Can LLMs Compress (and Decompress)? Evaluating Code Understanding and Execution via Invertibility

Nickil Maveli **Antonio Vergari** **Shay B. Cohen**
School of Informatics, University of Edinburgh
10 Crichton Street, Edinburgh, EH8 9AB
{nickil.maveli,avergari,scohen}@ed.ac.uk

Abstract

LLMs demonstrate strong performance on code benchmarks, yet consistent reasoning across forward and backward execution remains elusive. We present RoundTripCodeEval (RTCE), a benchmark of four code execution reasoning tasks that evaluates round-trip consistency through execution-free, exact-match assessment of bijection fidelity across four lossless compression algorithms. We evaluate state-of-the-art Code-LLMs under zero-shot prompting, supervised fine-tuning on execution traces, and iterative self-reflection. All approaches yield only modest improvements and none closes the gap, revealing that current LLMs lack the internal coherence required for reliable bidirectional code reasoning. RTCE surfaces findings invisible to existing benchmarks: models frequently pass individual forward and backward tasks yet fail the combined round-trip, exposing mutually inconsistent internal representations; SFT and self-reflection saturate after one revision round, indicating they cannot repair fundamental algorithmic misunderstandings; and failures persist even on simple bijections such as RLE, suggesting that algorithmic complexity is not the sole root cause.[1]

1 Introduction

Can LLMs understand code and algorithms? What would it mean for them to understand code and algorithms to begin with? We frame such understanding as a *code inversion* problem. Indeed, recent progress in Code-LLMs (Zheng et al., 2024; Lozhkov et al., 2024; Hui et al., 2024; Guo et al., 2024) has demonstrated remarkable performance across various software engineering applications. However, evaluating the reasoning ability of Code-LLMs requires going beyond isolated input–output predictions. Most existing code reasoning benchmarks evaluate single-direction execution, either forward execution, i.e., predicting the output from the input and code (Gu et al., 2024; Jain et al., 2025; Chen et al., 2025; Li et al., 2025; Liu et al., 2024) or backward execution, i.e., inferring the input from the output and code (Gu et al., 2024; Li et al., 2025). While these tasks are valuable, they overlook a key property of robust reasoning systems: the ability to integrate forward and backward execution into a coherent and reversible process (Jiang et al., 2024). Forward execution can often be solved through surface-level pattern matching, memorisation, or statistical correlation rather than genuine mechanistic understanding (Gu et al., 2024; Jain et al., 2025). Inversion, however, is fundamentally different. It requires the model to understand the forward execution as a bijective encoding function and then construct a corresponding decoding function that perfectly recovers the original input. An LLM might achieve high accuracy in one direction, yet fail to maintain logical compatibility when the process is inverted, leading to inconsistencies (Min et al., 2024; Allamanis et al., 2024; Xu et al., 2025; Liu et al., 2025). Success on a round-trip inversion provides more substantial evidence of deep semantic code understanding than forward-only accuracy. Because inversion cannot be solved by local pattern matching alone, models must implicitly construct a consistent internal execution model. Failures to close the loop and achieve self-consistency thus indicate that forward correctness was fragile, derived from template matching and API memorisation (Wang et al., 2024) rather than mechanistic reasoning (He et al., 2025) about data flow and control logic. Investigation into these areas is vital to advance LLMs from pattern-based generation tools to mechanistically grounded, trustworthy code assistants capable of deep, bidirectional understanding.

To address this gap, we introduce a round-trip code evaluation framework where the model must apply a transformation (encoding) and its inverse

[1] Code and dataset are available at https://github.com/Nickil21/round-trip-code-compression. denotes shared supervision.

Findings of the Association for Computational Linguistics: ACL 2026, pages 25625–25660
July 2-7, 2026

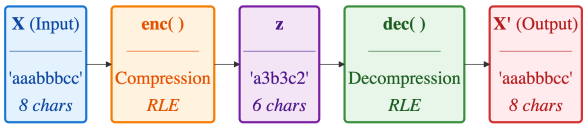


Figure 1: A standard lossless compression pipeline, where an input X is encoded into a compact representation z and then decoded back into X'. Here, enc and dec can be referred to as the encoding and decoding functions. We outline the overall workflow of our pipeline using a concrete example in Appendix A.

