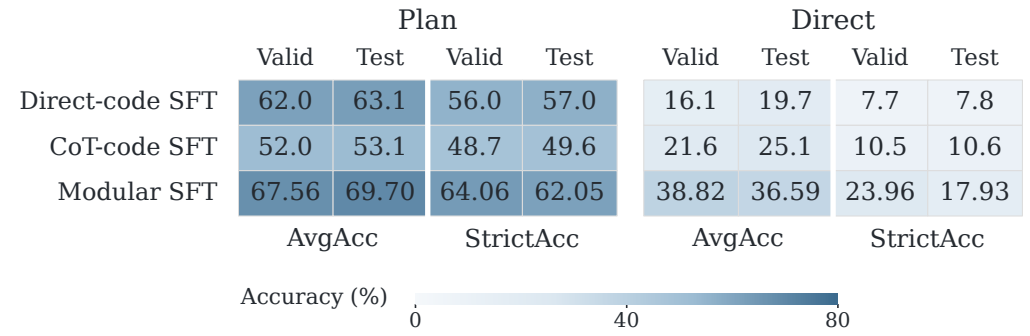


Figure 2: Supervision formats on Qwen2.5-Coder-7B-Instruct with shared training problems and filtering. Direct-code, CoT-code, and ModuCoder use code, reasoning-then-code, and modular targets, respectively.

Modular SFT has higher AvgAcc and StrictAcc in both input conditions and splits. Under the shared data selection, description–code targets yield higher generation scores than either code-only or reasoning-then-code targets.

4.4 Backbone Comparison

We examine whether the changes after modular supervision occur on both backbones. Figure 3 compares each Qwen and DeepSeek base model with its modular SFT counterpart under Plan and Direct inputs.

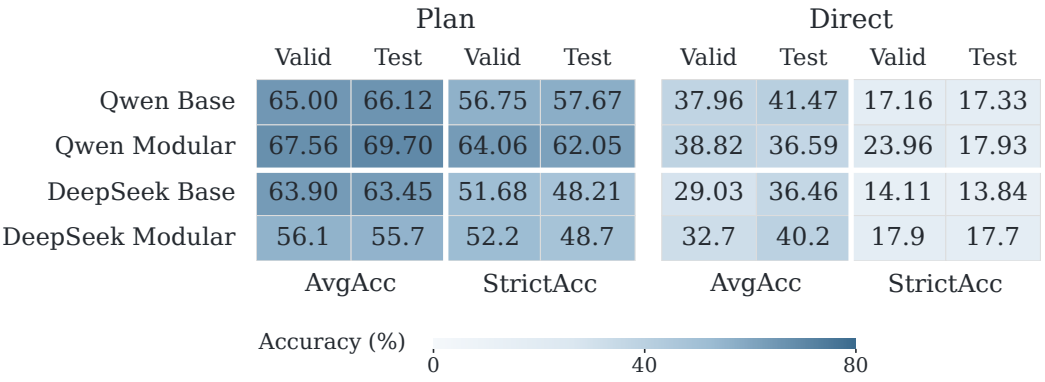


Figure 3: Qwen and DeepSeek before and after modular SFT (Section 4.1). Base denotes the corresponding Instruct checkpoint before task-specific SFT.

StrictAcc increases after modular SFT for both backbones. AvgAcc declines for DeepSeek Plan in both splits and for Qwen Direct on Test. The comparison therefore shows more fully solved problems alongside a tradeoff in partial test coverage in these conditions. On Direct Test, DeepSeek AvgAcc rises from 36.46% to 40.2%, while StrictAcc rises from 13.84% to 17.7%. Qwen instead combines higher StrictAcc with lower AvgAcc on this split. The two backbones thus share an increase in complete solutions, while their average test coverage responds differently to the same modular supervision format.

5 ANALYSIS

We analyze execution feedback, description–code correspondence, and scorer choice to examine how revision, output structure, and response weighting relate to generation performance.

5.1 MODULE EXECUTION FEEDBACK

To compare revision feedback, we evaluate TRUSTTRACE-Plan on 397 problems randomly sampled from the 1,004-problem Test split of $\mathcal{D}_{\text{plan}}$, the same problems used in the static scoring comparison. Each condition is evaluated once, with one final program per problem. Candidate generation and reflection revisions use temperature 1.0, and prefixes $k \in \{1, 3, 5\}$ are taken from the same sampled candidate pools. Every revision condition receives the selected candidate and its score. Score-only uses that information; Score+errors adds whole-program test failures; Score+errors+traces adds module inputs, outputs, execution status, and corresponding revision guidance. We construct feedback using $\mathcal{T}_x^{\text{fb}}$ and evaluate each selected or revised program using $\mathcal{T}_x^{\text{eval}}$. Table 3 compares complete feedback schemes: the last two conditions differ in both evidence and revision instructions.

Table 3: Final-program success (%) on 397 problems with shared candidate prefixes. Score, Errors, and Traces denote score-only, score+errors, and score+errors+traces feedback with corresponding guidance. Each revision uses the selected candidate and one additional generation.

