CODEBLOCK: LEARNING TO SUPERVISE CODE AT THE RIGHT GRANULARITY

Anonymous authors
Paper under double-blind review

ABSTRACT

Supervised fine-tuning of code LLMs typically applies uniform cross-entropy loss to all response tokens, implicitly assuming that every token provides equally useful learning signals. Recent token-level selection methods challenge this assumption in natural-language SFT by supervising only high-value tokens. However, such pointwise selection can fragment the syntactic structures and program dependencies of code, leaving supervision scattered across incomplete code units. In experiments on 30K code instruction–response pairs, under the same 10% token budget, supervising complete coding blocks improves average performance by 11.7 points over prior approaches that supervise isolated tokens. Motivated by this observation, we propose CODEBLOCK, a structure-aware sparse supervision framework that uses complete, parser-aligned coding items as the basic units of supervision. CODEBLOCK constructs coding items from high-quality code instruction data, estimates their supervision utility using GCE, which is more robust to low-probability tokens, and further adjusts their supervision priority using data-flow reach and bridge signals. During training, the full response is retained as context, while loss is applied only to the selected supervision units. Experiments show that partial supervision over only about 6.9% of response tokens consistently outperforms full-token SFT across all five model settings, suggesting that effective code SFT depends not only on identifying high-value supervision, but also on allocating it at the appropriate structural granularity.

1 INTRODUCTION

Supervised fine-tuning (SFT) is a standard way to adapt code LLMs to instruction-following and program-generation tasks (Hui et al., 2024; Guo et al., 2024). As code instruction corpora grow, improving SFT increasingly depends on extracting dense, reliable, and high-marginal-utility supervision rather than simply adding more examples (Wei et al., 2024b; Li et al., 2024; Yu et al., 2025). This has motivated data selection methods that filter high-quality, less noisy, or more diverse instruction–response pairs before training (Chen et al., 2023; Xia et al., 2024; Liu et al., 2023). However, most existing methods remain sample-level: they keep or discard entire instruction–response pairs, while still assigning uniform cross-entropy supervision to every token in a selected response (Chen et al., 2025; Wu et al., 2024).

This assumption has already been challenged in natural-language SFT. Recent token-level selection methods challenge the need for dense supervision by keeping only high-value tokens, often estimated by pointwise loss or excess-loss scores (Pang et al., 2025; Lin et al., 2024a). These methods are effective in natural-language settings because individual tokens can often be treated as approximate local learning units (Lin et al., 2024b; Qin et al., 2025; Fu et al., 2026). However, directly transferring pointwise token selection from natural-language settings to code SFT is unreliable. Unlike ordinary text, code is organized by a formal grammar, exposes mechanically recoverable local constructs and definition–use relations, and is evaluated by executing the generated program (Allamanis et al., 2017; Guo et al., 2020). An isolated variable name or operator does not specify the assignment, branch condition, API call, or return expression in which it participates. Figure 2 illustrates this granularity mismatch: pointwise token selection tends to pick scattered tokens from different statements, whereas structure-aware selection preserves materialized parser-guided items that form locally meaningful program evidence. In Sec. 4, we connect this structural distinction to the learning objective: the summed teacher-forced loss over a materialized item bounds the parser or test failure of a reference-

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.

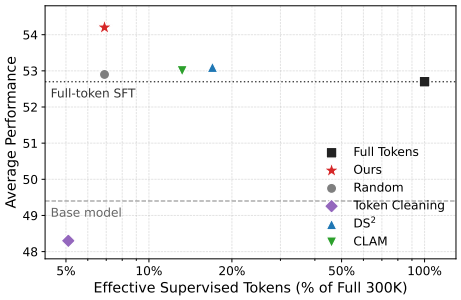


Figure 1: Performance–efficiency trade-off on Qwen2.5-Coder-1.5B-Instruct. CODEBLOCK achieves the highest average performance with only 6.9% effective supervised tokens, showing a better trade-off than full-token SFT and sparse-selection baselines.

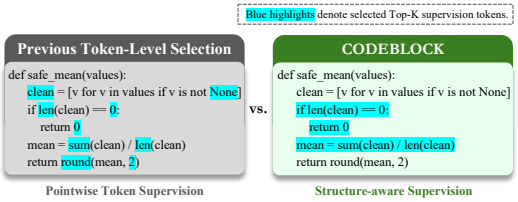


Figure 2: Comparison between previous token-level selection and CODEBLOCK. While prior methods select isolated high-score tokens, CODEBLOCK supervises materialized parser-guided coding items and uses their data dependencies for selection.

conditioned local greedy replacement, and full-item supervision places every term of this bound inside the masked objective. A matched-budget diagnostic then verifies that this granularity is beneficial in downstream code generation. Together, these results motivate using token scores to locate valuable evidence while allocating the final supervision at coding-item granularity.

To this end, we propose CODEBLOCK, a structure-aware sparse supervision framework for code LLM fine-tuning. CODEBLOCK uses high-scoring tokens only as anchors for locating useful code evidence: it constructs parser-guided coding items, scores each item by the concentration of informative logic tokens using generalized cross-entropy, and then reranks items with lightweight data-flow signals that measure downstream influence and dependency-path connectivity. During training, the full response remains available as autoregressive context, while the loss is applied only to selected code items and informative natural-language tokens. As shown in Figure 1, across six code-generation benchmarks and five model settings, CODEBLOCK consistently matches or improves full-token SFT while using only about 6.9% supervised response tokens, and achieves the best or second-best average performance among competitive selection baselines.

The contributions of this paper are as follows:

- On a 30K diagnostic subset at the top-10% token budget, pointwise selection leaves 92.3% of touched coding items only partially covered. Under the same code-token budget, supervising complete materialized items improves average performance by 11.7 points over isolated top-token supervision. A reference-conditioned local bound further characterizes the loss terms omitted by partial item supervision.
- We propose CODEBLOCK, a structure-aware sparse supervision framework that selects coding items rather than individual tokens, combining item partitioning, GCE-based utility scoring, and data-flow-aware reranking.
- Experiments across six code-generation benchmarks show that CODEBLOCK matches or outperforms full-token SFT with only about 6.9% supervised response tokens, achieving a stronger performance–efficiency trade-off than competitive baselines.

2 RELATED WORK

LLM Data Selection. Instruction-tuning performance depends strongly on data quality and composition (Deng et al., 2025). Existing methods select training samples using LLM-based quality or difficulty signals, as in AlpaGasus, DEITA, IFD, DS2, D3 and Scar (Chen et al., 2023; Liu et al., 2023; Li et al., 2024; Pang et al., 2024; Zhang et al., 2025; Li et al., 2025); influence-based objectives, as in LESS and NICE (Xia et al., 2024; Wang et al., 2025b); or diversity and information criteria, as in NovelSum and MIG (Yang et al., 2025; Chen et al., 2025). Despite different selection signals, these methods operate primarily at the sample level and retain complete responses once an instruction–response pair is selected.

For code LLMs, data curation similarly emphasizes execution validity, redundancy reduction, and quality–diversity trade-offs. SelfCodeAlign retains self-generated responses through sandbox verifi-

cation (Wei et al., 2024a), while CSAM, distribution-consistent selection, and XCoder filter or select code instruction data based on redundancy, distribution and diversity, or instruction complexity and response quality (Tsai et al., 2024; Lyu et al., 2025; Wang et al., 2024). These approaches improve which instruction–response pairs enter training, but still treat each selected response as a uniform supervision unit. In contrast, CODEBLOCK studies supervision within selected code responses, determining which complete code fragments should contribute training gradients while retaining the full response as context.