(decoding) such that the final reconstruction is identical to the original input. The transformation is a bijection because it is both injective, mapping each input to a unique encoded output, and surjective, covering the entire encoded space so that decoding perfectly recovers the original input. The inherent invertibility and enforced self-consistency offer a robust and comprehensive diagnostic for LLM reasoning capabilities. By simultaneously testing the completeness of the forward mapping and the accuracy of the inverse mapping, this framework exposes subtle asymmetries, inconsistencies, and reasoning failures that are undetectable by conventional single-direction evaluations. Through this closed-loop evaluation, we move beyond isolated input–output correctness, instead quantifying a model's internal coherence, bidirectional reasoning strength, and reliability, which are indispensable in code-based applications that demand high precision, transparency, and robustness.

We construct RTCE in a controlled setting with 250 inputs per algorithm drawn from diverse real-world sources, providing a rigorous measure of LLM reasoning in challenging scenarios. The benchmark's core challenge is the inversion task, which goes beyond the forward execution traces on which LLMs are primarily trained. The forward task is essentially a code-completion or execution task familiar to the model; the inverse task forces the model to treat the given function's output as input and infer the original semantics to reverse the operation. We evaluate models under zero-shot prompting to establish a baseline, supervised fine-tuning on execution traces to maximise the forward signal, and iterative self-reflection to test whether models can self-correct systematic errors. Together, these three paradigms determine whether any current approach can close the round-trip gap, or whether the failure is fundamental to how models internalise code semantics.

2 Problem Statement

Code invertibility, in the context of programming, refers to the ability to reverse a piece of code to its original state or to retrieve the initial input from the output. In this work, we study code invertibility by framing it as a *round-trip* through a lossless, symmetric compression–decompression pipeline as illustrated in Figure 1. We evaluate it through the lens of self-consistency, the requirement that ensures re-encoding any decoded input reproduces the original representation exactly.

Formally, let $x \in \mathcal{X}$ be the original input string, $z \in \mathcal{Z}$ its encoded compressed output, and $x' \in \mathcal{X}$ the reconstruction after decoding. Define the encoding and decoding maps

$$\mathsf{enc}\colon \mathcal{X} \to \mathcal{Z}, \qquad \mathsf{dec}\colon \mathcal{Z} \to \mathcal{X}.$$

We write

$$z = \mathsf{enc}(x), \qquad x' = \mathsf{dec}(z),$$

subject to the *lossless round-trip* constraint

$$\forall\, x \in \mathcal{X}, \quad \mathsf{dec}(\mathsf{enc}(x)) = x.$$

Unlike round-trip tasks involving natural language translation (e.g., code → description → code), which only require semantic equivalence (the final code behaves the same), lossless compression requires exact data identity. This is a much stricter, unambiguous test of mechanistic reasoning. Within this framework, we delineate four closely related prediction tasks as illustrated in Figure 2:

- **Output Prediction:** Given $(x,\ \mathsf{enc})$, predict $\hat{z} \approx \mathsf{enc}(x)$. This measures forward reasoning and checks the LLM's ability to simulate the compression process. We denote it by $x \xrightarrow{\mathsf{enc}} z$.
- **Input Prediction *with Inversion*:** Given $(z,\ \mathsf{enc})$, predict $\hat{x}' \approx \mathsf{enc}^{-1}(z) \equiv \mathsf{dec}(z)$. This evaluates backward reasoning and checks whether the LLM can internally reinterpret the decoder as its inverse encoder to reconstruct the input. We denote it by $z \xrightarrow{\mathsf{enc}^{-1}} x'$.
- **Output Prediction *with Inversion*:** Given $(x,\ \mathsf{dec})$, predict $\hat{z} \approx \mathsf{dec}^{-1}(x) \equiv \mathsf{enc}(x)$. This evaluates backward reasoning and checks whether the LLM can internally reinterpret the encoder as its inverse decoder to obtain the compressed output. We denote it by $x \xrightarrow{\mathsf{dec}^{-1}} z$.
- **Input Prediction:** Given $(z,\ \mathsf{dec})$, predict $\hat{x}' \approx \mathsf{dec}(z)$. This tests forward reasoning and checks the LLM's ability to simulate the decompression process. We denote it by $z \xrightarrow{\mathsf{dec}} x'$.