	Selection		Revision		
Prefix k	First	Rerank	Score	Errors	Traces
1	50.88	50.88	53.15	52.64	**54.91**
3	50.88	51.39	53.40	52.90	**55.67**
5	50.88	60.71	64.74	66.50	**67.51**

At $k = 1, 3, 5$, adding whole-program errors to score feedback changes success from 53.15%, 53.40%, and 64.74% to 52.64%, 52.90%, and 66.50%, respectively. Module traces with corresponding revision guidance yield 54.91%, 55.67%, and 67.51%, exceeding both alternatives at each prefix. Comparisons with the first candidate measure the combined effects of selection and revision.

As a complementary analysis, Table 14 compares first-candidate selection, reranking, and score-guided reflection on MODUCODER-Plan outputs from the same problems.

5.2 REPRESENTATION ANALYSIS

We examine description–code correspondence and output validity for CoT-code SFT and modular SFT on $\mathcal{D}_{\text{direct}}$. Rule-based evaluation retains complete rationales and parseable code. DeepSeek-V3.2 (DeepSeek-AI et al., 2025) judges the supplied problem, rationale, and code on five dimensions; Table 4 reports these scores. Code nonempty and AST parseability rates are measured before filtering.

Table 4: LLM-as-a-Judge consistency evaluation. Scores are on a 0–5 scale.

Method	Step	Completeness	Logic	Structure	Traceability	Overall
CoT-code SFT	3.534	**3.208**	3.173	3.454	3.480	3.370
Modular SFT	**3.646**	2.775	**3.373**	**3.753**	**3.521**	**3.413**

Modular SFT scores higher on structure, steps, logic, and traceability. Reasoning completeness falls from 3.208 to 2.775, while the overall score rises from 3.370 to 3.413. The rule score increases from 0.1991 to 0.2403, with higher identifier consistency but lower algorithm-operation consistency (Table 15). Code nonempty rates are 95.97% and 92.44%, and AST parseability rates are 91.44% and 89.67%, for CoT-code and modular SFT, respectively. Thus, higher overall correspondence scores coexist with lower completeness and output validity.

5.3 Scorer and Weighting Analysis

We examine how scorer choice relates to weighted generation. Figure 4 compares MLP and logistic-regression weights under the same modular SFT setup. Scorer accuracy, AUC, and F1 on the independent scoring test set, together with classification accuracy on generator training data, assess response-quality classification (Tables 10 and 11).

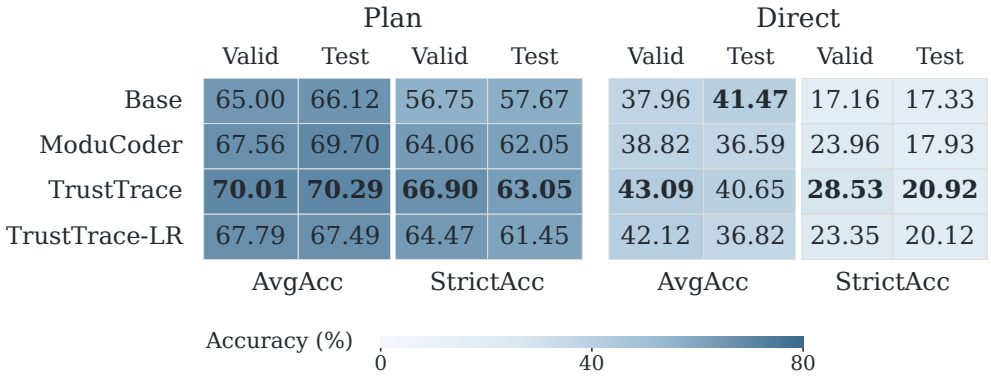


Figure 4: Generation with MLP (TrustTrace) or logistic-regression (TrustTrace-LR) weights. Base and ModuCoder denote the base model and ordinary modular SFT. Bold marks column maxima.

MLP weighting yields higher AvgAcc and StrictAcc than logistic-regression weighting on both splits and input conditions: Test StrictAcc is 63.05% versus 61.45% for Plan and 20.92% versus 20.12% for Direct.

The MLP scorer reaches 87.91% test accuracy, 85.79% AUC, 72.21% Macro-F1, and 51.33% minority-class F1. Logistic regression has higher classification accuracy on generator training data but lower test accuracy, AUC, and Macro-F1. For these two scorers, independent test metrics and weighted generation give the same ranking.

The minimum-weight comparison gives Valid/Test StrictAcc of 28.53%/20.92% at $w_{\min} = 0.3$ and 17.26%/23.41% at $w_{\min} = 0.8$. We select $w_{\min} = 0.3$ and $(\tau - \epsilon, \tau)$ using validation results; this threshold pair ties for the highest Test StrictAcc in the threshold comparison.

