Are you Synthesizing or Recalling?
Evaluating LLMs on Algorithmic Code Retrieval

Anonymous ACL submission

Abstract

Large language models (LLMs) have demonstrated strong performance in code generation, where success depends on both recalling relevant algorithmic knowledge and reasoning about how to apply it. However, existing LLM pipelines are opaque, with no explicit separation between these two components. We argue that for well-known algorithms whose canonical implementations are widely accessible in pretraining corpora, code generation is better measured as *parametric code retrieval*: reproducing a named algorithm from internalized knowledge rather than synthesizing a novel one. We introduce AlgoReval, a benchmark of 599 problems spanning classical 77 algorithms across 14 domains, 7 programming languages, and 4 graph-input representations to evaluate this capability in isolation, and assess 15 models (7B–34B parameters) in a zero-shot setting. We find substantial variation in retrieval accuracy across languages and input representations, even for widely documented algorithms and show that prompt augmentation with retrieved code snippets or structured algorithmic hints improve accuracy on complex algorithms, while SFT achieves broader language gains and GRPO achieves larger per-language gains on specific languages. Together, our results establish parametric code retrieval as a distinct, measurable capability and caution against deploying AI-generated algorithmic code without systematic validation.

1 Introduction

When an LLM writes Dijkstra's algorithm, is it *synthesizing*, *retrieving*, or *reciting*? *Synthesizing* means reasoning from the problem statement to a solution, deriving logic the model has not seen in the exact form during pre-training (Chen et al., 2021; Austin et al., 2021b). *Retrieving* means reconstructing a correct implementation from internalized parametric knowledge. The model has encoded the algorithm's structure, invariants, and control flow in its weights and can express it correctly even in forms it has not seen verbatim (Roberts et al., 2020; Gekhman et al., 2026). *Reciting* means replaying a near-verbatim token sequence cached from a specific training example: correct output is a side-effect of having stored that string, not of understanding what it computes (Carlini et al., 2023; Huang et al., 2024). Synthesis, retrieval, and recitation are distinct capabilities with different failure modes and different implications for reliability: synthesis may fail on novel problems; retrieval degrades when surface form shifts away from the training distribution; recitation breaks the moment the expected transcription is absent. Prior work conflates all three under the label of *code generation*. We focus on the "retrieval" aspect. The distinction has real stakes: an estimated 72% of developers now use AI coding tools on a daily basis, and AI currently generates 42% of all committed code.[1] Yet, 66% report their top frustration is code that is almost right, but not quite, which precisely describes retrieval failure: the model knows the algorithm but gets a detail wrong (Zhu et al., 2025).

In agentic coding pipelines, where models plan, generate, and commit code without per-step human review, a retrieval failure in a graph algorithm may not surface until a production incident exposes the wrong output being computed silently (Sapkota et al., 2025; Cemri et al., 2025). Knowing where and how retrieval breaks is the minimum information needed to decide when AI-generated algorithmic code can be trusted without verification.

Most code generation research frames the task as *synthesis*: given a problem statement, construct a solution. This is appropriate for novel problems, but for well-known algorithms whose canonical implementations appear across thousands of open-source repositories and almost certainly in the pre-

[1] https://www.sonarsource.com/state-of-code-developer-survey-report.pdf

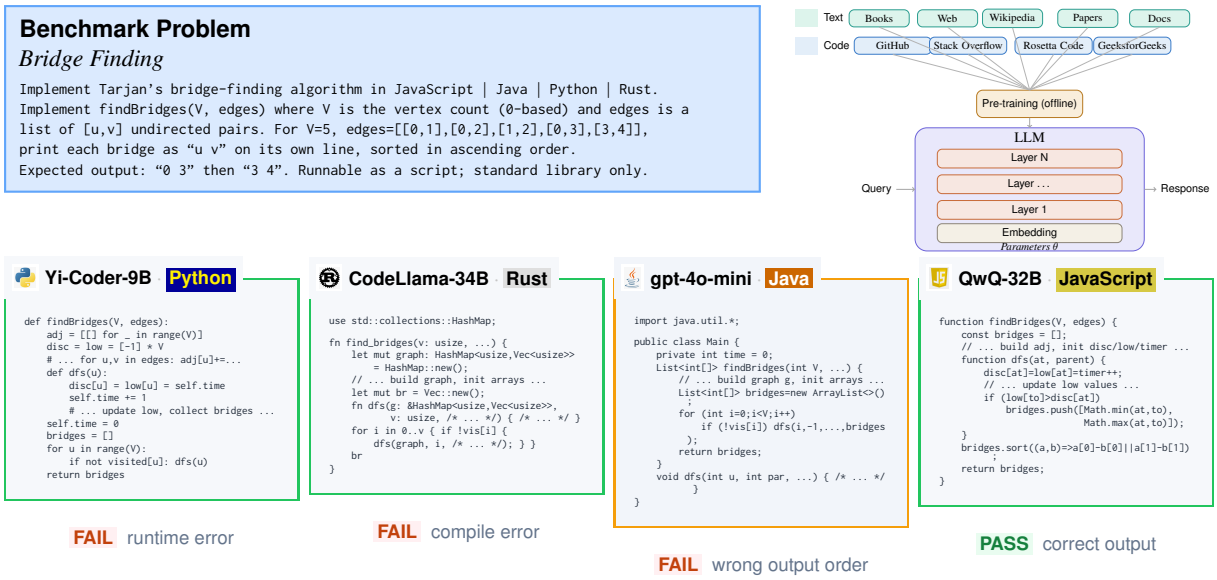

Figure 1: **Parametric code retrieval evaluation pipeline.** LLMs generate solutions zero-shot, drawing solely on algorithmic knowledge encoded in their pretraining weights. The four panels show four models (three open-weight, one closed API) on the same Bridge Finding problem, each in a different language: Yi-Coder-9B (Python) uses `self.time` in a plain function with no class, causing a *runtime error*; CodeLlama-34B (Rust) passes `graph` by value where a reference is required, producing a *compile error*; gpt-4o-mini (Java, closed API) finds the correct bridges but omits the sort step, yielding *wrong output order*; QwQ-32B (JavaScript) normalises, sorts, and passes. We report pass rate by domain, language, and representation, and analyse output diversity as a proxy for whether models retrieve a shared canonical implementation or synthesize independently. The border color indicates open-source (green) vs. closed-source (amber) models.

training corpora of every model we evaluate. The model need not invent anything; it must *retrieve*. Conflating the two obscures a distinct failure mode: a model may demonstrate impressive synthesis on novel tasks while quietly failing to reproduce even the most widely documented algorithms, or vice-versa (Riddell et al., 2024; Matton et al., 2024). We isolate this as *parametric code retrieval*: reproducing a correct implementation of a named algorithm in a specified language from parametric memory alone. Code constitutes roughly 13% of the Dolma corpus (411 B of 3.1 T tokens) (Kocetkov et al., 2022; Soldaini et al., 2024), with JavaScript (486 GB), Java (271 GB), and C (222 GB) accounting for over half by volume, which are precisely the language families our benchmark targets.

Contributions. We introduce AlgoReval, a benchmark of 599 problems spanning 77 algorithms, 7 programming languages, and 4 graph-input representations, and evaluate fifteen models zero-shot. We then ask whether retrieval failures can be remediated: first via prompt augmentation with retrieved code snippets or structured algorithmic hints, and then via GRPO fine-tuning with execution-based rewards compared against a supervised fine-tuning baseline. Our main findings are threefold. First, retrieval accuracy varies widely across models, and the variance is governed by surface form (programming language and input representation) and not model size or code specialisation. The dominant failure mode in Python and JavaScript is wrong output: runnable code that silently returns the wrong answer. Second, prompt augmentation does not reliably improve retrieval in aggregate, and actively hurts when parametric knowledge is already present; executable snippets are more effective than abstract hints only on problems where the model's parametric trace is already thin. Third, GRPO fine-tuning with execution-based rewards converges faster than SFT and closes language gaps that binary supervision cannot address, particularly in Rust where partial-credit signals from compile failures carry information that correct/incorrect labels discard.

2 Task Formulation

We define *parametric code retrieval* as the task of recovering a correct, executable implementation of a named algorithm from an LLM's parametric memory, in a specified programming language and input representation, without access to external doc-

uments or in-context examples.