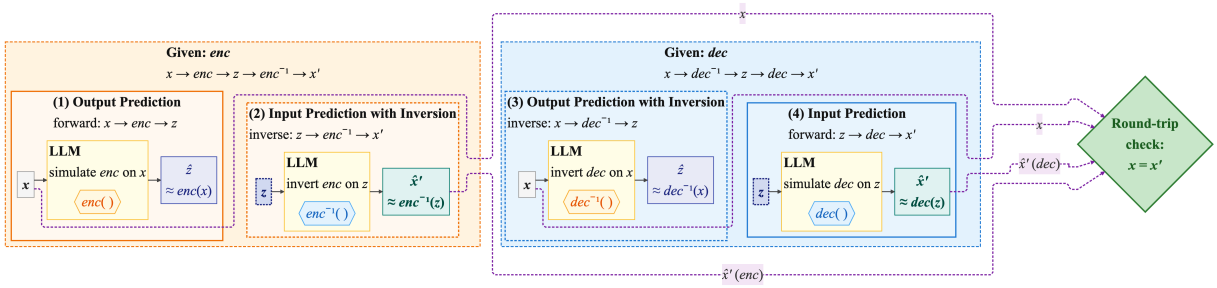


Figure 2: Overview of the reasoning tasks which depict our four-step round-trip procedure for assessing code compression self-consistency. In (1), **Output Prediction**, the input is transformed using the actual encoder $x \xrightarrow{\text{enc}} z$ and the LLM predicts $\hat{z}$ from x and enc. In (2), **Input Prediction *with Inversion***, the compressed output is mapped back to the input space $z \xrightarrow{\text{enc}^{-1}} x'$ using the inverted encoder enc^{-1} to reconstruct $\hat{x}'$. In (3), **Output Prediction *with Inversion***, the input is passed through the inverted decoder $x \xrightarrow{\text{dec}^{-1}} z$ to recover $\hat{z}$. In (4), **Input Prediction**, the compressed output is decoded $z \xrightarrow{\text{dec}} x'$ using the actual decoder dec to produce the final reconstruction $\hat{x}'$. A round-trip check $x{=}x'$ verifies perfect reconstruction fidelity, returning a binary result that indicates if the LLM is self-consistent or not. This corresponds to a chain of length 1, but this could be extended to an arbitrary length.

While there is a class of algorithms for which code inversion is possible (see §4), code inversion is a non-trivial problem in the general case. Consider that any arbitrary algorithmic problem $A(x) = z$ could be changed into a function $B(x) = (x, z) = (x, A(x))$ (including the input in the output). Inverting $B(x)$ into $C(x, z)$ is algorithmically trivial (define C as returning x given (x, z)), but the inversion of C is the arbitrary A. Therefore, for undecidable problems where a witness helps verify an input-output pair (such as a variant of the Post Correspondence problem, where x is two equal-length lists of strings and z is non-empty sublists of each list, with identical indices from the original lists, such that the two concatenations of strings in each sublist are the same), it is easy to construct $C(x, z) = x$ (while implementing a simple check to see whether z is indeed a witness), but inverting C is non-trivial (with the code for the simple check provided), because the original A, solving the PC problem (or providing a witness to a solution), does not exist as a normally executable algorithm.

3 Related Work

Existing benchmarks evaluate forward or backward execution in isolation; none ask whether both directions are mutually consistent. RTCE closes this loop, drawing on the following themes.