5.4 Response Weighting and Execution Feedback

Fixed response weights emphasize higher-scoring training responses, whereas execution feedback describes a particular candidate. For weights restricted to the lower and upper bounds, Equation 8 decomposes weighted SFT into a uniform-response component and an additional higher-weight-subset component. Both operations use the description–code format. We separately measure description–code correspondence, output completeness, parseability, whole-problem success, and partial test coverage.

6 Conclusion

We introduced TrustTrace, which represents solutions through task strategies, module descriptions, and code implementations. We use complete-response quality scores to weight modular SFT and module execution records to guide revision. Response weighting improves AvgAcc and StrictAcc over ordinary modular SFT on both splits and input conditions. Module traces with corresponding guidance also yield higher revision success than score-and-error feedback at each evaluated candidate prefix. On Direct Test, weighted SFT reaches 20.92% StrictAcc and 40.65% AvgAcc; the base model reaches 17.33% and 41.47%, respectively. Modular supervision also produces higher description–code correspondence scores and lower completeness and parseability. Across these evaluations, we observe improvements in complete-program success together with changes in partial test coverage and output structure.

AI USE STATEMENT

Generative AI assisted manuscript organization and language polishing, result interpretation, method clarification, numerical transcription, figure-generation and verification scripts, and source-code review. Generative models also supported modular data construction and automated consistency evaluation.

ETHICS STATEMENT

Generated code may contain functional errors or security vulnerabilities, and code-generation systems can be misused. Applications require independent code review and testing. Source datasets and pretrained models should be used under their respective licenses and usage terms.

REPRODUCIBILITY STATEMENT

Section 3 defines the representation and objectives; the setup and appendices specify training, evaluation, features, and prompts. Supplementary code implements response-level normalization (Equations 1 and 3) and the two-level loss decomposition (Equation 8).

REFERENCES

Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, Nicholas Joseph, Saurav Kadavath, Jackson Kernion, Tom Conerly, Sheer El-Showk, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Tristan Hume, Scott Johnston, Shauna Kravec, Liane Lovitt, Neel Nanda, Catherine Olsson, Dario Amodei, Tom Brown, Jack Clark, Sam McCandlish, Chris Olah, Ben Mann, and Jared Kaplan. Training a helpful and harmless assistant with reinforcement learning from human feedback, 2022. URL https://arxiv.org/abs/2204.05862.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023. URL https://arxiv.org/abs/2304.05128.

DeepSeek-AI et al. DeepSeek-V3.2: Pushing the frontier of open large language models, 2025. URL https://arxiv.org/abs/2512.02556.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In *Proceedings of the Forty-second International Conference on Machine Learning*, volume 267, pp. 19034–19055, 2025. URL https://proceedings.mlr.press/v267/gehring25a.html.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. URL https://arxiv.org/abs/2401.14196.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, 2025. URL https://doi.org/10.1038/s41586-025-09422-z.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2.5-Coder technical report, 2024. URL https://arxiv.org/abs/2409.12186.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30, 2024. URL https://doi.org/10.1145/3672456.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, pp. 21314–21328, 2022. URL https://doi.org/10.52202/068431-1549.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–23, 2025a. URL https://doi.org/10.1145/3690635.

Jingyao Li, Pengguang Chen, Bin Xia, Hong Xu, and Jiaya Jia. MoTCoder: Elevating large language models with modular of thought for challenging programming tasks, 2025b. URL https://arxiv.org/abs/2312.15960v5. Version 5.

Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. CodePRM: Execution feedback-enhanced process reward model for code generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 8169–8182, 2025c. URL https://aclanthology.org/2025.findings-acl.428/.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. RLTF: Reinforcement learning from unit test feedback. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=hjYmsV6nXZ.

Ren-Biao Liu, Anqi Li, Chaoding Yang, Hui Sun, and Ming Li. Revisiting chain-of-thought in code generation: Do language models need to learn reasoning before coding? In *Proceedings of the Forty-second International Conference on Machine Learning*, volume 267, pp. 38809–38826, 2025a. URL https://proceedings.mlr.press/v267/liu25ah.html.

Xu-Ying Liu, Jianxin Wu, and Zhi-Hua Zhou. Exploratory undersampling for class-imbalance learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 39(2):539–550, 2009. URL https://doi.org/10.1109/TSMCB.2008.2007853.

Yifei Liu, Li Lyna Zhang, Yi Zhu, Bingcheng Dong, Xudong Zhou, Ning Shang, Fan Yang, and Mao Yang. rstar-coder: Scaling competitive code reasoning with a large-scale verified dataset, 2025b. URL https://arxiv.org/abs/2505.21297.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=UnUwSIgK5W.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594, 2023. URL https://doi.org/10.52202/075280-2019.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950v3.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652, 2023. URL https://doi.org/10.52202/075280-0377.

Charlie Snell, Dan Klein, and Ruiqi Zhong. Learning by distilling context, 2022. URL https://arxiv.org/abs/2209.15189.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Proceedings of the Forty-first International Conference on Machine Learning*, volume 235, pp. 50208–50232, 2024. URL https://proceedings.mlr.press/v235/wang24h.html.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, 2021. URL https://aclanthology.org/2021.emnlp-main.685/.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837. Curran Associates, Inc., 2022. URL https://doi.org/10.52202/068431-1800.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct, 2024. URL https://arxiv.org/abs/2312.02120v2.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 851–870, 2024. URL https://aclanthology.org/2024.findings-acl.49/.