Problem instances. Let $\mathcal{A}$ denote a set of algorithm names, $\mathcal{L} = \{\text{Python, C++, Java, C, JavaScript, Rust, Go}\}$ a set of target languages, and $\mathcal{R}_a$ the set of input-representation formats applicable to algorithm a (e.g., edge list, adjacency list, adjacency matrix for graph algorithms; a single canonical format for all others). Each problem instance $p \in \mathcal{P}$ is a tuple

$$p = (a,\ \ell,\ r,\ \sigma_p,\ t_p,\ y_p),$$

where σ_p is the required function or class signature, t_p is a fixed test input, and y_p is the expected standard output produced by executing a reference implementation on t_p. Each instance is additionally tagged with a domain label $d \in \mathcal{D}$ used for stratified reporting but not part of the prompt or scoring. Multiple instances may share the same (a, ℓ) while differing only in r; these probe representational robustness of the same underlying algorithm without changing its identity.

Prompt structure. The prompt x_p names the algorithm a, specifies the target language ℓ and input representation r, states the required signature σ_p, and embeds the concrete test input t_p. No reference implementation, pseudocode, or chain-of-thought scaffolding is provided. This zero-shot constraint ensures that any correct solution must originate from the model's parametric memory rather than from in-context retrieval.

Scoring. A model m generates code $c_m(p)$ from x_p. Executing $c_m(p)$ in a sandboxed subprocess yields output $\hat{y}_m(p)$, and retrieval succeeds when $\hat{y}_m(p) = y_p$ (stdout match after whitespace normalisation). We use the pass rate (Chen et al., 2021) of model m over subset $S \subseteq \mathcal{P}$ as

$$\mathrm{Pass}(m,\ S)\ =\ \frac{1}{|S|}\sum\nolimits_{p \in S} \mathbf{1}[\hat{y}_m(p) = y_p]\,,$$

and report $\mathrm{Pass}(m, \mathcal{P})$ overall, stratified by language ℓ and domain d across all instances, and by representation r within the graph-algorithm subset $\mathcal{P}_{\text{graph}} \subset \mathcal{P}$ where multiple representations exist. Failures fall into four categories: *static failure* (code fails to parse or compile before execution), *runtime error* (code raises an exception during execution), *timeout* (execution exceeds the time budget), and *wrong output* ($\hat{y}_m(p) \neq y_p$ despite clean execution). Wrong-output failures are the most diagnostic: they show the model produced algorithmically plausible but incorrect logic.

3 Dataset Construction

We build the benchmark with 107 Python problems drawn from RosettaCode[2] and GeeksforGeeks[3], covering 77 distinct algorithms across 14 domains: graph algorithms, sorting, dynamic programming, string algorithms, number theory, image processing, data structures, computational geometry, numerical methods, compression, cryptography, backtracking, and parsing. Some algorithms contribute to more than one problem, e.g., a weighted and an unweighted variant, accounting for the gap between 107 problems and 77 algorithms. We select algorithms that are extensively documented across open-source repositories and therefore almost certainly present in any large pretraining corpus. This is a deliberate design choice: we are not testing generalisation to unseen problems, but the fidelity of recall for algorithms a model has already been exposed to. Ground truth for each problem is established by executing a reference implementation on the fixed test input t_p and recording its standard output as y_p; a model passes if and only if its generated code reproduces y_p exactly after whitespace normalisation. Each algorithm is assigned to exactly one primary domain based on its canonical problem family, not all techniques it may employ: an expression parser is labeled *parsing* rather than dynamic programming, even if it uses memoisation internally.

Each Python problem is adapted to six additional languages (C++, Java, C, JavaScript, Rust, and Go) by applying three transformations: (1) replacing the target language name throughout the prompt, (2) substituting the embedded Python test block with an explicit expected-output specification, and (3) rewriting Python-specific entry-point idioms to language-appropriate forms (e.g., `int main()` for C/C++, `fn main()` for Rust). This yields a multilingual benchmark in which the same algorithmic task is posed identically across all 7 languages, isolating language-level retrieval differences from differences in problem framing.

Parametric retrieval is interface-sensitive: a model that correctly implements Dijkstra's algorithm given an edge list may fail when the same graph is presented as an adjacency matrix, even though the underlying algorithm is unchanged. To probe this, we augment the 24 graph-algorithm problems with variants across four input represen-

[2] https://rosettacode.org/wiki/Rosetta_Code
[3] https://www.geeksforgeeks.org

Domain	Problem Types	Entries
Graph algorithms	24	228
Image processing	10	70
Data structures	5	35
Computational geometry	4	28
Sorting algorithms	4	28
Dynamic programming	4	28
Numerical methods	4	28
Backtracking	3	21
Cryptography	3	21
String algorithms	3	21
Number theory	3	21
Parsing	2	14
Compression	2	14
Other	6	42
Total	**77**	**599**

Table 1: **Dataset breakdown by domain.** Each problem type is instantiated across 7 languages. The graph-algorithm problems additionally include up to 4 input-representation variants.

tations, yielding 228 graph-algorithm entries in total:

- **Edge list** (`edge_list`): the default; `solve(n, edges, ...)` where `edges` is a list of `(u,v)` or `(u,v,w)` tuples.
- **Adjacency list** (`adj_list`): `solve(adj, ...)` where `adj[u]` is a list of neighbours or (neighbour, weight) pairs; vertex count is implicit as `len(adj)`.
- **Adjacency matrix** (`adj_matrix`): `solve(M, ...)` where `M[u][v]` encodes edge weight (∞ for absent edges; $1/0$ for unweighted graphs).
- **Incidence matrix** (`incidence_matrix`): an $(n \times m)$ vertex-by-edge matrix used for selected bipartite-matching problems.

Representation coverage varies by algorithm: adjacency list is available for 20 of the 24 graph algorithms, adjacency matrix for 10, and incidence matrix for 2. The remaining 371 non-graph entries use edge list as their sole representation. The final benchmark contains 599 entries spanning 7 programming languages and 4 input representations. Table 1 provides a full breakdown by domain and language.

4 Related Work

4.1 Parametric Memory in LLMs.

Pretraining encodes factual and procedural knowledge directly into model weights as *parametric memory* (Petroni et al., 2019), which can be queried at inference time without external retrieval. Probing studies on factual QA show that recall is reliable when inference-time surface forms closely match training data, but degrades sharply under paraphrase or format change (Roberts et al., 2020; Gekhman et al., 2026). We study the same degradation in code: the surface shift is a change in programming language or input representation rather than a change in natural-language phrasing, and the failure mode is wrong output rather than a factual error.

4.2 Code Generation Benchmarks

The dominant paradigm in code generation research treats the task as synthesis: given a natural-language specification, produce a correct program. HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021a) operationalise this with short functional puzzles. APPS (Hendrycks et al., 2021) scales difficulty to competitive programming. LiveCodeBench (Jain et al., 2024) reduces contamination by time-gating problems after model training cutoffs. All of these treat training-data overlap as a confound to eliminate. We take the opposite stance: overlap is the *precondition* for our task, and the question worth asking is not whether a model can produce code but whether it faithfully reproduces correct implementations of algorithms it has already encountered. Crucially, because existing benchmarks do not control for training exposure, a model's pass rate on a well-known algorithm conflates two fundamentally different capabilities: *retrieval*, i.e., recalling a correct implementation from parametric memory, and *understanding*, i.e., deriving a solution through reasoning (Maveli et al., 2026).

4.3 Retrieval-augmented generation (RAG)

RAG-based code generation assumes a parametric knowledge deficit and supplies retrieved context to fill it (Wang et al., 2025). The retrieved examples add retrieval noise, misaligned context, and surface-level code patterns that compete with, rather than complement, what the model already encodes parametrically. Models perform significantly better on the subset of benchmarks where similar solutions were seen during training, and are more likely to emit memorized training data when they have seen it multiple times (Riddell et al., 2024). We therefore focus on parametric code recall: the model's capacity to retrieve correct algorithmic solutions from weights alone, without retrieval augmentation.

5 Experimental Setup

5.1 Models

Programming language distribution in LLM pretraining data varies dramatically by model type and is rarely disclosed at the per-language level. General-purpose models allocate a modest fraction to code: LLaMA-1 (Touvron et al., 2023) reserved only 4.5% of its 1.4T tokens for GitHub content, while LLaMA-3 (Grattafiori et al., 2024) scaled this to approximately 17% of 15T+ tokens. Code-centric models invest far more: DeepSeek-Coder (Guo et al., 2024) allocates 87% of its 2T tokens to code across 87 languages, and StarCoder2 (Lozhkov et al., 2024) trains almost entirely on The Stack v2 (619 languages, ~4T tokens). This variation matters for our benchmark: a model's pretraining code budget directly shapes which languages it has had sufficient exposure to for reliable parametric retrieval.

We evaluate 15 models spanning three axes of variation: (1) *specialisation*: general-purpose instruction-tuned models versus code-specialised ones; (2) *scale*: 7B to 34B parameters, and (3) *inference regime*: standard decoding versus extended chain-of-thought reasoning. Full model identifiers, parameter counts, and licenses are listed in Appendix B.