Self-consistency. The principle of self-consistency (Wang et al., 2023) has recently gained traction in LLM evaluation, especially for code generation. The IdentityChain framework (Min et al., 2024) reveals that models often generate code and corresponding specifications that appear correct in isolation but become inconsistent when composed, exposing reasoning failures that conventional one-directional benchmarks overlook. Round-Trip Correctness (RTC; Allamanis et al. 2024) evaluates a model's ability to describe code in natural language and then regenerate the original code from that description. By enforcing consistency between the original and reconstructed code, RTC provides an unsupervised and scalable evaluation method that applies across real-world code domains without the need for costly human annotations. Unlike RTC, which tests semantic equivalence between code and its natural language description, and IdentityChain, which checks consistency across specification-code pairs, our closed-loop evaluation operates directly within program execution and enforces exact bijection rather than semantic equivalence.

Round-trip code evaluation. CodeIO (Li et al., 2025), CRUXEval (Gu et al., 2024) and its multilingual extension CRUXEval-X (Xu et al., 2025) probe bidirectional input–output reasoning over executable Python functions and score each direction independently. CodeMind (Liu et al., 2024) and ExeRScope (Liu and Jabbarvand, 2025) check reasoning ability about program execution and require models to track control flow, state changes, and dependencies across program steps. REVAL (Chen et al., 2024) and CACP (Hooda et al., 2024) analyse intermediate reasoning traces during execution to expose concept-level misunderstandings and inconsistencies in LLMs. Our framework unifies and

Benchmark	Fwd	Bwd	RT	Invert?	Exec?	Comparison Level
IdentityChain	✓	✓	Partial	×	No	Spec ↔ Code
RTC	✓	✓	Strict	×	No	Code ↔ NL descriptions
CodeIO	✓	✓	No	×	Yes	Function I/O
CRUXEval	✓	✓	No	×	Yes	Function exec I/O
CodeMind	✓	×	No	×	Yes	Task-level exec reasoning
ExeRScope	✓	×	No	×	Yes	Feature-level exec reasoning
REVAL	✓	×	No	×	Yes	Runtime behavior
CACP	✓	×	No	×	Yes	Concept-level traces
RTCE (ours)	✓	✓	Strict	✓	Yes	Exact enc/dec I/O bijection

Table 1: Comparison between RTCE and previously released benchmarks. Here, RT: Round-trip.

extends these directions under the notion of round-trip self-consistency, where a prediction is considered valid only if its logical inverse reconstructs the original input exactly. We do not rely on annotated traces or concept-level labelling; instead, by requiring predictions to be reconstructible through their logical inverse, our evaluation surfaces asymmetric reasoning failures that remain hidden under one-directional execution checks, trace-based diagnostics, or loosely paired evaluations.

Iterative self-correction. Several works have explored whether LLMs can improve their own outputs through iterative critique and revision (Madaan et al., 2023; Asai et al., 2024; Shinn et al., 2023; Gou et al., 2024). Olausson et al. (2024) show that self-repair for code generation offers only marginal gains without grounding in external execution feedback, a finding that directly aligns with the saturation we observe after the first revision round in RTCE. Unlike these works, which target code synthesis, we apply iterative self-reflection as a diagnostic to probe whether models can recover round-trip consistency, demonstrating that the failure mode in RTCE is not addressable through self-correction alone.

Table 1 shows the comparison table that highlights what RTCE uniquely measures compared to previous works.

4 Benchmark Construction

Unlike prevailing code-generation benchmarks that grade programs against hidden unit tests or online judges (Chen et al., 2021; Austin et al., 2021), we target exact algorithmic behaviour with a fully deterministic, execution-free evaluation. RTCE is built through a three-stage process: *generation*, where synthetic inputs are created across four source families (patterned strings, structured logs, YAML-like configs, tabular data) with multiple subcategories to ensure diversity and balance representative of real-world coding and data processing tasks; *validation*, where deterministic reference implementations of four compression algorithms, namely, Lempel–Ziv–Welch (LZW; Welch 1984), Arithmetic Encoding (AE; Rissanen 1976), Run-Length Encoding (RLE; Golomb 1966) and Huffman (Huffman, 1952), produce exact ground-truth outputs under a fixed seed; and *serialisation*, where validated input–output pairs are stored both as raw files by algorithm and as a unified JSON corpus containing descriptions, labels, and metadata. All random operations are governed by a fixed seed for reproducibility.