A Additional Experimental Results

A.1 Training-Configuration Differences

Section 4.2 compares base, modular, and weighted SFT. We report adjacent-configuration differences by input condition, split, and metric; positive values favor the later configuration.

Table 5: Adjacent training-configuration differences on the primary backbone, on the 0–1 accuracy scale. Modular and Weighted denote unweighted modular SFT and score-weighted SFT.

		Modular − base		Weighted − modular	
Setting	Split	AvgAcc	StrictAcc	AvgAcc	StrictAcc
Plan	Valid	+0.0256	+0.0731	+0.0245	+0.0284
Plan	Test	+0.0358	+0.0438	+0.0059	+0.0100
Direct	Valid	+0.0086	+0.0680	+0.0427	+0.0457
Direct	Test	−0.0488	+0.0060	+0.0406	+0.0299

Weighting increases AvgAcc and StrictAcc on both splits and input conditions. Modular SFT also improves these metrics over the base model except for Direct Test AvgAcc, which decreases.

A.2 Plan Baseline Comparison

Table 6 compares modular supervision with code and reasoning model baselines under supplied rationales, using problem-level AvgAcc and StrictAcc. The additional baselines include OlympicCoder-7B[1] and DeepSeek-R1-Distill-Qwen-7B (Guo et al., 2025).

Table 6: ModuCoder-Plan and baselines on $\mathcal{D}_{\text{plan}}$ using AvgAcc (%) and StrictAcc (%).

	Valid		Test	
Model	AvgAcc	StrictAcc	AvgAcc	StrictAcc
Qwen2.5-Coder-7B-Base	6.69	6.60	5.50	5.48
CodeLlama-7B-Instruct	31.05	24.97	29.79	23.01
CodeLlama-7B-Instruct-Python	39.16	30.46	39.25	32.07
DeepSeek-Coder-6.7B-Instruct	63.90	51.68	63.45	48.21
OlympicCoder-7B	55.49	50.36	57.29	51.00
DeepSeek-R1-Distill-Qwen-7B	60.61	55.63	55.75	50.00
DeepSeek-Coder-6.7B-Base	**67.69**	58.38	66.45	55.58
Qwen2.5-Coder-7B-Instruct	65.00	56.75	66.12	57.67
ModuCoder-Plan	67.56	**64.06**	**69.70**	**62.05**

ModuCoder-Plan leads in StrictAcc on both splits and in Test AvgAcc. DeepSeek-Coder-6.7B-Base has slightly higher Valid AvgAcc, indicating different rankings for complete success and partial coverage. On Test, ModuCoder-Plan attains 69.70% AvgAcc and 62.05% StrictAcc, compared with 66.12% and 57.67% for Qwen2.5-Coder-7B-Instruct. DeepSeek-Coder-6.7B-Base attains 66.45% and 55.58%, respectively. With the rationale supplied in each case, these comparisons evaluate how the models translate solution information into programs that satisfy the problem tests.

A.3 Direct Baseline Comparison

Table 7 compares the same baselines given the problem alone, assessing generation without supplied rationales.

[1] https://huggingface.co/open-r1/OlympicCoder-7B

Table 7: MODUCODER-Direct and baselines on $\mathcal{D}_{\text{direct}}$ using AvgAcc (%) and StrictAcc (%).

	Valid		Test	
Model	AvgAcc	StrictAcc	AvgAcc	StrictAcc
Qwen2.5-Coder-7B-Base	10.16	6.50	11.95	6.27
CodeLlama-7B-Instruct-Python	16.36	7.51	17.93	9.36
CodeLlama-7B-Instruct	17.15	8.02	22.30	8.96
DeepSeek-Coder-6.7B-Instruct	29.03	14.11	36.46	13.84
DeepSeek-R1-Distill-Qwen-7B	17.04	16.45	12.77	12.15
OlympicCoder-7B	22.05	16.85	20.68	14.34
DeepSeek-Coder-6.7B-Base	31.90	16.14	37.18	16.93
Qwen2.5-Coder-7B-Instruct	37.96	17.16	**41.47**	17.33
MODUCODER-Direct	**38.82**	**23.96**	36.59	**17.93**

MODUCODER-Direct leads in StrictAcc on both splits and in Valid AvgAcc. Qwen2.5-Coder-7B-Instruct retains higher Test AvgAcc, revealing a tradeoff between complete success and partial coverage.

A.4 WEIGHTED LEARNING WITH PLANS

Table 8 assesses weighting on $\mathcal{D}_{\text{plan}}$, holding the backbone, modular targets, and response-level normalization fixed between ordinary and weighted SFT.

Table 8: Full TRUSTTRACE-Plan results on $\mathcal{D}_{\text{plan}}$.