Fine-Grained Supervision Selection and Data Cleaning. Beyond sample-level filtering, recent works increasingly exploit token-level training signals. Rho-1 selectively applies loss to valuable pretraining tokens, while Token Cleaning and TOSS mask uninformative or unsafe SFT tokens (Lin et al., 2024a; Pang et al., 2025; Li et al., 2026). T-SHIRT incorporates token informativeness into hierarchical instruction-data selection, TokenTune jointly models token- and sample-level utility with gated optimization, and MSL masks invalid output tokens in recommendation fine-tuning (Fu et al., 2026; Lin et al., 2026; Wang et al., 2025a). These works demonstrate the value of supervision below the sample level, but largely score or mask tokens as pointwise units. For code responses, supervision utility additionally depends on syntactic units, definition–use relations, and data-flow dependencies, motivating structure-aware selection over complete code fragments.

3 PRELIMINARY: NEXT-TOKEN PREDICTION WITH SPARSE SUPERVISION

Given an instruction-tuning dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, where p_θ denotes the model's next-token distribution, x_i is the instruction, and $y_i = \{y_{i,t}\}_{t=1}^{T_i}$ is the response, we formulate supervised fine-tuning with a token-level supervision mask. For each response token $y_{i,t}$, let $m_{i,t} \in \{0, 1\}$ indicate whether this token contributes to the training loss. The masked next-token prediction objective is defined as:

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

This formulation generalizes standard full-token SFT and sparse supervision. When $m_{i,t} = 1$ for all response tokens, the objective reduces to standard SFT, where every response token is supervised. Sparse supervision instead sets $m_{i,t} = 0$ for most tokens by default and only enables selected tokens with $m_{i,t} = 1$. Tokens with $m_{i,t} = 0$ are still kept in the autoregressive context, but they are ignored in the loss computation.

Prior fine-grained selection methods (Pang et al., 2025; Lin et al., 2024a) usually construct $m_{i,t}$ by scoring individual tokens, for example using token-level loss or excess-loss signals, and then retaining high-scoring tokens. However, this pointwise view is less suitable for code, where token roles are organized by parser-defined constructs and definition–use dependencies. This motivates our later use of materialized parser-guided coding items as the basic units for sparse supervision.

4 MOTIVATING ANALYSIS

Token-level selection assumes that high-scoring tokens are suitable training targets, but this assumption may not hold for code, where token semantics often depend on surrounding syntactic and dependency structures. We therefore first examine whether high-scoring code tokens can directly serve as sparse supervision units.

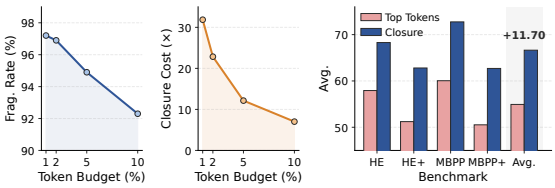


Figure 3: Motivating analysis of token-level sparse supervision in code. Left: fragmentation rate of coding items under different token budgets. Middle: closure cost for expanding selected tokens into materialized local fragments. Right: same-budget mini-SFT results comparing isolated top-token supervision with materialized-item supervision.

Definition of coding items. For each code response y_i, we partition its code regions into parser-guided local units $\mathcal{U}_i = \{u_{i,1}, \ldots, u_{i,K_i}\}$. A coding item u is a coherent local computation, such as an assignment, branch condition, API call, or return expression. We denote its core logic tokens

(identifiers, literals, operators, and API names) by $C(u)$ and its contiguous materialized span by $M(u)$; the latter also retains associated boundary and protected syntax positions. Thus $C(u) \subseteq M(u)$, and an item is fully materialized when all positions in $M(u)$ are supervised. Section 5.2 gives the operational construction.

Pointwise selection fragments coding items. We sample 30K code responses from OpenCode-Instruct (Ahmad et al., 2025) and use a frozen Qwen2.5-Coder-1.5B-Instruct (Hui et al., 2024) to compute token-level CE scores. At each budget, the top-scoring code positions A are mapped to their enclosing items. We measure the fraction of touched items that are only partially covered and the closure expansion $R = |\operatorname{Cl}(A)|/|A|$, where $\operatorname{Cl}(A) = \bigcup_{u:M(u)\cap A\neq\emptyset} M(u)$. Figure 3 shows that, at the top-10% budget, 92.3% of touched items are incomplete and materializing their local spans requires $7.02\times$ as many positions.

From materialized-item loss to local parser–test failure. Code provides two mechanically checkable outcomes: parser acceptance and test-suite success (Aho & Ullman, 1972; Chen et al., 2021). We analyze an extracted executable code region and identify each model-token sequence with its decoded source string. Let $\operatorname{Pass}_{\mathcal{T}}(z) \in \{0, 1\}$ equal one exactly when z executes and passes every test in $\mathcal{T}$, and zero for a syntax, runtime, or test failure. Let $y = (y_1, \ldots, y_T) \in \operatorname{Lang}(G)$ be a reference program with $\operatorname{Pass}_{\mathcal{T}}(y) = 1$, and define the teacher-forced loss $\ell_t(\theta) = -\log p_\theta(y_t \mid x, y_{<t})$. For $M(u) = [s, e]$, greedily decode exactly $|M(u)|$ tokens from the reference prefix $y_{<s}$, treating an emitted end-of-sequence token as a mismatch. Denote the continuation by $\hat{y}_{M(u)}$, splice it into the reference as $\tilde{y}^{(u)} = y_{<s} \circ \hat{y}_{M(u)} \circ y_{>e}$, and define

$$F_{G,\mathcal{T}}(\tilde{y}^{(u)}) = \mathbf{1}\left[\tilde{y}^{(u)} \notin \operatorname{Lang}(G) \ \vee \ \operatorname{Pass}_{\mathcal{T}}(\tilde{y}^{(u)}) = 0\right].$$

Proposition 1 (Reference-conditioned local parser–test failure bound)**.** *For any model parameters θ and any parser-guided coding-item span $M(u)$,*

$$F_{G,\mathcal{T}}(\tilde{y}^{(u)}) \leq \mathbf{1}[\hat{y}_{M(u)} \neq y_{M(u)}] \leq \min\left\{1, \frac{\sum_{t\in M(u)} \ell_t(\theta)}{\log 2}\right\}. \tag{2}$$

The first inequality holds because exact item recovery restores the valid, test-passing reference program. For the second, the first greedy mismatch occurs under the same prefix as teacher forcing; the selected incorrect token has probability at least that of the reference token, so the reference-token probability is at most $1/2$ and its loss is at least $\log 2$. Appendix A gives the complete proof.

Let $S = \{t : m_t = 1\}$ be the supervised positions. If $M(u) \subseteq S$, every loss term in the bound appears in the numerator of the masked objective; if $S \cap M(u) \subsetneq M(u)$, at least one term is omitted. For example, omitting $\star$ from `total = price * qty` leaves an operator loss outside the objective even though replacing it can preserve parsing but change the test outcome. This provides a local diagnostic motivation for allocating code supervision at item rather than isolated-token granularity.

Matched-budget mini-SFT. On the same 30K subset, we fine-tune Qwen2.5-Coder-1.5B-Instruct for 3,000 steps and compare isolated top-CE supervision with materialized-item supervision. The item arm maps top-CE anchors to their $M(u)$ spans; if their union contains K distinct code positions, the pointwise arm supervises exactly the top-K CE positions. The two arms therefore use the same number of supervised code tokens. Across HumanEval, HumanEval+, MBPP, and MBPP+, materialized-item supervision improves the average by 11.70 points (Figure 3, right), supporting item-level allocation under a controlled budget.