5.2 Evaluation

Execution harness. Each model-generated program is executed in an isolated subprocess with a 30-second wall-clock timeout. Interpreted languages (Python, JavaScript) are run directly, whereas compiled languages (C, C++, Java, Rust, Go) go through a mandatory compilation step first. A compilation failure is recorded immediately as a static failure without attempting execution. Standard output is compared against the ground-truth string after whitespace normalisation; a submission passes if and only if the strings match exactly. Failures are categorised into four mutually exclusive types: *static failure* (parse or compile error), *runtime error* (uncaught exception during execution), *timeout* (execution exceeds 30 s), and *wrong output* (clean execution but incorrect stdout).

Metrics. Our primary metric is **pass rate**: the fraction of problem instances on which a model's generated code passes, reported overall and stratified by domain, language ℓ, and input representation r. Pass rate captures correctness but not consistency: two models can both pass a problem while producing structurally unrelated implementations. To characterise solution diversity, we follow the AlgoSim evaluation framework (Lee et al., 2025): model predictions are pooled, grouped by problem label, and evaluated for pairwise lexical similarity across every pair of models that each produced code for the same problem.

With the benchmark and evaluation harness in place, we organise our experiments around three questions. The first establishes where and how retrieval fails: without a clear failure map, any proposed remedy is unmotivated. The second asks whether those failures can be patched at inference time through prompt augmentation, which requires no model changes but assumes that injecting external context into a prompt where parametric knowledge already exists is net-positive. The third asks whether the gaps can instead be closed at the weight level through execution-based fine-tuning, separating the contribution of reward signal (GRPO) from the contribution of training on correct solutions alone (SFT). In this paper, we mainly investigate the following research questions through experiments.

RQ1. How reliably do LLMs retrieve algorithmic code from parametric knowledge across languages and input representations?

RQ2. Does augmenting LLM prompts with retrieved code snippets or structured algorithmic hints improve parametric code retrieval accuracy, and which source better compensates for failures on complex and esoteric algorithms?

RQ3. Does GRPO fine-tuning with execution-based rewards reduce representation and language accuracy gaps relative to SFT on algorithmic code retrieval?

6 Analysis

We report results for each research question in turn, moving from baseline measurement of retrieval fidelity (RQ1) through inference-time intervention via prompt augmentation (RQ2) to weight-level adaptation via fine-tuning (RQ3).

6.1 RQ1: Parametric Code Retrieval Across Languages and Input Representations

We evaluate all models in a zero-shot setting with no in-context examples, no chain-of-thought scaffolding, and no hints about the solution approach. The prompt template and generation hyperparameters are detailed in Appendices F and C.

The aggregate pass rate across all 15 models is 41.2%, with substantial variation across languages and input representations but little correlation with model size or code specialisation (Table 2). gpt-oss-20B achieves the highest overall pass rate despite having only 20B active parameters, outperforming larger models, including QwQ-32B (32B) and GPT-4o-mini. Figure 2 breaks down pass rates by model and programming language: general-purpose models lead in most languages, while code-specialised models exhibit more uneven language coverage. The dominant failure mode in Python and JavaScript is wrong output (runnable code that silently computes the incorrect answer), whereas compiled languages show a higher proportion of static failures (Figure 8). Representation sensitivity is pronounced (Figure 3): for the same model on the same algorithm, pass rates drop by over 10 percentage points when the input encoding shifts from edge list to adjacency matrix, consistent with the interface-sensitivity argument in Section 1.

Category	Model	Weights	Params	Pass@1 (%)
Code	Codestral-22B	Open	22	**52.6**
	DeepSeek-Coder-33B	Open	33	46.4
	Yi-Coder-9B	Open	9	46.2
	CodeGemma-7B	Open	7	25.9
	StarCoder2-15B	Open	15	13.7
	CodeLlama-34B	Open	34	12.9
General	gpt-oss-20B	Open	20	**88.0**
	GPT-4.1-mini	Closed	–	80.8
	GPT-4o-mini	Closed	–	66.3
	Phi-4	Open	14	59.4
	Qwen2.5-7B	Open	7	37.4
	Llama-3.1-8B	Open	8	31.4
Reasoning	QwQ-32B	Open	32	**69.6**
	DeepSeek-R1-Distill-Qwen-14B	Open	14	39.6
	DeepSeek-R1-Distill-Llama-8B	Open	8	12.9

Table 2: **Zero-shot Pass@1 (%) grouped by category and ranked within each group by Pass@1**. Closed API models (GPT-4.1-mini, GPT-4o-mini) have undisclosed sizes.

Solution diversity. To test whether correct solutions reflect recitation of stored tokens or genuine reconstruction of algorithmic knowledge, we compute pairwise lexical similarity across all model predictions (2,625 pairs over 25 problems). The mean SequenceMatcher similarity is 0.21, and models produce syntactically distinct implementations

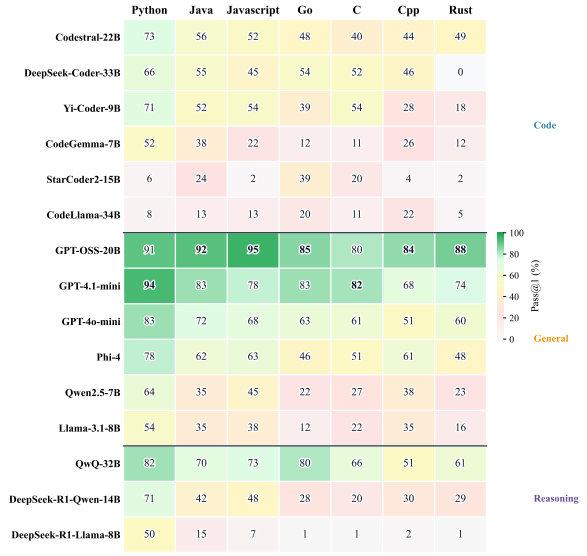

Figure 2: **Pass@1 heatmap by model and programming language.** Rows are grouped by model category (Code, General, Reasoning) and ordered by mean pass rate within each group.

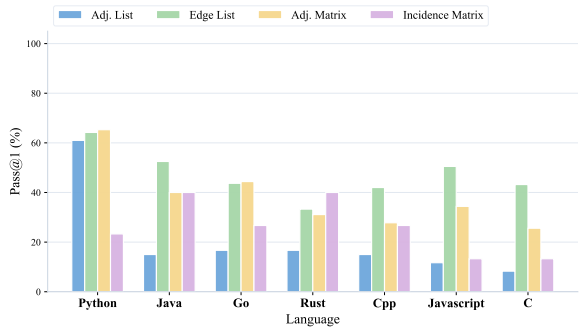

Figure 3: **Pass@1 by programming language, broken down by input representation and aggregated across all model predictions.** Each group compares edge-list, adjacency-list, adjacency-matrix, and incidence-matrix inputs within a language.

even when solving the same problem correctly (EM=0 across all pairs). For semantic-level analysis we apply AlgoSim (Lee et al., 2025), which groups pairs by whether they share the same underlying algorithmic approach. Across 2,215 pairs spanning 37 algorithms, 80% of pairs share the same underlying approach despite surface variation, while 20% represent structurally distinct implementations of the same algorithm. This combination of high surface diversity, predominantly shared algorithmic structure, and zero verbatim matches is consistent with parametric retrieval: models converge on the same algorithm from memory but express it independently rather than replaying a cached token sequence. Figure 9 further shows that pass rate and

Model	Base	+Hint	Δ(H)	+Snip	Δ(S)	Better
StarCoder2-15B	13.7	12.0	-1.7	8.7	-5.0	H
DeepSeek-R1-Distill-Llama-8B	12.9	11.2	-1.7	12.4	-0.5	S
CodeGemma-7B	25.9	26.0	+0.2	24.5	-1.3	H
CodeLlama-34B	12.9	11.7	-1.2	11.7	-1.2	tie
Llama-3.1-8B	31.4	27.7	-3.7	29.7	-1.7	S
Qwen2.5-7B	37.4	37.6	+0.2	37.1	-0.3	H
Yi-Coder-9B	46.2	46.2	0.0	42.7	-3.5	H
DeepSeek-Coder-33B	46.4	46.4	0.0	47.4	+1.0	S
DeepSeek-R1-Distill-Qwen-14B	39.6	37.6	-2.0	39.9	+0.3	S
Phi-4	59.4	58.9	-0.5	59.6	+0.2	S
Codestral-22B	52.6	50.4	-2.2	52.8	+0.2	S
QwQ-32B	69.6	73.6	+4.0	74.6	+5.0	S
gpt-oss-20B	88.0	89.0	+1.0	90.2	+2.2	S
Macro avg.	41.2	40.6	-0.6	40.9	-0.4	H:4 S:8

Table 3: **Pass rates (%) under three conditions across 13 models on all 599 benchmark examples.** *Base*: zero-shot baseline. *+Hint*: GPT-4o-mini-generated pseudocode. *+Snip*: language-matched GitHub snippet. **Better** indicates which augmentation condition outperforms the other per model.

consensus recall are correlated but not identical: some models produce correct outputs that deviate from the plurality answer, indicating independent retrieval rather than convergence on a single cached response. A breakdown of AlgoSim novel rate per model and language is provided in Figure 7.