	Valid		Test	
Model	AvgAcc	StrictAcc	AvgAcc	StrictAcc
Qwen2.5-Coder-7B-Instruct	65.00	56.75	66.12	57.67
MODUCODER-Plan	67.56	64.06	69.70	62.05
TRUSTTRACE-Plan	**70.01**	**66.90**	**70.29**	**63.05**

Weighted SFT leads on both splits: Test StrictAcc rises from 62.05% to 63.05% and AvgAcc from 69.70% to 70.29% over modular SFT.

A.5 WEIGHTED LEARNING WITHOUT PLANS

Table 9 assesses the corresponding changes in complete success and partial test coverage without supplied rationales.

Table 9: Full TRUSTTRACE-Direct results on $\mathcal{D}_{\text{direct}}$.

	Valid		Test	
Model	AvgAcc	StrictAcc	AvgAcc	StrictAcc
Qwen2.5-Coder-7B-Instruct	37.96	17.16	**41.47**	17.33
MODUCODER-Direct	38.82	23.96	36.59	17.93
TRUSTTRACE-Direct	**43.09**	**28.53**	40.65	**20.92**

Weighting improves both metrics on both splits over modular SFT. Test StrictAcc reaches 20.92%; AvgAcc is 40.65%, compared with 41.47% for the base model.

A.6 Scorer Configuration and Analysis

We assess response-quality classification using an independent scoring test set and generator training distributions. The MLP settings are hidden dimension 1024, projection dimension 256, and dropout 0.4. Training uses AdamW with learning rate 3×10^{-4} and weight decay 10^{-3} for at most 80 epochs. ReduceLROnPlateau uses factor 0.5, patience 2, and minimum learning rate 10^{-6}; early stopping uses patience 10. The scorer uses batch size 128 and random seed 42. The independent scoring test set contains 3,795 responses: 3,095 positive and 700 negative examples. After holding out the independent scoring test set by problem, the remaining input data are split by class into 95% training and 5% validation. Each EasyEnsemble member retains all minority-class samples and subsamples the majority class into an approximately balanced subset. The classification threshold η is selected using Macro-F1 on the scoring validation split.

Feature construction. The feature builder instantiates Equation 2 with Qwen-tokenizer frequency vectors. Token IDs are reduced modulo a default hash dimension of 4096, accumulated as counts, and L2-normalized; tokenization truncates each module text to 512 tokens by default and adds no special tokens. Natural-language and code vectors are averaged separately over module metadata, including a MAIN fallback when present. Their means, absolute difference, and elementwise product are concatenated with 11 scalars: module count, MAIN presence, empty-code ratio, and mean/max/min/standard deviation of cosine similarity and CodeBLEU. The feature builder sets the four CodeBLEU statistics to zero, retaining their positions in the feature vector.

The scorer uses 9 ensemble members and a feature-subspace ratio of 0.85. Its weighted aggregation normalizes each member's best validation-objective value and averages member probabilities.

Table 10: Scoring model performance on the independent scoring test set; Plan and Direct accuracy use generator training data.

	Independent scoring test				Generator train	
Scorer	ACC	AUC	Macro-F1	Minority F1	Plan ACC	Direct ACC
MLP scorer	87.91	85.79	72.21	51.33	95.06	92.83

The MLP reaches 87.91% test accuracy and 51.33% minority-class F1; classification accuracy is higher on generator training data.

A.7 Scorer Comparison

We compare MLP and logistic-regression classification on the same independent scoring test set and Plan/Direct generator training data.

Table 11: Scorer comparison on the independent scoring test set and generator training data.

	Independent scoring test			Generator train	
Scorer	ACC	AUC	Macro-F1	Plan ACC	Direct ACC
MLP scorer	**87.91**	**85.79**	**72.21**	95.06	92.83
Logistic regression	83.21	67.71	60.64	**98.13**	**97.97**

The MLP leads in test accuracy, AUC, and Macro-F1, whereas logistic regression leads in generator-training accuracy. The two scorers therefore have different rankings on the evaluated populations.

A.8 Minimum-Weight Analysis

Table 12 varies the minimum weight on $\mathcal{D}_{\text{direct}}$. The main setting uses $w_{\min} = 0.3$, $w_{\max} = 1.0$, and effective thresholds $(0.5 - 10^{-6}, 0.5)$.

Interpreting the two-level weights. For a collection of N responses with losses ℓ_i as in Equation 3, let $H = \{i : w_i = w_{\max}\}$ be the higher-weight subset and $q_H = |H|/N$. When H is nonempty and $0 < w_{\min} \le w_{\max}$, define $\mathcal{L}_H = |H|^{-1} \sum_{i \in H} \ell_i$. The ordinary SFT loss is $\mathcal{L}_{\text{SFT}} = N^{-1} \sum_i \ell_i$. For weights restricted to $\{w_{\min}, w_{\max}\}$, the objective satisfies

$$\mathcal{L}_{\text{W-SFT}} = \alpha \mathcal{L}_{\text{SFT}} + (1 - \alpha)\mathcal{L}_H, \qquad \alpha = \frac{w_{\min}}{w_{\min} + (w_{\max} - w_{\min})q_H}. \tag{8}$$

To obtain this identity, expand the numerator of Equation 3 as $w_{\min} \sum_i \ell_i + (w_{\max} - w_{\min}) \sum_{i \in H} \ell_i$ and its denominator as $N w_{\min} + (w_{\max} - w_{\min})|H|$. Thus, a positive floor retains a uniform-response loss component while emphasizing the higher-weight subset. If H is empty or contains every response, the objective reduces to ordinary SFT in Equation 1.