5 METHOD

In this section, we propose CODEBLOCK, a structure-aware sparse supervision method for supervised fine-tuning of code LLMs. As illustrated in Figure 4, CODEBLOCK consists of four main components: sample-level selection, GCE-based coding item scoring, data-flow-guided item selection, and sparse fine-tuning.

5.1 SAMPLE-LEVEL SELECTION

We first select a compact training subset from the large code instruction pool. Each instruction-response pair is scored by Qwen2.5-Coder-32B-Instruct (Hui et al., 2024) for instruction following and

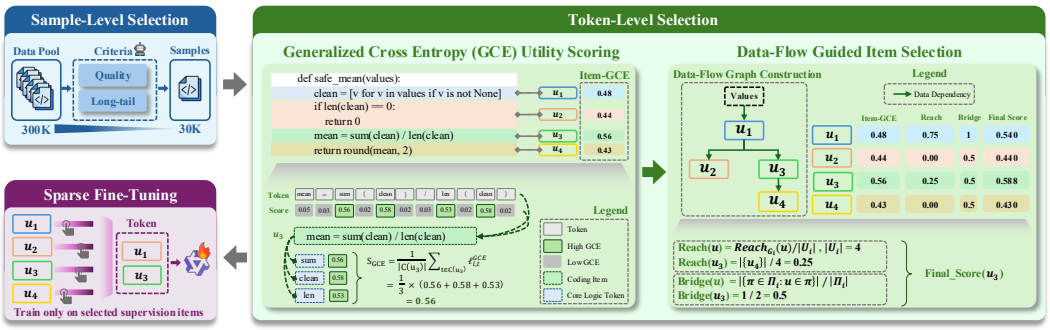


Figure 4: Overview of CODEBLOCK, which combines sample-level selection, GCE-based coding item scoring, data-flow guided item selection, and sparse fine-tuning for efficient code LLM supervised fine-tuning.

code correctness, and by a lightweight long-tail score based on structural/task features such as control flow, API usage, and code complexity. We rank sample i by $s_i^{\text{sample}} = \alpha s_i^{\text{qual}} + (1-\alpha)s_i^{\text{tail}}$, Here, s_i^{qual} is a normalized quality score derived from LLM-judge ratings, with a small auxiliary calibration term from available unit test scores. The long-tail score s_i^{tail} measures the rarity of structural and task-level features, including API usage, control-flow patterns, and coarse code-complexity buckets. More selection details are provided in Appendix C.1. After deduplication, the top 30K samples form $\mathcal{S}$ for coding-item-level supervision selection.

5.2 GCE-BASED CODING ITEM SCORING

After sample selection, CODEBLOCK selects where to apply supervision within each response. For natural-language regions, we directly apply token-level GCE scoring and keep the top-ρ_{NL} fraction of tokens. For code regions, we first construct coding items through a lightweight parser-guided local-closure procedure, as detailed in Algorithm 1. Specifically, we extract code regions from each response and parse them with Tree-sitter[1], then align syntax spans back to model-token indices using character offsets. Each token is labeled as a core logic token, a protected syntax token, or an other token. Core logic tokens, such as identifiers, literals, arithmetic/comparison operators, and API names, form the core set $C(u)$ of a coding item. Protected syntax tokens, such as brackets, commas, colons, delimiters, and statement boundaries, are retained when materializing $M(u)$, while other tokens act as separators. During the left-to-right scan, we close the current item when reaching a top-level statement boundary, crossing an unrelated separator, or reaching the end of the code region. For a terminated scan segment $[a, b]$, let $Q = C_{\text{cur}} \cup \{j \in [a, b] : z_j = \text{P}\}$. The LOCALCLOSURE rule is the deterministic interval $M(u) = [\min Q, \max Q]$: an O separator is excluded from $[a, b]$, while a closing boundary labeled P is included. Let $B(u) = M(u) \setminus C(u)$ denote the retained boundary and protected positions, so $M(u) = C(u) \dot{\cup} B(u)$. Following the motivation in Sec. 4, we estimate utility over $C(u)$ but apply supervision to the materialized span $M(u)$.

For a coding item u, we define its utility by averaging token-level GCE scores over its core logic tokens:

$$S_{\text{GCE}}(u) = \frac{1}{|C(u)|} \sum_{t \in C(u)} \ell_{i,t}^{\text{GCE}}, \tag{3}$$

where $\ell_{i,t}^{\text{GCE}} = \frac{1-p_{i,t}^q}{q}$, $p_{i,t} = p_{\theta_0}(y_{i,t} \mid x_i, y_{i,<t})$, and θ_0 is the frozen base model. Here, $q \in (0, 1]$ controls the degree of loss tempering.

We use GCE rather than raw CE to temper extremely low-probability outliers, which in code often correspond to rare identifiers, unusual literals, or noisy fragments. The item score is computed on $C(u)$ to measure uncertainty over logic-bearing tokens, while supervision is applied to the materialized span $M(u)$.

[1] https://tree-sitter.github.io/tree-sitter/

5.3 DATA-FLOW GUIDED ITEM SELECTION

GCE identifies coding items that are uncertain under the base model, but uncertainty alone does not necessarily reflect structural importance. For example, a rare variable name or literal may obtain a high GCE score while affecting no later computation, whereas an intermediate assignment with a moderate GCE score may be reused by several subsequent statements and eventually determine the returned value. Therefore, CODEBLOCK further incorporates data-flow structure to prioritize items that are both learnable and structurally consequential. Following the code-representation view that definition–use relations expose how values propagate toward later computations (Guo et al., 2020), the two signals below operationalize downstream reach and path connectivity as lightweight ranking cues. They enter the priority as a gated bonus with a small weight rather than as the primary utility criterion.

For each code response, we construct a lightweight data-flow graph $\mathcal{G}_i = (\mathcal{U}_i, \mathcal{E}_i)$, where each node is a coding item and an edge $(u, v) \in \mathcal{E}_i$ indicates that item v uses a value defined or updated by item u. Based on this graph, we compute two normalized structural signals:

- **Reach** measures the downstream influence of an item. It captures whether the value defined or updated by item u affects many later computations. Formally, we define $\mathrm{reach}(u) = |\mathrm{Reach}_{\mathcal{G}_i}(u)|/|\mathcal{U}_i|$, where $\mathrm{Reach}_{\mathcal{G}_i}(u)$ denotes the set of items reachable from u in $\mathcal{G}_i$.
- **Bridge** measures dependency-path connectivity. It captures whether item u lies on important dependency chains that connect early definitions to terminal computations. Let Π_i denote the set of source-to-sink paths in $\mathcal{G}_i$, where source nodes introduce values and sink nodes correspond to return statements, printed outputs, or final assignments. We define $\mathrm{bridge}(u) = |\{\pi \in \Pi_i : u \in \pi\}|/|\Pi_i|$ when $\Pi_i \neq \emptyset$, and $\mathrm{bridge}(u) = 0$ otherwise.

We combine GCE utility with these data-flow signals through a gated priority function:

$$P_{\text{CODEBLOCK}}(u) = S_{\mathrm{GCE}}(u)\Big[1 + g(u)\lambda\big(\alpha_r \,\mathrm{reach}(u) + \alpha_b \,\mathrm{bridge}(u)\big)\Big]. \tag{4}$$

where $g(u) = \mathbf{1}\left[S_{\mathrm{GCE}}(u) \geq Q_\eta^{(i)}\right]$. Here, $Q_\eta^{(i)}$ is the within-response GCE threshold for selecting the top-η fraction of coding items, λ controls the overall strength of the data-flow bonus, and α_r and α_b control the relative contributions of reach and bridge. Because our data-flow graph is constructed by lightweight static analysis, its structural signals may be approximate. The gate prevents structurally central but low-utility items from being over-selected due to noisy data-flow estimates, ensuring that data-flow only reranks items that are already sufficiently informative under GCE.