6.2 RQ2: Prompt Augmentation via Retrieved Snippets and Structured Hints

When parametric retrieval fails, a natural remedy is to inject external knowledge directly into the prompt. We test two distinct sources of such knowledge and ask whether either one can rescue failing models. The **+Hint** condition prepends language-agnostic pseudocode (up to 20 lines) generated offline by GPT-4o-mini, covering key data structures and asymptotic complexity. The **+Snippet** condition injects a language-matched reference implementation retrieved from GitHub (up to 80 lines, centred on the most relevant region), explicitly excluding RosettaCode and GeeksforGeeks to avoid overlap with our benchmark sources. Both conditions otherwise use identical prompts and generation hyperparameters to the zero-shot baseline. All 599 benchmark problems received a language-matched GitHub snippet. Full setup details and prompt examples appear in Appendices D and G.

Across all problems, neither augmentation source improves aggregate pass rate. The macro-average pass rate under the baseline is 41.2%; adding hints yields 40.6% ($\Delta = -0.6$pp) and adding snippets yields 40.9% ($\Delta = -0.4$ pp). Snippets outperform hints on 8 of 13 models, hints win on 4, and 1 model ties (Table 3). Neither source improves aggregate accuracy, but snippets have a slight edge in breadth.

Augmentation helps where parametric knowledge is weakest. The picture changes substantially on the 219 problems where the zero-shot baseline pass rate is below 25% (collectively the "hard" subset). On these, hints lift the pooled pass rate by $+2.9$pp (12.9% $\rightarrow$ 15.8%) and snippets by $+3.9$ pp (12.9% $\rightarrow$ 16.9%). The positive signal is confined to problems where the model's parametric knowledge is sparse enough that external context genuinely fills a gap. In cases where the model already retrieves the algorithm reliably, injecting context provides no additional signal and can introduce noise that displaces correct reasoning.

Snippets vs. hints on complex algorithms. On the hardest graph-algorithm instances (e.g., Hopcroft–Karp, Stoer-Wagner, Blossom), snippets provide more consistent gains than hints. Within the graph-algorithm domain specifically, snippets lift the pooled pass rate by $+1.4$ pp versus $+0.6$ pp for hints, where a concrete, executable reference is more informative than language-agnostic pseudocode. QwQ-32B, the strongest reasoning model, gains $+5.0$ pp from snippets versus $+4.0$pp from hints, consistent with the view that executable context benefits models already capable of integrating it. This suggests that for low-frequency, structurally complex algorithms whose pseudocode descriptions are themselves specialized, a concrete executable snippet is a more effective retrieval cue than an abstract hint.

6.3 RQ3: SFT vs. GRPO Fine-Tuning for Closing Language and Representation Gaps

Prompt augmentation treats retrieval failures as a deployment-time problem, patching them case by case. Fine-tuning with execution-based rewards takes a different approach: adjusting the model's weights so that retrieval becomes more reliable across languages and representations without requiring external context. We apply GRPO (Shao et al., 2024) to gpt-oss-20B using execution correctness as the reward signal, and compare the resulting checkpoint against both the base model and a supervised fine-tuning (SFT) baseline trained on the positive-reward subset of the same training examples.

We fine-tune gpt-oss-20B using LoRA adapters (r=16, α=32, dropout = 0.05) on all seven projection matrices, keeping approximately 0.5% of

parameters trainable. Both GRPO and SFT are trained on the 56-algorithm training split (447 examples spanning all 7 languages and up to 3 input representations per algorithm), with 22 algorithms held out for evaluation. Training examples are organized into representative groups where all examples sharing a canonical algorithm name (across languages and representations) constitute one group. The reward function uses a four-tier ladder: exact stdout match ($r = 1.0$), normalized match after whitespace collapsing ($r = 0.8$), wrong output with clean execution ($r = 0.3$), and any runtime crash or static failure ($r = 0.0$–0.1). Advantages are computed within each repr group via standard GRPO normalization. The groups with zero reward variance are skipped. The SFT baseline is trained via rejection sampling on the 367 examples where the base model already achieves reward 1.0, using the same LoRA configuration, isolating the effect of policy-gradient credit assignment from the effect of simply training on correct solutions. Full hyperparameters appear in Appendix E.

	Pass@1 (%) on held-out test split							
Method	**Overall**	**C**	**C++**	**Go**	**Java**	**JS**	**Python**	**Rust**
Base	90.1	81.0	85.7	**100.0**	95.2	85.7	**96.2**	85.7
SFT	**94.1**	**85.7**	**95.2**	**100.0**	**100.0**	**95.2**	**96.2**	85.7
GRPO	92.1	**85.7**	76.2	**100.0**	95.2	**100.0**	**96.2**	**90.5**
Δ *SFT*	**+3.9**	**+4.8**	**+9.5**	0.0	**+4.8**	**+9.5**	0.0	0.0
Δ *GRPO*	**+2.0**	**+4.8**	**−9.5**	0.0	0.0	**+14.3**	0.0	**+4.8**

Table 4: **Pass@1 (%) on the held-out test split (152 problems, 22 algorithms unseen during training) for the base model gpt-oss-20b, SFT, and GRPO.** Δ rows show absolute change vs. base in percentage points. SFT leads overall (+3.9 pp) with consistent gains across C, C++, Java, and JavaScript; GRPO achieves the largest JavaScript gain (+14.3 pp) and improves C and Rust, but regresses on C++ (−9.5 pp).

GRPO and SFT improve retrieval accuracy in complementary ways (Table 4). SFT achieves broader and more consistent gains, lifting the overall pass rate by +3.9 pp relative to the base model. Because SFT is trained only on already-correct outputs, these gains reflect consolidation: the model reproduces algorithms it has already partially internalised with greater consistency across languages. GRPO's overall gain is smaller (+2.0 pp), but its partial-credit tiers deliver signal that binary supervision cannot. Taken together, execution-based rewards add value beyond training on correct solutions alone, but their benefit is concentrated in languages whose failure distribution carries gradable structure.

7 Conclusion

Code generation benchmarks treat training-data overlap as a confound to eliminate. We argue the opposite: for well-known algorithms, overlap is the precondition, and the question worth asking is not whether a model can synthesise a solution but whether it reliably reproduces one it has already internalised. We find that the most operationally significant failures are wrong-output errors: syntactically valid, runnable code that silently computes the wrong result, invisible without a concrete test case, and present even in the best-performing models. Retrieval failures are partially remediable, but in complementary ways. Prompt augmentation helps only where parametric knowledge is genuinely absent. Execution-based fine-tuning acts at the weight level: SFT achieves broader and more consistent gains across languages (+3.9 pp overall), while GRPO's execution-based reward concentrates signal on specific languages, yielding the largest single-language gain on JavaScript (+14.3 pp) but regressing on C++ (−9.5 pp). Taken together, parametric code retrieval is a measurable and improvable capability, but one that fails silently in ways current evaluation practice tends to miss, and that merits systematic validation before AI-generated algorithmic code is trusted in production.

Limitations

The results above are subject to three constraints that bound what behavioral evaluation can establish about parametric code retrieval.

A model may pass t_p while silently failing on edge cases such as empty graphs, disconnected components, duplicate keys, or negative-weight cycles, that the fixed input test case does not exercise. EvalPlus (Liu et al., 2023) demonstrated that augmenting HumanEval with additional test cases reduces pass rates substantially. A similar augmentation of ALGOREVAL would tighten the correctness criterion, but was not feasible at the scale of 599 problems across 7 languages without a verified oracle for each variant.

Our core framing assumes that all 77 algorithms appear in the pretraining corpus of every model we evaluate, and interprets failures as surface-form brittleness rather than genuine knowledge absence. This assumption is untestable: pretraining data composition is rarely disclosed at per-algorithm or

per-language granularity. For some low-frequency algorithms in low-resource languages (Go, Rust), what we label a retrieval failure may instead reflect a genuine coverage gap in the pretraining data. Without access to training data, the retrieval and knowledge-absence failure modes cannot be separated.