Table 12: Minimum-weight ablation on $\mathcal{D}_{\text{direct}}$ (%). Avg./Strict denote AvgAcc/StrictAcc.

	Valid		Test	
Setting	Avg.	Strict	Avg.	Strict
Base	37.96	17.16	41.47	17.33
MODUCODER	38.82	23.96	36.59	17.93
$w_{\min} = 0.0$	35.85	16.24	36.63	19.32
$w_{\min} = 0.1$	41.47	26.90	**46.22**	23.11
$w_{\min} = 0.3$	**43.09**	**28.53**	40.65	20.92
$w_{\min} = 0.5$	32.79	19.29	38.93	21.12
$w_{\min} = 0.8$	33.07	17.26	46.14	**23.41**

Validation selects $w_{\min} = 0.3$, which leads in both Valid metrics. Test StrictAcc peaks at $w_{\min} = 0.8$ and AvgAcc at 0.1, showing nonmonotonic, split-dependent rankings.

A.9 THRESHOLD ANALYSIS

We examine the score-to-weight mapping by varying its lower and upper thresholds in Table 13. Here $\tau = 0.5$, $\epsilon = 10^{-6}$, and τ_q denotes the q-quantile of training scores. The $(\tau - \epsilon, \tau)$ configuration gives the effective thresholds; other threshold configurations give the requested values, with the lower value capped at the upper value minus ϵ before interpolation.

Table 13: Threshold ablation on $\mathcal{D}_{\text{direct}}$ (%). Avg./Strict denote AvgAcc/StrictAcc.

	Valid		Test	
Setting	Avg.	Strict	Avg.	Strict
Base	37.96	17.16	41.47	17.33
MODUCODER	38.82	23.96	36.59	17.93
$(\tau - \epsilon, \tau)$	**43.09**	**28.53**	40.65	**20.92**
$\tau_{0.15} - \tau_{0.4}$	36.45	18.78	38.26	19.72
$\tau_{0.15} - \tau_{0.8}$	33.30	17.26	37.98	19.62
$\tau - \tau_{0.4}$	33.81	21.83	**43.38**	**20.92**
$\tau - \tau_{0.8}$	32.96	20.30	39.23	19.12

The $(\tau - \epsilon, \tau)$ setting leads in both Valid metrics and ties with τ–$\tau_{0.4}$ in Test StrictAcc. The latter yields 43.38% Test AvgAcc, compared with 40.65% for the validation-selected setting.

A.10 STATIC SCORING AND REFLECTION

We assess score-based selection and revision using MODUCODER-Plan on the same 397 Test problems as the module-feedback comparison. Temperature-1.0 candidates share a pool across prefixes.

Each condition is evaluated once, with one final program per problem; Pass@1 is the percentage of successful problems.

Table 14: Score-model post-processing on MODUCODER-Plan outputs over 397 problems from $\mathcal{D}_{\text{plan}}$. Pass@1 values are percentages. Mode pos. denotes the mode of the selected candidates' original generation positions across problems, before reranking.

Prefix k	First	Rerank	Mode pos.	Reflection
1	51.89	51.89	1st	56.68
3	51.89	52.64	2nd	56.93
5	51.89	53.15	2nd	57.43

At $k = 5$, success rises from 51.89% to 53.15% after reranking and 57.43% after reflection. Reflection exceeds reranking at every prefix; reranking success increases across the three evaluated prefixes.

A.11 RULE-BASED CONSISTENCY

We measure correspondence between rationale entities and code identifiers, and between step labels and code operations. Their mean is the rule score. We score responses with complete rationales and parseable code on a 0–1 scale.

Table 15: Rule-based reasoning-code consistency evaluation.

Method	Identifier consistency	Algorithm consistency	Rule score
CoT-code SFT	0.1012	**0.2971**	0.1991
Modular SFT	**0.2646**	0.2160	**0.2403**

Identifier consistency rises from 0.1012 to 0.2646, outweighing the algorithm-consistency decline from 0.2971 to 0.2160. Their mean therefore rises from 0.1991 to 0.2403.

A.12 METRIC AGGREGATION AND REPORTING UNITS

Generation metrics weight problems equally and use nonempty scoring sets; consistency analyses score retained responses. Valid and Test are reported separately, with differences on the 0–1 accuracy scale.

A.13 EVALUATION PROTOCOL

Problem and scoring splits. Section 3.4 specifies scoring sources and splits. Scorer training excludes generator Valid and Test problems.