Finally, we derive the code-side supervision mask from the priority scores. Let $\mathrm{Top}_{\rho_{\mathrm{code}}}(\mathcal{U}_i; P_{\text{CODEBLOCK}})$ return the highest-priority coding items whose materialized spans $M(u)$ fit within the code-token budget ratio ρ_{code}. The selected code positions and the corresponding supervision mask are defined as:

$$\mathcal{M}_i^{\mathrm{code}} = \bigcup_{u \in \mathrm{Top}_{\rho_{\mathrm{code}}}(\mathcal{U}_i; P_{\text{CODEBLOCK}})} M(u), \; m_{i,t}^{\mathrm{code}} = \mathbf{1}\left[t \in \mathcal{M}_i^{\mathrm{code}}\right]. \tag{5}$$

Here, $M(u)$ is the materialized local span of item u. Tokens with $m_{i,t}^{\mathrm{code}} = 1$ contribute to the sparse next-token prediction loss, while the remaining code tokens are kept as autoregressive context but ignored in the loss computation.

5.4 SPARSE FINE-TUNING

Given the selected subset $\mathcal{S}$, we instantiate the sparse supervision mask defined in Sec. 3 by combining the code-side and natural-language-side masks. For code regions, we use $m_{i,t}^{\mathrm{code}}$ from the data-flow-guided item selection in Sec. 5.3. For natural-language regions, we keep the top-ρ_{NL} tokens according to GCE scores, denoted by $m_{i,t}^{\mathrm{NL}}$. The final mask is

$$m_{i,t} = \mathbb{1}\left[m_{i,t}^{\mathrm{code}} = 1 \vee m_{i,t}^{\mathrm{NL}} = 1\right]. \tag{6}$$

We then optimize the masked next-token prediction objective in Eq. (1), with unselected response tokens kept in the input sequence but ignored by the loss.

Table 1: Performance comparison of different baselines on various benchmarks. We highlight the best result in **boldface** and the second-best with underline.

Method	HumanEval	HumanEval+	MBPP	MBPP+	BCB-hard	BCB-full	Eff. Tokens (%)	Average
Base model: Qwen2.5-Coder-1.5B-Instruct								
BASE	71.3	67.1	69.6	59.8	4.1	24.8	–	49.4
FULL TOKENS	75.6	69.5	73.3	62.4	5.4	29.8	100.0	52.7
RANDOM SELECTION	73.8	67.7	**75.7**	**64.6**	5.4	30.4	6.9	52.9
DS^2	74.4	69.5	73.5	62.7	7.4	**31.2**	17.0	53.1
TOKEN CLEANING	73.2	66.5	65.6	56.4	5.4	22.5	**5.1**	48.3
CLAM	73.2	68.9	74.1	63.8	**8.1**	29.7	13.2	53.0
CODEBLOCK	**77.4**	**73.2**	75.4	63.2	6.1	29.9	6.9	**54.2**
Base model: Seed-Coder-8B-Instruct								
BASE	78.7	68.3	82.0	69.0	25.7	51.3	–	62.5
FULL TOKENS	80.5	75.6	82.8	70.8	28.3	**52.8**	100.0	65.1
RANDOM SELECTION	79.9	76.2	**86.5**	**73.8**	25.0	45.1	6.9	64.4
DS^2	80.5	**76.8**	86.0	73.0	25.7	45.1	17.0	64.5
TOKEN CLEANING	72.0	65.9	77.5	65.3	14.9	37.3	**5.1**	55.5
CLAM	79.3	73.8	**86.5**	73.3	25.7	44.8	13.2	63.9
CODEBLOCK	**82.3**	**76.8**	85.7	71.7	**30.4**	46.0	6.9	**65.5**
Base model: OpenCoder-8B-Base								
BASE	67.1	59.8	79.6	68.3	11.5	40.9	–	54.5
FULL TOKENS	64.0	60.9	78.0	**68.7**	12.8	**42.8**	100.0	54.5
RANDOM SELECTION	70.7	62.8	79.9	68.3	11.5	40.7	6.9	55.7
DS^2	71.3	63.4	**80.7**	67.5	**15.5**	38.7	17.0	56.2
TOKEN CLEANING	71.3	65.2	78.8	67.7	12.2	37.7	**5.1**	55.5
CLAM	73.8	**67.1**	78.6	68.5	14.2	39.7	13.2	57.0
CODEBLOCK	**76.2**	**67.1**	80.2	67.5	13.5	40.3	6.9	**57.5**

Table 2: We remove key components of CODEBLOCK to evaluate their contributions. The best result is shown in **bold** and the second-best result is underlined.

Method	HumanEval	HumanEval+	MBPP	MBPP+	BCB-hard	BCB-full	average
Base model: Qwen2.5-Coder-1.5B-Instruct							
CODEBLOCK	**77.4**	**73.2**	**75.4**	63.2	6.1	29.9	**54.2**
TOKEN-OFF	73.2	69.5	73.3	63.2	5.4	**30.1**	52.5
SAMPLE-OFF	76.8	70.1	72.2	62.4	4.1	29.2	52.5
GCE-ONLY	**77.4**	**73.2**	74.6	63.0	4.1	29.0	53.5
DATAFLOW-ONLY	75.6	70.1	73.3	**63.5**	2.7	28.4	52.3

6 EXPERIMENTS

6.1 EXPERIMENTS SETUP

Dataset. We construct our dataset by randomly sampling 300K instruction-tuning examples from OpenCodeInstruct (Ahmad et al., 2025). All training subsets and baseline variants are derived from this same pool to ensure a controlled comparison across data selection strategies. In the sample-level selection stage, we rank this pool and retain the top 30K examples as the curated training subset.

Base Models. We evaluate our method on a diverse set of open-source code and general instruction-tuned LLMs, including Qwen2.5-Coder-1.5B-Instruct, Qwen2.5-Coder-3B-Instruct, Qwen2.5-Coder-7B-Instruct (Hui et al., 2024), Seed-Coder-8B-Instruct (Seed et al., 2025), and OpenCoder-8B-Base (Huang et al., 2025). These models are fine-tuned using samples selected from our data pool under the corresponding experimental settings.

Baselines. We compare our method against several representative data selection and token selection baselines. 1) Full Tokens fine-tunes the model on all 300K examples from the source pool, with supervision applied to every response token. 2) Random samples training examples or response tokens uniformly at random under the same data or token budget. 3) DS2 (Pang et al., 2024) selects a curated subset using score-based data selection that favors high-quality and diverse examples. 4) Token Cleaning (Pang et al., 2025) removes low-utility tokens according to a fixed-model token-level filtering criterion before fine-tuning. 5) CLAM (Tsai et al., 2024) performs clustering- and diversity-based sample selection to construct a representative training subset. All baselines are trained with the same data pool, the same one-epoch budget, and the same optimizer configuration, and differ only in which samples or response tokens contribute to the loss.

Evaluation. We evaluate the fine-tuned models on six code generation benchmarks, including HumanEval, HumanEval+ (Chen et al., 2021), MBPP, MBPP+ (Austin et al., 2021), BigCodeBench-

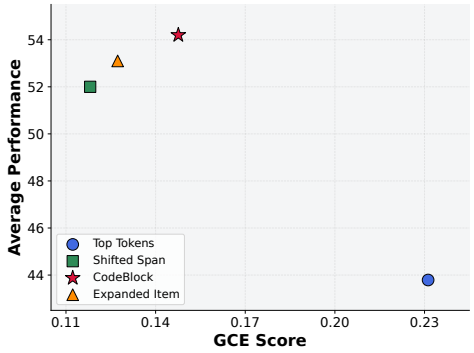


Figure 5: Performance under different code supervision granularities as a function of mean GCE score.