The paper frames correct model outputs as evidence of parametric retrieval, but external behavior alone cannot fully isolate retrieval from synthesis. A clean separation would require either mechanistic tracing of which training examples causally influence a given generation, or behavioral probes on which a retrieved and a synthesized implementation would make systematically different predictions, neither is available to output-level evaluation.

References

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

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021b. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. 2023. Quantifying memorization across neural language models. In *The Eleventh International Conference on Learning Representations*.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei A Zaharia, Joseph Gonzalez, and Ion Stoica. 2025. Why do multi-agent llm systems fail? In *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc.

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, and 39 others. 2021. Evaluating large language models trained on code.

Zorik Gekhman, Roee Aharoni, Eran Ofek, Mor Geva, Roi Reichart, and Jonathan Herzig. 2026. Thinking to recall: How reasoning unlocks parametric knowledge in llms. *Preprint*, arXiv:2603.09906.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

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. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.

Jing Huang, Diyi Yang, and Christopher Potts. 2024. Demystifying verbatim memorization in large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 10711–10732, Miami, Florida, USA. Association for Computational Linguistics.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *Preprint*, arXiv:2403.07974.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The stack: 3 tb of permissively licensed source code. *Preprint*, arXiv:2211.15533.

Seonghyeon Lee, HeeJae Chon, Joonwon Jang, Dongha Lee, and Hwanjo Yu. 2025. How diversely can language models solve problems? exploring the algorithmic diversity of model-generated code. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 152–167, Suzhou, China. Association for Computational Linguistics.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 47 others. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Alexandre Matton, Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon, and Matthias Gallé. 2024. On leakage of code generation evaluation datasets. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 13215–13223, Miami, Florida, USA. Association for Computational Linguistics.

Nickil Maveli, Antonio Vergari, and Shay B. Cohen. 2026. Can llms compress (and decompress)? evaluating code understanding and execution via invertibility. *Preprint*, arXiv:2601.13398.

Fabio Petroni, Tim Rocktäschel, Sebastian Riedel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, and Alexander Miller. 2019. Language models as knowledge bases? In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2463–2473, Hong Kong, China. Association for Computational Linguistics.

Martin Riddell, Ansong Ni, and Arman Cohan. 2024. Quantifying contamination in evaluating code generation capabilities of language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14116–14137, Bangkok, Thailand. Association for Computational Linguistics.

Adam Roberts, Colin Raffel, and Noam Shazeer. 2020. How much knowledge can you pack into the parameters of a language model? In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 5418–5426, Online. Association for Computational Linguistics.

Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. 2025. Vibe coding vs. agentic coding: Fundamentals and practical implications of agentic ai. *Preprint*, arXiv:2505.19443.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *Preprint*, arXiv:2402.03300.

Luca Soldaini, Rodney Kinney, Akshita Bhagia, Dustin Schwenk, David Atkinson, Russell Authur, Ben Bogin, Khyathi Chandu, Jennifer Dumas, Yanai Elazar, Valentin Hofmann, Ananya Harsh Jha, Sachin Kumar, Li Lucy, Xinxi Lyu, Nathan Lambert, Ian Magnusson, Jacob Morrison, Niklas Muennighoff, and 17 others. 2024. Dolma: An Open Corpus of Three Trillion Tokens for Language Model Pretraining Research. *arXiv preprint*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *Preprint*, arXiv:2302.13971.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025. CodeRAG-bench: Can retrieval augment code generation? In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3199–3214, Albuquerque, New Mexico. Association for Computational Linguistics.

Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. 2025. Where llm agents fail and how they can learn from failures. *Preprint*, arXiv:2509.25370.

A Algorithms in AlgoREval

Table 5 lists all 77 algorithms included in AlgoREval, grouped by domain.

B Models

Table 6 lists all 15 models evaluated in this work, along with their HuggingFace model IDs (or API identifiers for closed models), parameter counts, organizations, and licenses.

C Zero-Shot Inference Hyperparameters

Table 7 summarises the generation parameters used. Open-weight models were served locally via vLLM and run with greedy decoding (temperature $= 0$, top-$p = 1.0$). Closed API models (GPT-4o-mini, GPT-4.1-mini) were queried through the OpenAI API with temperature $= 0.7$ and a smaller token budget, as the API enforces its own safety caps. Thinking models (QwQ-32B, DeepSeek-R1-Distill-Llama-8B, DeepSeek-R1-Distill-Qwen-14B) share the vLLM settings but require a large output budget to avoid truncating the internal reasoning trace before code is emitted; the 32,768-token limit serves as both the thinking budget and the code output window. A per-problem execution timeout of 30 seconds was applied uniformly across all models and languages.

D Prompt-Augmentation Setup

It tests whether retrieval failures can be remediated by injecting external context into the prompt. We evaluate two conditions using **+hint** and **+snippet** across the same 13 open-weight vLLM models used in the zero-shot evaluation (no closed-API models). Generation hyperparameters are identical to zero-shot: temperature $= 0$, max output tokens $= 32{,}768$, top-$p = 1.0$, execution timeout $= 30$ s.

+Hint condition. Structured algorithmic hints were generated offline using `gpt-4o-mini-2024-07-18` at a temperature of 0. Each hint covers three fields: (1) language-agnostic pseudocode ($\leq$20 lines), (2) key data structures, and (3) Big-O time/space complexity. Because the hints are language-agnostic, the same hint is prepended to all seven language variants of a given algorithm. Hints were generated for all 77 unique algorithms.

+Snippet condition. One reference implementation per (algorithm, language) pair was retrieved from GitHub using the GitHub Search API (10 results per query, sorted by index date). Results from RosettaCode and GeeksforGeeks were excluded to avoid overlap with the benchmark's source material.

Each retrieved file was truncated to at most 80 lines, centred on the most keyword-relevant region. A 2.2 s inter-request delay was used to respect the GitHub API's rate limit. Coverage-wise, 431 of 599 problems received an injected snippet. The remaining 168 (no language-matched result found) fell back to the unaugmented baseline prompt.

Prompt injection. In both conditions, the system message remains unchanged relative to the zero-shot baseline. The injected block is inserted immediately before the stdlib-constraint footer, so it is not buried after the final instruction.

The snippet block uses the prefix: *"For reference, here is an implementation of this algorithm in {lang} retrieved from a different source. You may use it as a reference but must write your own solution."* followed by a fenced code block.

The hint block uses the header *"Algorithmic hint:"* followed by the pseudocode, data-structure list, and complexity note. Table 8 summarises the setup.

E Fine-Tuning Hyperparameters

Both GRPO and SFT fine-tune gpt-oss-20B using LoRA adapters applied to all seven projection matrices (q, k, v, o, gate, up, down), keeping approximately 0.5% of parameters trainable. Table 9 lists the shared and condition-specific hyperparameters.

F Prompt Template

All models are prompted with a fixed system message followed by a per-problem user message. Figure 4 shows a representative example for the Hopcroft–Karp algorithm in Python (edge-list representation).

G Prompt Augmentation Examples

Figures 5 and 6 show the complete user messages fed to models under each RQ2 augmentation condition for the Hopcroft–Karp (Python) example. The system message is identical to Figure 4 in all conditions. The augmentation block (shaded region) is inserted immediately before the stdlib-constraint

System message

```
You are an expert software engineer tasked with implementing algorithms correctly and efficiently.
Output ONLY a single fenced code block with the complete, self-contained program. No explanations,
no prose, no install commands — just the code block.
Requirements:
• The program must run/compile as-is without any modifications.
• Use only the language's standard library.
• Do not add any extra output beyond what the problem specifies.
• The code must be in a single file.
[language-specific notes omitted for brevity]
```

User message (Hopcroft–Karp, Python, edge-list)

```
Implement the Hopcroft-Karp algorithm for maximum matching in a bipartite graph in Python.
Requirements:
• Provide a class BipGraph(m, n) with:
  – addEdge(u, v) using 1-based indices (u in U=1..m, v in V=1..n)
  – hopcroftKarp() returning the integer size of the maximum matching.
• The code must be runnable as a script.
• Include the following exact test case under if __name__ == "__main__": (do not change it):

if __name__ == "__main__":
    g = BipGraph(4, 4)
    g.addEdge(1, 2)
    g.addEdge(1, 3)
    g.addEdge(2, 1)
    g.addEdge(3, 2)
    g.addEdge(4, 2)
    g.addEdge(4, 4)
    print("Size of maximum matching is %d"
          % g.hopcroftKarp())

Runtime constraints (strict):
• Use only the language standard library.
• Do NOT use third-party packages or package managers.
• Output only executable code.
```

Figure 4: Prompt structure used for all evaluations. The system message is fixed across all problems and languages. The user message is generated per problem; it names the algorithm and target language, specifies the required interface, and embeds an executable test case whose output is compared against the ground-truth reference output.

footer so the injected context is visible before the final instruction.