Training objectives and features. The weighted trainer encodes instructions and responses separately, appends an end-of-sequence token, and masks instruction and padding labels. Shifted logits and labels yield next-token cross entropy, averaged over each response's unmasked tokens. Response losses receive a uniform mean (Equation 1) or normalized weighted mean (Equation 3). Sample IDs index weights computed once before training; normalization uses the batch weight sum.

Feedback access and computation. Revision starts from the highest-scoring prefix candidate and uses temperature 1.0. We execute $\mathcal{T}_x^{\text{fb}}$ to obtain module interfaces, status, and representative failures, and $\mathcal{T}_x^{\text{eval}}$ to score the final program. Prefixes share a candidate pool. Revision adds feedback execution and one model call to the k sampled candidates. Success requires passing every final scoring test for the problem.

B PROMPT TEMPLATES

The prompts below connect an existing global rationale to module responsibilities and executable code. Main-step and substep extraction organize the rationale into module descriptions; modular synthesis specifies dependencies and implementations; reflection uses quality scores and execution evidence to revise the program. Placeholders in braces are replaced by the corresponding problem, reasoning, code, score, or trace fields at run time.

B.1 FROM GLOBAL REASONING TO MODULES AND CODE

System prompt for modular synthesis

You are a helpful assistant in the field of code. Here is a problem needed to be solved:
[Problem]
{problem}

Main-step extraction prompt

Main-Step Extraction
Analyze the given reasoning chain for a math or coding problem and extract every **major** step. Ignore substeps and only keep the top-level insights or actions.
Output format: S1: [First major step]
S2: [Second major step]
S3: [Third major step]
...
Guidelines: Label each top-level step consecutively; capture the complete thought process; keep each item concise; do not introduce sub-numbering; ignore test cases, verification steps, and result checks.
[Original Reasoning]
{original reasoning chain}

Substep extraction prompt

Given all main steps from a reasoning chain, break each main step into internal substeps only when it can be meaningfully decomposed.
Output format:
S1: [Description of step 1]
S2: [Description of step 2]
  S2.1 [Description of step 2.1]
  S2.2 [Description of step 2.2]
Guidelines: Retain the main-step number when listing its substeps; capture the full thought process; do not introduce nesting deeper than two levels; ignore verification-only steps.
[Original Reasoning]
{original reasoning chain}
[Main Steps]
{main steps}

Modular rationale prompt

Restructure the project plan S1, S2, ... into a sequence of highly independent and structurally distinct code modules M1, M2, ... so that each module forms a standalone and executable code block.
Requirements: Identify strict logical and data dependencies; group steps into complete functional stages; preserve original step labels; define execution order, dependency source, and required outputs; ensure every original step is covered exactly once unless it only describes verification.
Output format: Code Execution Sequence. M1: [Module name/function]. Dependencies: None or [specific output]. Functional Description: [description].
[Original Reasoning] {original reasoning chain}
[Detailed Substeps] {substeps}

Modular code generation prompt

First, based on the provided block idea, divide the problem-solving method into modules, including function headers and signatures. Define code modules with clear function names and explicit input-output specifications. Output these function headers and specifications in plain text first. Then write the complete implementation for each defined module.

Modular code generation prompt (continued)

Please generate the modules in the same order as defined in the plan. Every function must process the inputs and explicitly return the result. Place the implementation of each module into its own separate Python code block. The beginning of each function body must match the previously designed function header and signature. Remember to provide a `main()` function so that all blocks can be concatenated and executed successfully.
[Required markers] STEP 1: GENERATE SUB-MODULES. STEP 2: GENERATE PYTHON CODE.
[Task] {problem description}
[Instruction] {code module decomposition}

B.2 Using Quality and Execution Evidence

Score-only reflection prompt

You are revising a draft solution for a programming problem. Only a ranker score is available; no testcase errors or execution traces are provided, so avoid blind rewrites.
[Feedback mode] score_only. ranker_score: {ranker score, higher is better}.
The revised response should keep exact input-output behavior, preserve required signatures and return formats, fix logic bugs, edge cases, and complexity bottlenecks, review modular decomposition granularity, and avoid assumptions not implied by the problem.
[Output format] Return only final Python code in one triple-backtick block with no extra explanation.
[Problem] {problem description}
[Draft solution] {draft code}

Score+errors reflection prompt

You are revising a draft solution for a programming problem. You are given a ranker score and testcase failure signals.
[Feedback mode] error_score. ranker_score: {ranker score, higher is better}.
[Test summary] {total testcases}; {passed testcases}; {failed testcases}; {runtime error count}; {compile error count}; {overall status}.
[Failure logs] {failure log 1}; {failure log 2}.
Infer the likely root cause from test signals before revising code. Keep exact I/O behavior and required signatures. Fix compile or runtime issues first, then correctness and edge cases. Improve complexity if the current approach is too slow. Rewrite from scratch if the draft is fundamentally flawed.