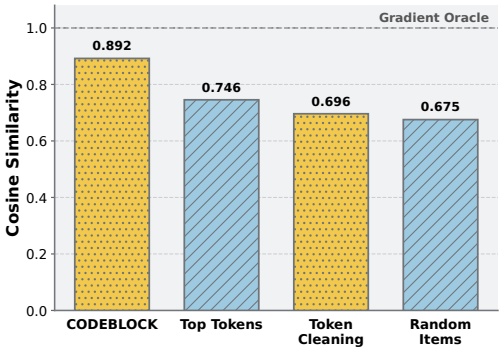


Figure 6: Cosine similarity between each method's aggregate gradient and the per-item Gradient Oracle.

Hard, and BigCodeBench-Full (Zhuo et al., 2025). These benchmarks provide a broad assessment of code generation performance, covering functional correctness, instruction following, and robustness across programming tasks of varying difficulty. We report pass@1 for each benchmark and use the average score over the six tasks as the main metric. Unless otherwise specified, we use the default settings in Table 3 for all experiments. More evaluation and training details are provided in Appendix C.4.

6.2 Main Results

CodeBlock achieves the best performance–token trade-off. CodeBlock achieves the best average performance across all three models using only 6.9% effective supervised tokens. On Qwen2.5-Coder-1.5B-Instruct, Seed-Coder-8B-Instruct, and OpenCoder-8B-Base, CodeBlock obtains average scores of 54.2, 65.5, and 57.5, outperforming Full Tokens by 1.5, 0.4, and 3.0 points, respectively. Under the same supervision budget, CodeBlock also consistently outperforms Random Selection, indicating that the gains arise from selecting more informative supervision targets rather than sparsity alone. It further surpasses DS^2 and CLAM while using substantially fewer supervised tokens, demonstrating a better performance–efficiency trade-off than dense full-token supervision and existing selection baselines.

Parser-guided item allocation benefits token selection in code. Although Token Cleaning is designed as a general token-level data selection method, its performance is unstable in the code domain. It underperforms the corresponding base model on two of the three model settings, with average scores dropping from 49.4 to 48.3 on Qwen2.5-Coder-1.5B-Instruct and from 62.5 to 55.5 on Seed-Coder-8B-Instruct. This observation suggests that directly selecting isolated tokens may be less suitable for code, where the usefulness of a token is often coupled with the local syntactic and semantic structure in which it appears. In contrast, CodeBlock allocates supervision at the level of parser-guided code items and preserves structurally coherent supervision regions, resulting in substantially more stable improvements across different models and benchmarks.

CodeBlock is effective on both base and instruction-tuned models. As shown in Table 1, CodeBlock achieves the highest average performance across both instruction-tuned models and the OpenCoder-8B base model. This suggests that the effectiveness of our sparse supervision strategy is not restricted to a particular model initialization: it can further improve models that have already undergone instruction tuning while also providing substantial gains for a base model. Additional results on Qwen2.5-Coder-3B-Instruct and Qwen2.5-Coder-7B-Instruct are reported in Table 6, further demonstrating the generality of CodeBlock across model scales.

6.3 Ablation Study

Component Ablation Study. We conduct component ablations on Qwen2.5-Coder-1.5B-Instruct under the same training and evaluation setting as the main experiment. Token-off removes token-level selection, sample-off uses randomly selected samples, while GCE-only and dataflow-only retain only the corresponding scoring components. As shown in Table 2, removing any component reduces the average performance, with CodeBlock achieving the best overall result. GCE-only performs

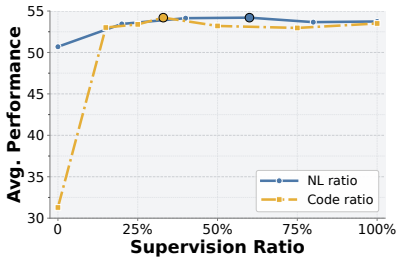


Figure 7: Sensitivity analysis of NL and code supervision ratios.

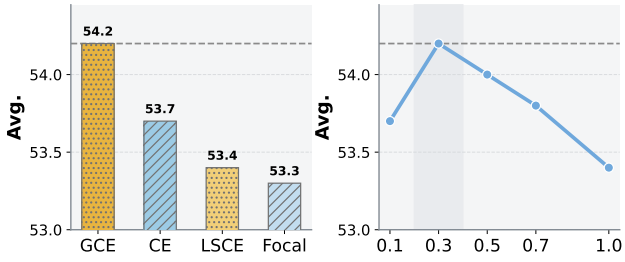


Figure 8: Sensitivity analysis of token scoring sources (left) and the gate ratio η (right).

better than dataflow-only, suggesting that GCE provides the primary utility signal while data-flow information offers complementary structural guidance.

What Is the Right Granularity for Code Supervision? In our main experiments, we use parser-aligned code items as the basic units of supervision. To examine whether this granularity is appropriate, we compare different supervision granularities under the same supervision budget. Top Tokens supervises isolated tokens with the highest GCE scores; Shifted Span groups them into contiguous, length-matched spans while shifting the spans away from the original item boundaries; CODEBLOCK preserves complete parser-aligned code items; and Expanded Item further includes their directly neighboring items. As shown in Figure 5, Top Tokens achieves the highest token-level GCE scores but performs substantially worse than the structurally coherent variants. CODEBLOCK achieves the best overall performance, while both fragmented supervision and unnecessary expansion are less effective. These results suggest that effective code supervision depends not only on token-level utility, but also on preserving an appropriate level of local structural coherence.

Approximating the Gradient Oracle. We further examine whether CODEBLOCK can recover the selection behavior of an expensive per-item Gradient Oracle without explicitly computing gradients for every candidate item. Specifically, we construct the oracle over 23,346 coding items from 1,024 fixed candidate examples using a held-out APPS gradient, and compare all methods under the same code-only supervision space and 33% per-example budget across 128 paired eight-example shards. We measure the cosine similarity between each method's aggregate gradient and the oracle-selected gradient. As shown in Figure 6, CODEBLOCK achieves substantially higher cosine similarity than both token-level baselines and random selection. These results suggest that structured coding-item selection provides an effective low-cost proxy for gradient-based oracle selection, capturing substantially more of the oracle-induced optimization direction than isolated token-level selection.

Sensitivity Analysis. We analyze the sensitivity of CODEBLOCK to supervision ratios, token scoring, and gating, as shown in Figures 7 and 8. For supervision ratios, performance improves as more informative NL and code tokens are retained, peaking at the default $\rho_{\text{NL}} = 0.6$ and $\rho_{\text{code}} = 0.33$; further increasing either ratio provides no additional benefit, suggesting that excessive supervision introduces redundant tokens. For token scoring, GCE performs best, outperforming CE, LSCE, and focal loss. Similarly, a moderate gate ratio is preferable, with $\eta = 30\%$ achieving the best performance, while both smaller and larger values degrade results. Overall, these results support using selective supervision, GCE-based scoring, and moderate data-flow gating.