H Error Analysis

Figure 8 breaks down evaluation outcomes per model across all 599 problems. Stronger models fail almost exclusively on wrong-output errors, while weaker models are dominated by compile errors, reflecting a gradient from syntactic to semantic retrieval failure. Figure 9 plots pass rate against consensus recall, showing that models producing correct outputs do not uniformly converge on the same answer — consistent with independent retrieval rather than recitation of a shared token sequence.

I Solution Diversity by Model and Language

Figure 7 shows the AlgoSim novel rate per model and language, computed over 2,215 cross-model pairwise comparisons across 40 algorithms, 7 languages, and 15 models ($\binom{15}{2}$=105 pairs per problem), judged by GPT-4.1-mini. 20.5% of pairs were judged novel overall.

J Per-Language Algorithm Difficulty Heatmaps

Figures 10–16 show per-language pass-rate heatmaps for all 77 algorithms in AlgoReval across 15 zero-shot models. Each cell reports the fraction of trials in which the model's output matched the ground-truth expected value, averaged over all input-representation variants available for

User message — +Hint condition (Hopcroft–Karp, Python)

```
Implement the Hopcroft-Karp algorithm for maximum matching in a bipartite graph in Python.
Requirements:
• Provide a class BipGraph(m, n) with:
  – addEdge(u, v) using 1-based indices (u in U=1..m, v in V=1..n)
  – hopcroftKarp() returning the integer size of the maximum matching.
• The code must be runnable as a script.
• Include the following exact test case under if __name__ == "__main__": (do not change it):

if __name__ == "__main__":
    g = BipGraph(4, 4)
    g.addEdge(1, 2)
    g.addEdge(1, 3)
    g.addEdge(2, 1)
    g.addEdge(3, 2)
    g.addEdge(4, 2)
    g.addEdge(4, 4)
    print("Size of maximum matching is %d" % g.hopcroftKarp())
```

Injected hint block

```
Algorithmic hint:
Pseudocode:
1. Initialize matching M to empty
2. While there exists an augmenting path P in the graph:
   2.1. Augment the matching M along path P
3. Return matching M
4. Function to find augmenting path P:
   4.1. Initialize queue Q and visited set V
   4.2. For each vertex u in the left set:
       4.2.1. If u is unmatched, add u to Q and V
   4.3. While Q is not empty:
       4.3.1. Dequeue vertex u from Q
       4.3.2. For each neighbor v of u in the right set:
           4.3.2.1. If v is unmatched or matched vertex of v
                    can find an augmenting path:
               4.3.2.1.1. Add v to Q and V
               4.3.2.1.2. Record the path from u to v
   4.4. If an augmenting path is found, return it
   4.5. Otherwise, return null
Key data structures: queue, visited set, matching set
Complexity: O(E*sqrt(V)) time, O(V) space
```

```
Runtime constraints (strict):
• Use only the language standard library.
• Do NOT use third-party packages/crates/modules.
• Do NOT use package managers (pip/npm/cargo/maven/gradle/etc.).
• If a data structure/algorithm is missing, implement it directly in code.
• Output only executable code.
```

Figure 5: Complete user message for Hopcroft–Karp (Python) under the **+Hint** condition. The green-shaded block is the only addition relative to the zero-shot baseline; the same language-agnostic hint is used for all seven language variants of the algorithm. Generated offline by `gpt-4o-mini-2024-07-18` at temperature $= 0$.

that (algorithm, language) pair. Grey cells indicate that no predictions were collected for that combination. Rows are sorted by global mean pass rate (averaged over all languages and all models), so the hardest algorithms appear at the top and the easiest at the bottom; this ordering is held *fixed* across all seven figures to facilitate cross-language comparison. Columns are sorted by each model's mean pass rate within the given language (weakest left, strongest right), so column order may vary between figures. The right-hand axis shows each algorithm's mean pass rate across all models for that language.

Domain	Algorithm
Backtracking	N-Queens Problem Permutations Generator Sudoku Solver
Compression	Huffman Coding Run-Length Encoding
Computational Geometry	Closest Pair of Points Convex Hull Point in Polygon Polygon Area
Cryptography	Caesar Cipher SHA-256 from Scratch Vigenere Cipher
Data Structures	AVL Tree Disjoint Set Union Fenwick Tree Segment Tree Trie
Dynamic Programming	0/1 Knapsack Edit Distance Longest Common Subsequence Longest Increasing Subsequence
Graph Algorithms	Articulation Points Bellman-Ford Shortest Path Blossom Algorithm — General Maximum Matching Borůvka's MST Bridge Finding Dijkstra's Shortest Path Dinic's Max Flow Eulerian Path/Circuit Floyd-Warshall All-Pairs Shortest Path Gabow's SCC Hierholzer's Eulerian Circuit Hopcroft–Karp Johnson's All-Pairs Shortest Paths Karger's Min Cut Kosaraju's SCC Kruskal's MST Min-Cost Max-Flow Prim's MST SCC by Mutual Reachability Shortest Path in DAG Stoer-Wagner Min Cut Tarjan's SCC Topological Sort Two-SAT
Image Processing	2D Convolution with Arbitrary Kernel Box Blur (3×3 Averaging Filter) Gaussian Blur 3×3 Histogram Equalization Median Filter 3×3 Otsu Thresholding Prewitt Edge Detection Rotate Image 90° Clockwise Rotate Image 180° Sobel Edge Detection
Number Theory	Extended GCD Modular Exponentiation Sieve of Eratosthenes
Numerical Methods	Fast Fourier Transform Gaussian Elimination Newton-Raphson Method Simpson's Rule
Other	Banker's Algorithm for Deadlock Avoidance FCFS CPU Scheduling FIFO Page Replacement LRU Page Replacement Non-Preemptive SJF CPU Scheduling Round Robin CPU Scheduling
Parsing	Postfix Expression Evaluator Shunting-Yard Algorithm
Sorting Algorithms	Counting Sort Heap Sort Merge Sort Radix Sort
String Algorithms	KMP Pattern Search Rabin-Karp Pattern Search Z-Algorithm Pattern Search

Table 5: All 77 algorithms in ALGOREVAL, grouped by domain (14 domains total).

Model	Model ID	Params	Organization	License
CodeGemma-7B	`google/codegemma-7b-it`	7B	Google	Gemma
CodeLlama-34B	`codellama/CodeLlama-34b-Instruct-hf`	34B	Meta	Llama 2
Codestral-22B	`mistralai/Codestral-22B-v0.1`	22B	Mistral AI	MNPL
DeepSeek-Coder-33B	`deepseek-ai/deepseek-coder-33b-instruct`	33B	DeepSeek	DeepSeek
DeepSeek-R1-Llama-8B	`deepseek-ai/DeepSeek-R1-Distill-Llama-8B`	8B	DeepSeek	MIT
DeepSeek-R1-Qwen-14B	`deepseek-ai/DeepSeek-R1-Distill-Qwen-14B`	14B	DeepSeek	MIT
GPT-4.1-mini	`gpt-4.1-mini-2025-04-14` (API)	—	OpenAI	Proprietary
GPT-4o-mini	`gpt-4o-mini-2024-07-18` (API)	—	OpenAI	Proprietary
gpt-oss-20B	`openai/gpt-oss-20b`	20B	OpenAI	MIT
Llama-3.1-8B	`meta-llama/Llama-3.1-8B-Instruct`	8B	Meta	Llama 3
Phi-4	`microsoft/phi-4`	14B	Microsoft	MIT
QwQ-32B	`Qwen/QwQ-32B`	32B	Alibaba	Apache 2.0
Qwen2.5-7B	`Qwen/Qwen2.5-7B-Instruct`	7B	Alibaba	Apache 2.0
StarCoder2-15B	`bigcode/starcoder2-15b-instruct-v0.1`	15B	BigCode	OpenRAIL-M
Yi-Coder-9B	`01-ai/Yi-Coder-9B-Chat`	9B	01.AI	Apache 2.0

Table 6: All 15 models evaluated in ALGOREVAL. Open-weight models were run locally via vLLM. Closed models were accessed through their respective APIs. Parameter counts for closed API models are not publicly disclosed.