We select algorithms that span the four canonical design paradigms in lossless compression: dictionary, statistical interval coding, run aggregation, and prefix coding, while remaining simple enough to yield unique and deterministic ground truths. For LZW, the output is a sequence of integer codes representing substrings found during scanning. For AE, the output consists of a single floating-point value in the range $[0, 1)$, denoting the midpoint of the final coding interval and a frequency dictionary containing counts for all symbols, including a special EOF marker, necessary for exact decoding. For RLE, the output is a list of $(\text{symbol}, \text{count})$ pairs. For Huffman Coding, the output includes a list of integers representing byte-packed Huffman-encoded bitstrings and a metadata JSON file containing the symbol-to-bitstring codebook and the padding length needed for decoding. Collectively, these algorithms span a spectrum of input complexities and compression strategies, making them highly suitable for evaluating the capacity of code LLMs to learn and invert diverse algorithmic transformations. Compression tasks involve iterative symbol substitutions, dictionary management, probabilistic coding, and bitwise encoding, demanding multi-step algorithmic reason-

Data Source	# Categories	Key Characteristics
Patterned String Data	15	Repeated characters, alternating/block patterns, palindromes and near-palindromes, pangrams, keyboard sequences, pseudo-random token insertions, and short natural language sentences (single and repeated). Mix of highly compressible patterns and high-entropy noise.
Structured Log Data	11	Deterministic Apache-style log lines with timestamp, log level, HTTP method, endpoint, module, synthetic user ID, request duration, and IP. Categories include slow requests, status/metrics checks, authentication events, database errors, and severity levels. Metadata provided for analysis.
YAML-Like Config Data	10	Compact but realistic configuration snippets: application configs, Kubernetes deployments, Docker Compose, Helm values, Ansible playbooks, Prometheus configs, GitHub Actions, CircleCI, CloudFormation, Terraform. Preserves indentation, key–value syntax, and multi-document formats.
Tabular Data	10	CSV/TSV tables (5×5) with numeric, alphanumeric, mixed type, sparse, and repeated-header variants. Variations in delimiter, value entropy, and sparsity capture real-world spreadsheet/log regularity and redundancy.

Table 2: Summary of the four synthetic data source families used in RTCE, including category counts and defining characteristics.

ing and memory. Such algorithmic complexity exceeds the relatively arbitrary construction of many generic bijective functions.

RTCE includes 250 unique input samples, each assessed across four distinct code execution tasks, resulting in a comprehensive dataset of 1000 evaluation examples. Input lengths range from a few characters to several hundred, offering a diverse evaluation spectrum. We are in a controlled setting, and we do this on purpose to clearly measure whether LLMs can reliably perform code understanding and execution. RTCE uses synthetic data that deliberately mirrors common artefacts in real developer workflows, rather than arbitrary toy strings. Because compression algorithms are content-agnostic, they operate solely on sequences of characters/symbols and never inspect the input's syntax or semantics. For this reason, what matters is the distribution of character patterns, not whether the string came from a GitHub repository or a synthetic generator. Detailed category-level statistics are summarised in Table 2. The precise implementations of the encoding and decoding functions for each compression algorithm are provided in Appendix B.

5 Experimental Setup

We provide information about the models we use and our evaluation method.

5.1 Models

LLMs can be grouped by how they are trained and optimised, with each type having its strengths and weaknesses:

General instruction: Broadly trained LLMs using diverse web-scale corpora spanning natural language, factual knowledge, and limited code, but lack the inductive biases required for highly structured, algorithm-sensitive reasoning.

Code generation: LLMs fine-tuned or pre-trained on large-scale, high-quality programming datasets, enabling execution reliability across a wide range of code-related tasks.