7 CONCLUSION

In this paper, we proposed CODEBLOCK, a structure-aware sparse supervision framework for code LLM fine-tuning. CODEBLOCK selects materialized parser-guided coding items rather than isolated high-scoring tokens, and prioritizes them with GCE utility and data-flow signals. Experiments across multiple code-generation benchmarks show that CODEBLOCK achieves competitive or better performance than full-token SFT and strong selection baselines while using far fewer supervised response tokens. These results support preserving parser-guided local spans and data-flow context in sparse code supervision, rather than treating tokens as independent learning targets. By keeping full responses as context and applying loss only to informative, dependency-critical code fragments, CODEBLOCK decouples contextual exposure from gradient supervision and offers a practical route toward efficient code model adaptation.

AI USE STATEMENT

In this work, we used generative AI tools for language polishing and for assisting with code drafting and debugging. All AI-assisted content was reviewed and revised by the authors, and generated code was verified through manual inspection and experimental testing. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We will release the source code and experimental implementation upon acceptance of the paper to facilitate reproducibility and future research.

REFERENCES

Wasi Uddin Ahmad, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Vahid Noroozi, Somshubra Majumdar, and Boris Ginsburg. Opencodeinstruct: A large-scale instruction tuning dataset for code llms. *arXiv preprint arXiv:2504.04030*, 2025.

Alfred V Aho and Jeffrey D Ullman. *The theory of parsing, translation, and compiling*, volume 1. Prentice-Hall Englewood Cliffs, NJ, 1972.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Lichang Chen, Shiyang Li, Jun Yan, Hai Wang, Kalpa Gunaratna, Vikas Yadav, Zheng Tang, Vijay Srinivasan, Tianyi Zhou, Heng Huang, et al. Alpagasus: Training a better alpaca with fewer data. *arXiv preprint arXiv:2307.08701*, 2023.

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

Yicheng Chen, Yining Li, Kai Hu, Ma Zerun, HaochenYe HaochenYe, and Kai Chen. Mig: Automatic data selection for instruction tuning by maximizing information gain in semantic space. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 9902–9915, 2025.

Zhijie Deng, Zhouan Shen, Ling Li, Yao Zhou, Zhaowei Zhu, Yanji He, Wei Wang, and Jiaheng Wei. Lm-mixup: Text data augmentation via language model based mixup. *arXiv preprint arXiv:2510.20449*, 2025.

Yanjun Fu, Faisal Hamman, and Sanghamitra Dutta. T-shirt: Token-selective hierarchical data selection for instruction tuning. *Advances in Neural Information Processing Systems*, 38:113932–113958, 2026.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yifan Wu, YK Li, et al. Deepseek-coder: when the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, et al. Opencoder: The open cookbook for top-tier code large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 33167–33193, 2025.

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. *arXiv preprint arXiv:2409.12186*, 2024.

Ming Li, Yong Zhang, Zhitao Li, Jiuhai Chen, Lichang Chen, Ning Cheng, Jianzong Wang, Tianyi Zhou, and Jing Xiao. From quantity to quality: Boosting llm performance with self-guided data selection for instruction tuning. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 7602–7635, 2024.

Yanping Li, Zhening Liu, Zijian Li, Zehong Lin, and Jun Zhang. Token-level data selection for safe llm fine-tuning. *arXiv preprint arXiv:2603.01185*, 2026.

Zhuang Li, Yuncheng Hua, Thuy Vu, Haolan Zhan, Lizhen Qu, and Gholamreza Haffari. Scar: Data selection via style consistency-aware response ranking for efficient instruction-tuning of large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12756–12790, 2025.

Xiaotian Lin, Yanlin Qi, Yuxiang Luo, Themis Palpanas, and Yuyu Luo. Tokentune: Dual-level utility estimation for scalable data selection in instruction tuning, 2026. URL `https://openreview.net/forum?id=LoPF9Zl7ic`.

Zhenghao Lin, Zhibin Gou, Yeyun Gong, Xiao Liu, Yelong Shen, Ruochen Xu, Chen Lin, Yujiu Yang, Jian Jiao, Nan Duan, et al. Rho-1: Not all tokens are what you need. *arXiv preprint arXiv:2404.07965*, 2024a.

Zicheng Lin, Tian Liang, Jiahao Xu, Qiuzhi Lin, Xing Wang, Ruilin Luo, Chufan Shi, Siheng Li, Yujiu Yang, and Zhaopeng Tu. Critical tokens matter: Token-level contrastive estimation enhances llm's reasoning capability. *arXiv preprint arXiv:2411.19943*, 2024b.

Wei Liu, Weihao Zeng, Keqing He, Yong Jiang, and Junxian He. What makes good data for alignment? a comprehensive study of automatic data selection in instruction tuning. *arXiv preprint arXiv:2312.15685*, 2023.

Weijie Lyu, Sheng-Jun Huang, and Xuan Xia. Efficient code llm training via distribution-consistent and diversity-aware data selection. *arXiv preprint arXiv:2507.02378*, 2025.

Jinlong Pang, Jiaheng Wei, Ankit Parag Shah, Zhaowei Zhu, Yaxuan Wang, Chen Qian, Yang Liu, Yujia Bao, and Wei Wei. Improving data efficiency via curating llm-driven rating systems. *arXiv preprint arXiv:2410.10877*, 2024.

Jinlong Pang, Na Di, Zhaowei Zhu, Jiaheng Wei, Hao Cheng, Chen Qian, and Yang Liu. Token cleaning: Fine-grained data selection for llm supervised fine-tuning. In *International Conference on Machine Learning*, pp. 47837–47858. PMLR, 2025.

Xiaohan Qin, Xiaoxing Wang, Ning Liao, Cancheng Zhang, Xiangdong Zhang, Mingquan Feng, Jingzhi Wang, and Junchi Yan. sstoken: Self-modulated and semantic-aware token selection for llm fine-tuning. *arXiv preprint arXiv:2510.18250*, 2025.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, et al. Seed-coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524*, 2025.

Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. Code less, align more: Efficient llm fine-tuning for code generation with data pruning. *arXiv preprint arXiv:2407.05040*, 2024.

Bohao Wang, Feng Liu, Jiawei Chen, Xingyu Lou, Changwang Zhang, Jun Wang, Yuegang Sun, Yan Feng, Chun Chen, and Can Wang. Msl: Not all tokens are what you need for tuning llm as a recommender. In *Proceedings of the 48th international ACM SIGIR conference on research and development in information retrieval*, pp. 1912–1922, 2025a.

Jingtan Wang, Xiaoqiang Lin, Rui Qiao, Pang Wei Koh, Chuan-Sheng Foo, and Bryan Kian Hsiang Low. Nice data selection for instruction tuning in llms with non-differentiable evaluation metric. In *Forty-second International Conference on Machine Learning*, 2025b.

Yejie Wang, Keqing He, Dayuan Fu, Zhuoma GongQue, Heyang Xu, Yanxu Chen, Zhexu Wang, Yujia Fu, Guanting Dong, Muxi Diao, et al. How do your code llms perform? empowering code instruction tuning with really good data. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 14027–14043, 2024.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and Lingming Zhang. Selfcodealign: Self-alignment for code generation. *Advances in Neural Information Processing Systems*, 37:62787–62874, 2024a.

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

Yang Wu, Huayi Zhang, Yizheng Jiao, Lin Ma, Xiaozhong Liu, Jinhong Yu, Dongyu Zhang, Dezhi Yu, and Wei Xu. Rose: A reward-oriented data selection framework for llm task-specific instruction tuning. *arXiv preprint arXiv:2412.00631*, 2024.

Mengzhou Xia, Sadhika Malladi, Suchin Gururangan, Sanjeev Arora, and Danqi Chen. Less: Selecting influential data for targeted instruction tuning. *arXiv preprint arXiv:2402.04333*, 2024.