User message — +Snippet condition (Hopcroft–Karp, Python)

Implement the Hopcroft-Karp algorithm for maximum matching in a bipartite graph in **Python**.
Requirements:
- Provide a class BipGraph(m, n) with:
  - addEdge(u, v) using **1-based** indices (u in U=1..m, v in V=1..n)
  - hopcroftKarp() returning the integer size of the maximum matching.
- The code must be runnable as a script.
- Include the following exact test case **under** if __name__ == "__main__": (do not change it):

```
if __name__ == "__main__":
    g = BipGraph(4, 4)
    g.addEdge(1, 2)
    g.addEdge(1, 3)
    g.addEdge(2, 1)
    g.addEdge(3, 2)
    g.addEdge(4, 2)
    g.addEdge(4, 4)
    print("Size of maximum matching is %d" % g.hopcroftKarp())
```

Injected snippet block

For reference, here is an implementation of this algorithm in python
retrieved from a different source. You may use it as a reference but
must write your own solution:

```python
"""BipartiteMatching.py

Hopcroft-Karp bipartite maximum-cardinality matching and maximum
independent set for bipartite graphs.

D. Eppstein, April 2002.
"""

from .strong_connectivity import StronglyConnectedComponents


def matching(graph):
    """Find maximum cardinality matching of a bipartite graph (U,V,E).
    The input format is a dictionary mapping members of U to lists
    of their neighbors in V.
    """
    # initialize greedy matching (redundant, but faster than full search)
    matching = {}
    for u in graph:
        for v in graph[u]:
            if v not in matching:
                matching[v] = u
                break

    while True:
        # structure residual graph into layers
        preds = {}
        unmatched = []
        pred = {u: unmatched for u in graph}
        for v in matching:
            del pred[matching[v]]
        layer = list(pred)

        # repeatedly extend layering structure by another pair of layers
        while layer and not unmatched:
            newLayer = {}
            for u in layer:
                for v in graph[u]:
                    if v not in preds:
                        newLayer.setdefault(v, []).append(u)
            layer = []
            for v in newLayer:
                preds[v] = newLayer[v]
                if v in matching:
                    layer.append(matching[v])
                    pred[matching[v]] = v
                else:
                    unmatched.append(v)

        # did we finish layering without finding any alternating paths?
        if not unmatched:
            unlayered = {}
            for u in graph:
                for v in graph[u]:
                    if v not in preds:
                        unlayered[v] = None
            return (matching, list(pred), list(unlayered))

        # recursively search backward through layers to find alternating paths
        def recurse(v):
            if v in preds:
                L = preds[v]
                del preds[v]
                for u in L:
                    if u in pred:
                        pu = pred[u]
                        del pred[u]
                        if pu is unmatched or recurse(pu):
```

Runtime constraints (strict):
- Use only the language standard library.
- Do NOT use third-party packages/crates/modules.
- Do NOT use package managers (pip/npm/cargo/maven/gradle/etc.).
- If a data structure/algorithm is missing, implement it directly in code.
- Output only executable code.

Figure 6: Complete user message for Hopcroft–Karp (Python) under the **+Snippet** condition. The orange-shaded block is the only addition relative to the zero-shot baseline. The snippet (D. Eppstein, 2002) was retrieved from GitHub via the GitHub Search API; RosettaCode and GeeksforGeeks results were excluded to avoid overlap with the benchmark's source material. Files are truncated to at most 80 lines centred on the most keyword-relevant region, hence the snippet ends mid-function. Each language variant receives its own language-matched snippet.

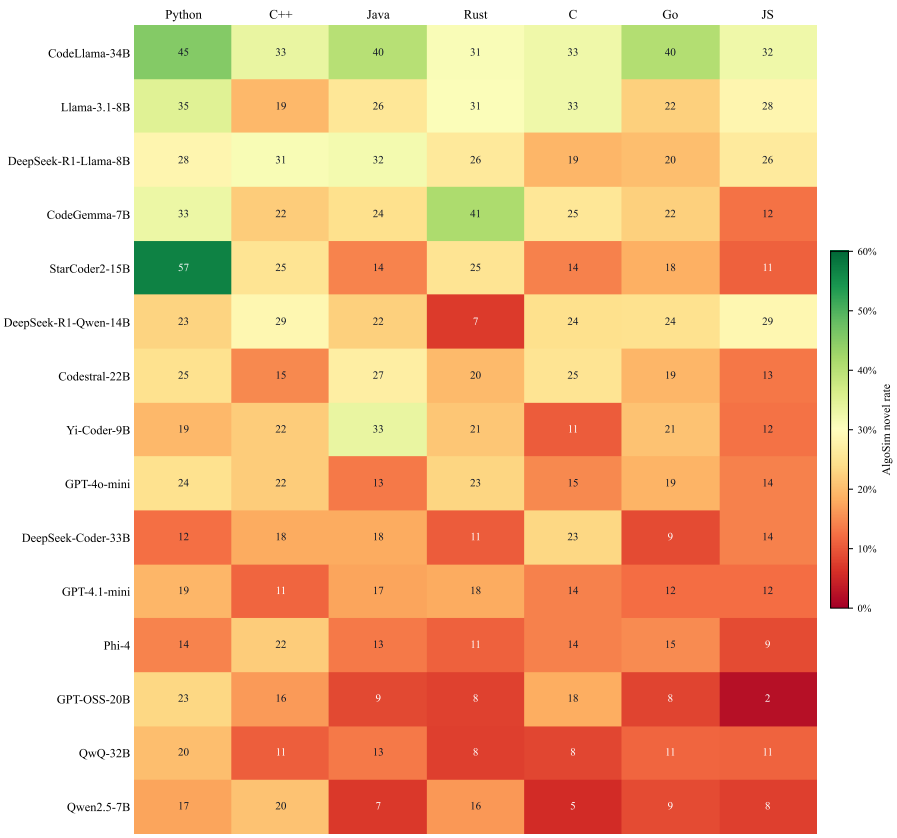

Figure 7: AlgoSim novel rate by model and programming language.

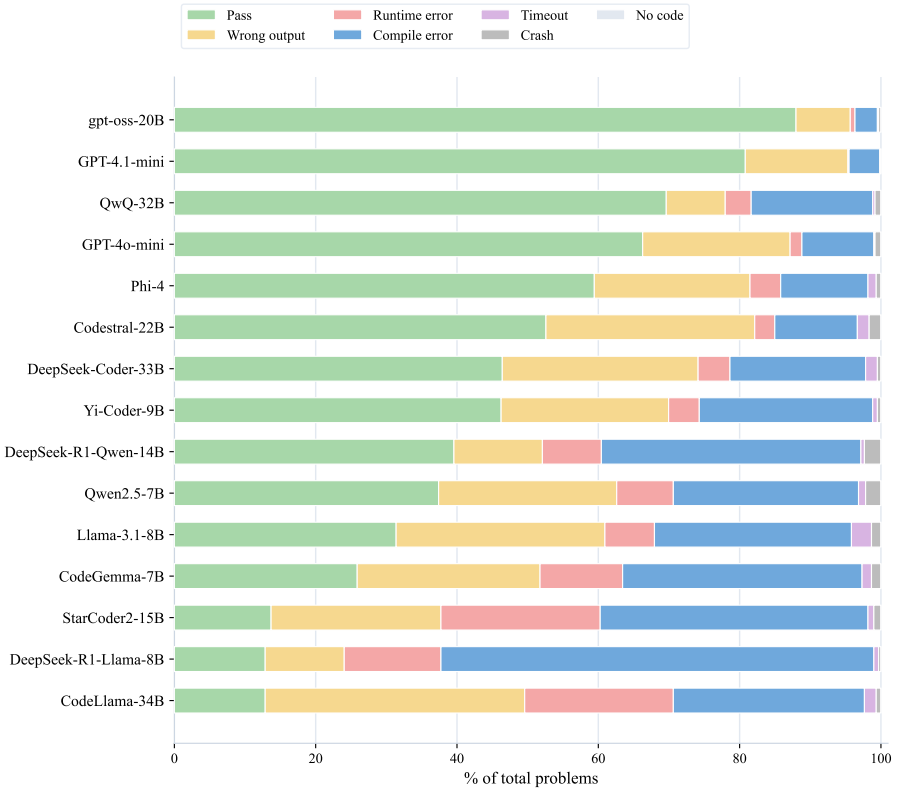

Figure 8: Outcome breakdown per model as a fraction of all 599 problems, sorted by pass rate. Stronger models fail almost exclusively on wrong output, while weaker models are dominated by compile errors, reflecting a gradient from syntactic to semantic retrieval failure.