Score+errors+traces reflection prompt

You are revising a draft solution for a programming problem. You are given ranker score, testcase failures, and module-level I/O traces.
[Feedback mode] error_score_io. ranker_score: {ranker score, higher is better}.
[Test summary] {total testcases}; {passed testcases}; {failed testcases}; {runtime error count}; {compile error count}; {overall status}.
[Failure logs] {failure log 1}; {failure log 2}.
[Module I/O traces] {failed module trace 1}; {failed module trace 2}; {passed module trace 1}; {passed module trace 2}.
Use the traces to localize the first likely wrong transformation or module. Preserve modules that look correct on passed traces when possible. Keep exact I/O behavior and required signatures. Fix interface or data-flow mismatches between helper modules, correctness bugs, and edge cases. Keep modular decomposition clear.

LLM-as-a-Judge consistency prompt

You are a strict and objective code reviewer. Evaluate only the consistency between the reasoning text and the final code implementation. Do not score performance, style preference, or readability preference. Return structured JSON only.
Rate each dimension from 0 to 5: step_consistency, reasoning_completeness, logical_consistency, structural_correspondence, and traceability. Also return major_issues and rationale.
[Problem] {problem description}
[Reasoning / CoT] {reasoning}
[Code] {code}

C AN EXAMPLE OF HIERARCHICAL PROGRAM KNOWLEDGE

This example connects a task-level counting objective to module responsibilities and executable helper functions. The global solution splits a signed interval, the interval-sum helper calls a cumulative-counting primitive, and the entry function manages input and output. These relationships illustrate how separate implementations compose through interfaces and data transfers.

C.1 TASK-LEVEL GOAL

Given T queries, each query contains an integer interval $[a, b]$. The task is to count the total number of one bits in the 32-bit two's-complement representations of all integers in the interval. The bounds are $-2^{31} \le a \le b \le 2^{31} - 1$. Cumulative counting handles intervals spanning up to 2^{32} integers.

C.2 MODULE RESPONSIBILITIES

Code execution sequence

M1: Data Input and Test Case Loop
Reads T and interval bounds (a, b) from standard input, passes the bounds to M2, and prints the result returned by M2.
M2: Core Two's-Complement Summation Logic
Receives interval bounds from M1 and calls M4. Splits the interval by sign, maps negative numbers to unsigned 32-bit values, and sums the interval counts returned by M4.
M3: Cumulative Bit Count Function
Receives a bound from M4 and computes the cumulative count directly. Implements `count_bits_sum(N)`, returning the total number of set bits from 0 to N inclusive using bit-periodicity.
M4: Interval Bit Sum Function
Receives bounds from M2 and calls M3. Implements `sum_bits(A,B)` as `count_bits_sum(B) - count_bits_sum(A-1)`, returning 0 when $A > B$.

C.3 EXECUTABLE IMPLEMENTATION

The cumulative helper counts set bits from zero through its argument. At bit position i, a full cycle has length 2^{i+1} and contributes 2^i set bits; the remainder contributes the portion beyond the zero-bit half of the cycle. Subtracting two cumulative counts gives an interval sum. Negative integers are mapped to unsigned 32-bit values before counting. An interval crossing zero is split into its negative and nonnegative parts, and their counts are added. These operations implement the responsibilities assigned to the counting and interval modules.

```python
def read_input():
    import sys
    data = sys.stdin.read().strip().split()
    if not data:
        return []
    T = int(data[0])
    test_cases = []
    idx = 1
    for _ in range(T):
        a = int(data[idx])
        b = int(data[idx + 1])
        test_cases.append((a, b))
        idx += 2
    return test_cases
```

```python
def count_bits_sum(N):
    if N < 0:
        return 0
    total = 0
    i = 0
    while (1 << i) <= N:
        cycle_len = 1 << (i + 1)
        full_cycles = (N + 1) // cycle_len
        contribution = full_cycles * (1 << i)
        remainder = (N + 1) % cycle_len
        contribution += max(0, remainder - (1 << i))
        total += contribution
        i += 1
    return total

def sum_bits(A, B):
    if A > B:
        return 0
    return count_bits_sum(B) - (count_bits_sum(A - 1) if A > 0 else 0)

def twos_complement_range_sum(a, b):
    MOD = 1 << 32
    def to_unsigned(x):
        return x & (MOD - 1)
    total = 0
    if a >= 0:
        return sum_bits(a, b)
    if b < 0:
        return sum_bits(to_unsigned(a), to_unsigned(b))
    total += sum_bits(to_unsigned(a), to_unsigned(-1))
    total += sum_bits(0, b)
    return total

def main():
    for a, b in read_input():
        print(twos_complement_range_sum(a, b))

if __name__ == "__main__":
    main()
```

The cumulative loop visits at most 32 bit positions under the stated input bounds. Each query uses a constant number of cumulative calls, giving $O(32)$ counting time per query and constant auxiliary space for the counting functions. For example, the interval $[-1, 1]$ contributes 32 set bits from -1, zero from 0, and one from 1, for a total of 33.