Yuming Yang, Yang Nan, Junjie Ye, Shihan Dou, Xiao Wang, Shuo Li, Huijie Lv, Tao Gui, Qi Zhang, and Xuan-Jing Huang. Measuring data diversity for instruction tuning: A systematic analysis and a reliable metric. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 18530–18549, 2025.

Jiapeng Yu, Yuqian Wu, Yajing Zhan, Wenhao Guo, Zhou Xu, and Raymond Lee. Co-learning: code learning for multi-agent reinforcement collaborative framework with conversational natural language interfaces. *Frontiers in Artificial Intelligence*, 8:1431003, 2025.

Jia Zhang, Chen-Xi Zhang, Yao Liu, Yi-Xuan Jin, Xiao-Wen Yang, Bo Zheng, Yi Liu, and Lan-Zhe Guo. D3: Diversity, difficulty, and dependability-aware data selection for sample-efficient llm instruction tuning. *arXiv preprint arXiv:2503.11441*, 2025.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations*, volume 2025, pp. 66602–66656, 2025.

A Proofs for the Motivating Analysis

Throughout, let $y \in \mathrm{Lang}(G)$ be a reference program with $\mathrm{Pass}_{\mathcal{T}}(y) = 1$, let $M(u) = [s, e]$ be a parser-guided coding-item span, and let $\hat{y}_{M(u)}$ be the $|M(u)|$-token greedy continuation decoded from $y_{<s}$.

Proof of Proposition 1. If $\hat{y}_{M(u)} = y_{M(u)}$, then $\tilde{y}^{(u)} = y$, so $F_{G,\mathcal{T}}(\tilde{y}^{(u)}) = 0$. Since $F_{G,\mathcal{T}}$ is binary, this proves the first inequality.

For the second inequality, the result is immediate when $\hat{y}_{M(u)} = y_{M(u)}$. Otherwise, let

$$t^\star = \min\{t \in M(u) : \hat{y}_t \neq y_t\}$$

be the first mismatched position. Local decoding starts from $y_{<s}$, and all locally generated tokens before $t^\star$ match their reference tokens; hence the decoder prefix at $t^\star$ is exactly $y_{<t^\star}$. Because greedy decoding chooses an incorrect token $\hat{y}_{t^\star}$,

$$p_\theta(\hat{y}_{t^\star} \mid x, y_{<t^\star}) \geq p_\theta(y_{t^\star} \mid x, y_{<t^\star}).$$

The probabilities of these two distinct tokens sum to at most one, so $p_\theta(y_{t^\star} \mid x, y_{<t^\star}) \leq 1/2$ and therefore $\ell_{t^\star}(\theta) \geq \log 2$. It follows that

$$\mathbf{1}[\hat{y}_{M(u)} \neq y_{M(u)}] \leq \frac{\sum_{t \in M(u)} \ell_t(\theta)}{\log 2}.$$

Combining this inequality with the trivial upper bound of one proves Eq. 2. □

B Details Algorithm of Coding Item Construction

Algorithm 1: CodeBlock Coding-Item Construction with Local Closure

Require: Response y_i, response-token indices $\mathcal{I}_i$ with character offsets, code regions $\mathcal{F}_i$, protected syntax set Ω_{P}
Ensure: Coding items $\mathcal{U}_i = \{u_{i,k}\}_{k=1}^{K_i}$, where each item $u = (C(u), M(u))$

$\mathcal{U}_i \leftarrow \emptyset$
for each code region $f \in \mathcal{F}_i$ **do**
  $z \leftarrow \textsc{LabelCodeTokens}(f, \mathcal{I}_i, \Omega_{\mathrm{P}})$ ▷ $z_r \in \{\mathrm{C}, \mathrm{P}, \mathrm{O}\}$
  $C_{\mathrm{cur}} \leftarrow \emptyset$, $d \leftarrow 0$, $a \leftarrow$ first token index in f
  for each token index r in f from left to right **do**
    if $z_r = \mathrm{C}$ **then**
      $C_{\mathrm{cur}} \leftarrow C_{\mathrm{cur}} \cup \{r\}$
    else if $C_{\mathrm{cur}} \neq \emptyset$ **then**
      if $\textsc{Boundary}(t_r, d)$ or $z_r = \mathrm{O}$ **then**
        $b \leftarrow r$ if $z_r = \mathrm{P}$, else $b \leftarrow r - 1$
        $M \leftarrow \textsc{LocalClosure}(C_{\mathrm{cur}}, [a, b], z)$
        $\mathcal{U}_i \leftarrow \mathcal{U}_i \cup \{(C_{\mathrm{cur}}, M)\}$
        $C_{\mathrm{cur}} \leftarrow \emptyset$
      end if
    end if
    if $\textsc{Boundary}(t_r, d)$ or $z_r = \mathrm{O}$ **then**
      $a \leftarrow r + 1$
    end if
    $d \leftarrow \textsc{UpdateDepth}(t_r, d)$
  end for
  if $C_{\mathrm{cur}} \neq \emptyset$ **then**
    $b \leftarrow$ last token index in f
    $M \leftarrow \textsc{LocalClosure}(C_{\mathrm{cur}}, [a, b], z)$
    $\mathcal{U}_i \leftarrow \mathcal{U}_i \cup \{(C_{\mathrm{cur}}, M)\}$
  end if
end for
return $\mathcal{U}_i$

Table 3: Default hyperparameters for token- and item-level supervision selection in CODEBLOCK.

Hyperparameter	Description	Default
q	Smoothing parameter in the GCE score.	0.7
ρ_{NL}	Retention ratio for natural-language tokens.	0.6
ρ_{code}	Code supervision budget ratio for selected coding items.	0.33
η	CODEBLOCK gate ratio; only the top-η GCE-scored items receive data-flow bonuses.	30%
λ	Overall strength of the data-flow bonus.	0.10
α_r	Weight of the reach signal.	1.00
α_b	Weight of the bridge signal.	0.50

C EXPERIMENTAL DETAILS

C.1 SAMPLE-LEVEL SELECTION SETTINGS

We perform sample-level selection from the 300K OpenCodeInstruct (OCI) source pool before applying item-level sparse supervision. Instead of re-querying an external judge model, we reuse the LLM-judge scores provided by OCI and follow its scoring protocol. Each sample is evaluated along three dimensions: requirement conformance, logical correctness, and edge-case consideration. We compute the mean judge score over these dimensions and combine it with a lightweight test-based auxiliary score:

$$s_{\mathrm{qual}}^{\mathrm{raw}} = s_{\mathrm{judge}} + 0.2 \cdot s_{\mathrm{test}}, \tag{7}$$

where s_{judge} denotes the average LLM-judge score and s_{test} denotes the average available test score. The test score is used only as a small auxiliary signal for tie-breaking and calibration, rather than as the dominant selection criterion. We then apply robust min–max normalization to obtain the normalized quality score $s_{\mathrm{qual}}^{\mathrm{norm}} \in [0, 1]$.

To avoid selecting only common and easy high-quality examples, we further compute a lightweight long-tail score. The long-tail score combines tag-level rarity and bucket-level rarity:

$$s_{\mathrm{tail}} = 0.7 \cdot s_{\mathrm{tag}} + 0.3 \cdot s_{\mathrm{bucket}}. \tag{8}$$

Here, s_{tag} captures the presence of relatively rare program structures or API-related tags, such as imports, classes, multiple functions, recursion, exception handling, graph-related operations, date/time operations, and asynchronous programming. The bucket score s_{bucket} measures rarity with respect to prompt length, code length, and coarse complexity buckets.