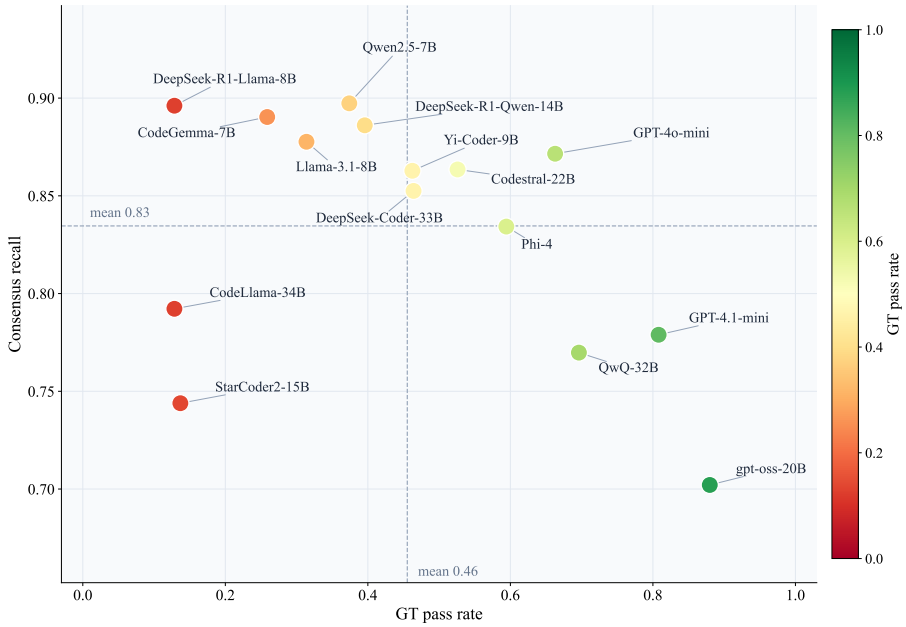

Figure 9: Pass rate vs. consensus recall. GT pass rate (x-axis) is the fraction of problems where a model's output matches the ground-truth expected value. Consensus recall (y-axis) is the fraction of a model's correct answers that also match the plurality output across all models.

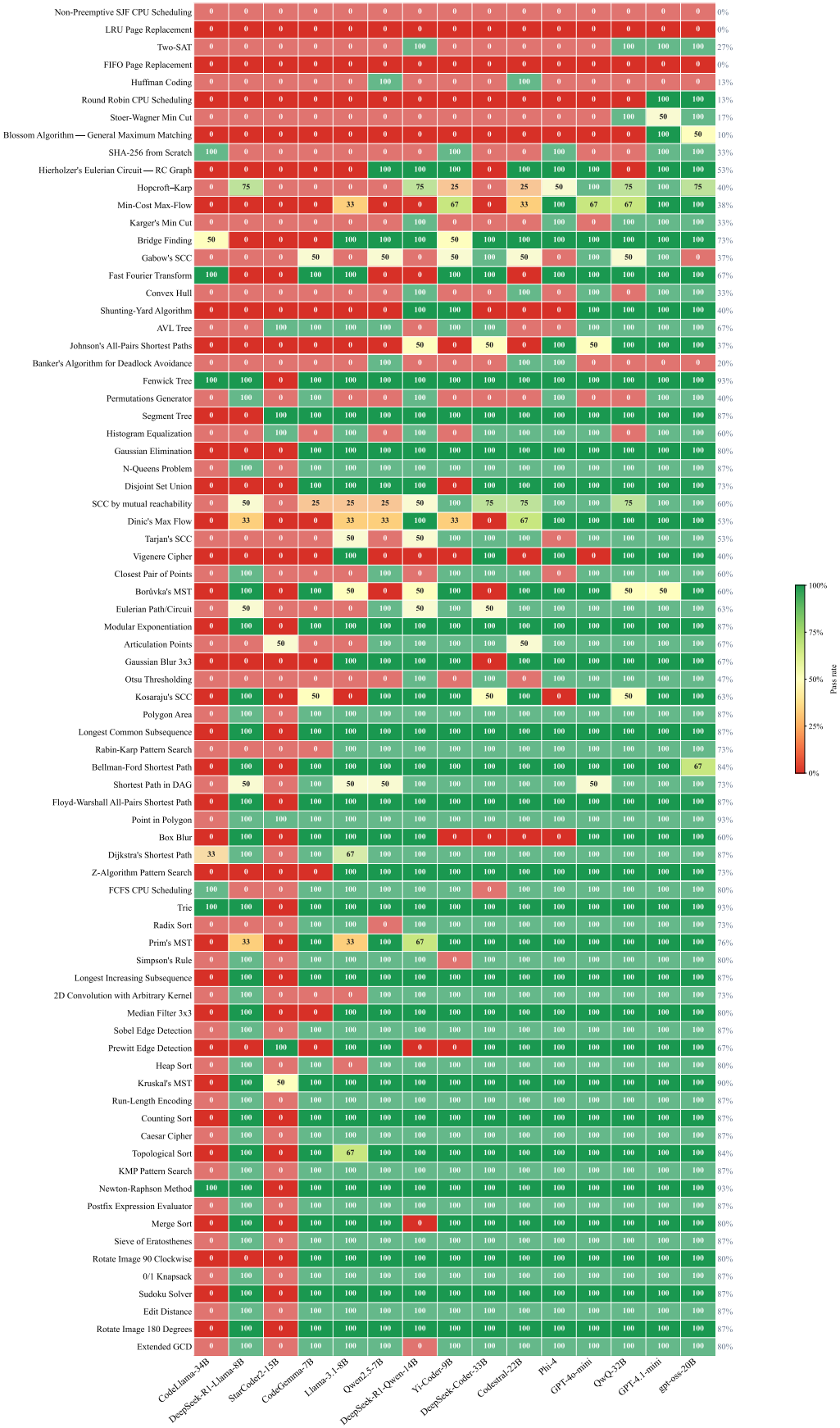

Figure 10: Algorithm difficulty heatmap for Python.

Figure 11: Algorithm difficulty heatmap for C++.

Figure 12: Algorithm difficulty heatmap for Java.

Figure 13: Algorithm difficulty heatmap for C.

Figure 14: Algorithm difficulty heatmap for JavaScript.

Figure 15: Algorithm difficulty heatmap for Rust.

Figure 16: Algorithm difficulty heatmap for Go.

Parameter	Open-weight (vLLM)	Closed API
Temperature	0 (greedy)	0.7
Max output tokens	32,768	8,192
Top-p	1.0	—
Decoding	Greedy	Sampling
Thinking budget	32,768 (thinking models)	N/A
Execution timeout	30 s	30 s
Prompting strategy	Zero-shot	Zero-shot

Table 7: Generation hyperparameters for the zero-shot evaluation. Open-weight models include all 13 locally-served models. Closed API covers GPT-4o-mini and GPT-4.1-mini. The thinking-model token floor applies to QwQ-32B, DeepSeek-R1-Distill-Llama-8B, and DeepSeek-R1-Distill-Qwen-14B.

Parameter	+Hint	+Snippet
Models evaluated	13 vLLM (open-weight)	13 vLLM (open-weight)
Temperature	0 (greedy)	0 (greedy)
Max output tokens	32,768	32,768
Top-p	1.0	1.0
Execution timeout	30 s	30 s
Context source	GPT-4o-mini (temp. 0)	GitHub Search API
Context granularity	Per algorithm (lang-agnostic)	Per (algorithm, language)
Max context length	$\leq$20 pseudocode lines	$\leq$80 code lines
Problem coverage	599/599	599/599
Excluded sources	N/A	RosettaCode, GeeksforGeeks

Table 8: Prompt-augmentation parameters. Generation settings are identical to the zero-shot baseline. Only the prompt contents differ between conditions.

Parameter	GRPO	SFT
Base model	gpt-oss-20B	gpt-oss-20B
LoRA rank r	16	16
LoRA α	32	32
LoRA dropout	0.05	0.05
Trainable params	$\approx$0.5%	$\approx$0.5%
Training algorithms	56 (447 examples)	56 (367 positive examples)
Training examples	All 7 langs, $\leq$3 repr	Rejection-sampled (r=1.0)
Held-out test split	22 algorithms, 152 problems	22 algorithms, 152 problems
Grouping	Repr groups (algo $\times$ lang $\times$ repr)	N/A
Reward tiers	1.0 / 0.8 / 0.3 / 0.0–0.1	N/A
Optimiser	AdamW	AdamW
Epochs	3	3
Execution timeout	30 s	30 s

Table 9: Fine-tuning hyperparameters for GRPO and SFT. Both conditions use the same LoRA configuration and train/test split. SFT trains only on examples where the base model already achieves r=1.0 (rejection sampling), isolating the contribution of policy-gradient credit assignment from the effect of training on correct solutions alone.