The final sample-level ranking score is computed as:

$$s_{\mathrm{sample}} = 0.7 \cdot s_{\mathrm{qual}}^{\mathrm{norm}} + 0.3 \cdot s_{\mathrm{tail}}. \tag{9}$$

C.2 TOKEN AND ITEM SELECTION SETTINGS

Table 3 summarizes the default hyperparameters used in the token- and item-level selection stage of CODEBLOCK. Unless otherwise specified, all experiments use the same configuration.

C.3 FINE-TUNING SETTINGS

We report the main fine-tuning settings in Table 4. All methods, including full-token SFT, CODEBLOCK, and every selection baseline, are fine-tuned for exactly one epoch over the training set produced by the corresponding selection strategy under an identical optimization setup. All experiments use an effective batch size of 8 and are conducted on one NVIDIA A800 GPU with bf16 precision.

For sparse-supervision variants, the input sequence is kept unchanged: unselected response tokens remain available as autoregressive context, but their labels are set to the ignore index and therefore do not contribute to the cross-entropy loss. Within the selected training examples, CODEBLOCK differs from full-token SFT only in the supervision mask, while using the same causal language modeling objective.

Table 4: Fine-tuning settings used for the main experiments.

Setting	Value
Training budget	1 epoch for all methods
Learning rate schedule	LR = 3×10^{-5}, scheduler = cosine
Effective batch size	8 = per-device batch size 2 $\times$ grad. accumulation 4 $\times$ GPUs 1
Maximum sequence length	2048 tokens
Optimization setup	AdamW, weight decay 0.0, warmup ratio 0.05, bf16

Table 5: Observed wall-clock runtime in minutes on the Qwen2.5-Coder-1.5B-Instruct setting. We include method-specific preprocessing and SFT training, and exclude evaluation.

Method	**Preprocess**	**Train**	**Total**	**Total / Full**
Full Tokens (300K)	8.9	257.7	266.6	100.0%
Random Selection	3.2	37.6	40.8	15.3%
CODEBLOCK	32.1	36.8	68.9	25.8%
Token Cleaning	32.6	12.1	44.7	16.8%
DS^2	29.0	38.6	67.6	25.4%
CLAM	16.5	35.5	52.0	19.5%

C.4 EVALUATION SETTINGS

For HumanEval and MBPP, we use EvalPlus and report sanitized pass@1 on HumanEval, HumanEval+, MBPP, and MBPP+. For BigCodeBench, we evaluate instruction-tuned models on the instruct split and base models on the complete split, reporting pass@1 on the hard and full subsets. All reported pass@1 scores use greedy decoding with temperature 0.0 and one generated sample per task.

D RUNTIME ANALYSIS

We additionally report observed wall-clock runtime on the Qwen2.5-Coder-1.5B-Instruct main setting. For each method, we count method-specific preprocessing and SFT training time, while excluding evaluation time. Shared dataset construction costs, such as LLM-judge scoring for constructing the common candidate pool, are not attributed to any individual method.

As shown in Table 5, Full Tokens requires 266.6 minutes in total, with most of the cost coming from training on the full 300K dataset. In contrast, CODEBLOCK requires 68.9 minutes, or 25.8% of the Full Tokens runtime. Although CODEBLOCK introduces additional preprocessing for token scoring and structured item construction, its reduced training workload substantially lowers the overall runtime. Its total runtime is comparable to DS^2 and remains far below full-token training, while being higher than Random Selection, Token Cleaning, and CLAM. Overall, the structure-aware preprocessing introduces moderate overhead while preserving a substantial runtime advantage over dense full-data supervision.

E ADDITIONAL EXPERIMENT RESULTS

E.1 MORE EXPERIMENTAL RESULTS

We report additional results on Qwen2.5-Coder-3B-Instruct and Qwen2.5-Coder-7B-Instruct to further examine the generality of CODEBLOCK. All methods use the same data pool, training protocol, and evaluation setting as the main experiments. As shown in Table 6, CODEBLOCK consistently achieves strong average performance with only about 6.9% supervised response tokens, further supporting the effectiveness of structure-aware sparse supervision across different model scales.

E.2 MORE ABLATION STUDY

Table 6: Additional comparison of different baselines on various benchmarks. We highlight the best result in **boldface** and the second-best with underline.

Method	HumanEval	HumanEval+	MBPP	MBPP+	BCB-hard	BCB-full	Eff. Tokens (%)	Average
Base model: Qwen2.5-Coder-3B-Instruct								
BASE	84.2	**79.3**	73.8	62.7	13.5	34.8	–	58.0
FULL TOKENS	82.3	75.0	81.2	**69.6**	14.9	37.2	100.0	60.0
RANDOM SELECTION	83.5	75.6	78.3	65.9	**16.9**	42.3	6.9	60.4
DS^2	**86.0**	78.1	79.1	66.7	**16.9**	**43.3**	17.0	**61.7**
TOKEN CLEANING	82.9	76.8	78.6	64.8	12.2	36.8	**5.1**	58.7
CLAM	84.8	78.1	**81.5**	66.7	14.2	41.8	13.2	61.2
CODEBLOCK	83.5	77.4	**81.5**	68.0	16.2	42.1	6.9	61.5
Base model: Qwen2.5-Coder-7B-Instruct								
BASE	87.2	83.1	83.5	71.6	19.4	39.7	–	64.1
FULL TOKENS	87.8	82.9	85.2	71.4	18.9	39.9	100.0	64.4
RANDOM SELECTION	87.2	83.5	**87.8**	75.1	18.9	41.3	6.9	65.6
DS^2	86.0	81.1	87.3	72.5	**24.3**	48.8	17.0	66.7
TOKEN CLEANING	81.7	76.8	84.9	73.3	18.2	40.3	**5.1**	62.5
CLAM	89.0	**84.2**	86.5	75.1	12.8	39.4	13.2	64.5
CODEBLOCK	**90.2**	83.5	86.8	**75.4**	20.3	**49.2**	6.9	**67.6**

We further study the effect of the data-flow bonus strength λ in the CODEBLOCK priority function. This hyperparameter controls how strongly the reach and bridge signals influence the final ranking of candidate coding items.

Figure 9 reports the results under different values of λ, while keeping all other hyperparameters fixed. The results show that incorporating the data-flow bonus is generally beneficial, with $\lambda = 0.10$ achieving the best average performance of 54.2. Compared with the $\lambda = 0$ baseline of 53.5, this improvement suggests that reach and bridge signals provide complementary structural information beyond token-level GCE utility.

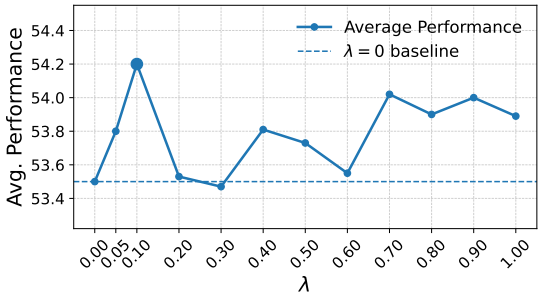


Figure 9: Sensitivity analysis of λ on the average pass@1 score across six code-generation benchmarks.

However, the trend is not monotonic, and further increasing λ does not consistently improve performance. While most non-zero settings outperform the $\lambda = 0$ baseline, overly emphasizing the structural bonus can reduce the benefit. One possible reason is that the data-flow graph is constructed through static code analysis and therefore provides only an approximate characterization of dependency importance, particularly for incomplete snippets, library calls, and implicit dependencies. Thus, data-flow information is more effective as a lightweight reranking signal that complements GCE utility rather than dominating the original ranking. Overall, these results support the effectiveness of data-flow-aware reranking while highlighting the importance of properly balancing structural and token-level utility signals. We therefore set $\lambda = 0.10$ as the default value in our experiments.