Reasoning / Reasoning-distilled: Architectures explicitly optimised for multi-step analytical reasoning, frequently distilled from larger reasoning-focused systems capable of structured problem decomposition while maintaining inference efficiency.

5.2 Evaluation

We report three complementary metrics across all tasks and models.

Exact Match (EM) counts a prediction as correct only when it is value-equivalent to the reference: integers exactly, floats via `isclose` (tol $= 10^{-3}$), and lists/dicts recursively; strings are case-folded and trimmed before comparison. **Edit Similarity (ES)** is a normalised Levenshtein score that captures partial credit for structurally plausible but symbol-imprecise outputs. **Pass@5** generates n=5 completions per instance and scores an instance correct if at least one completion achieves EM, measuring a model's ceiling under sampling independently of single-shot reliability.

In our experiments, we answer three interdependent research questions, increasing the level of complexity of using the LLMs to solve RTCE:

RQ1

How well can LLMs invert code in a zero-shot setting, relying solely on their parametric knowledge without any task-specific prompting or training?

RQ2

Can iterative self-reflection, where the model critiques and revises its own outputs, recover LLM performance on code inversion beyond what zero-shot prompting achieves?

Through RQ1 and RQ2, we probe whether code invertibility emerges naturally in LLMs or can be surfaced through guided self-reflection. Our conclusion across both RQs is that code invertibility is a deeply challenging problem for current LLMs, one that neither scale nor self-reflection alone appears sufficient to resolve. This naturally raises the question of whether the capability can instead be instilled through targeted supervision and learning from examples.

RQ3

Can fine-tuning on execution traces give LLMs the ability to invert code, or do persistent failures suggest a reasoning limitation that standard training alone cannot overcome?

6 Results

We turn to explore the results for the three posed RQs.

6.1 RQ1: Zero-shot model performance

We evaluate 15 LLMs spanning four size tiers (1B–33B) across four task types per algorithm under zero-shot prompting. The tasks cover both directions of the encoder–decoder duality: **O/P Pred** ($x \xrightarrow{\mathsf{enc}} z$) applies the encoder directly; **O/P Pred-I** ($x \xrightarrow{\mathsf{dec}^{-1}} z$) provides the decoder and asks the model to invert it; **I/P Pred** ($z \xrightarrow{\mathsf{dec}} x'$) applies the decoder directly; **I/P Pred-I** ($z \xrightarrow{\mathsf{enc}^{-1}} x'$) provides the encoder and asks the model to invert it. All variants are strictly zero-shot, receiving one worked example in the preamble but no task-specific demonstrations. Prompt templates, input examples, and the inference pipeline are in Appendix F–C. Table 3 reports Pass@5 results across all 15 LLMs and four algorithms.

Scale and algorithm difficulty. Models $\leq$3.8B score near zero across all algorithms, lacking the capacity for multi-step bookkeeping. The 7–9B range generalises on RLE (locally repetitive patterns) but fails on AE and LZW (long-span interval accumulation and dictionary management). Above 14B, RLE and LZW improve substantially, yet Huffman encoding remains unsolved for all 15 models. Reasoning-distilled models (e.g., DeepSeek-R1-14B; Guo et al. 2025) consistently outperform general-instruction counterparts of comparable size, confirming that chain-of-thought pretraining helps with structured multi-step simulation.

Encoding–decoding asymmetry. Encoding (O/P Pred) requires tracking the evolving state across every symbol (interval bounds for AE, a growing dictionary for LZW, run-length counters for RLE), so even a single bookkeeping error can collapse exact matches. Decoding (I/P Pred) allows the model to exploit surface regularities in the encoded string, and I/P Pred Pass@5 exceeds O/P Pred for most pairs. AE is the exception: QwQ-32B reaches 27.6% on AE encoding but only 2.3% on AE decoding (a 12$\times$ collapse), because decoding requires inverse floating-point interval arithmetic that is more numerically fragile than the forward pass.

Inverse variants and the Huffman paradox. On RLE and AE, QwQ-32B (Yang et al., 2024) scores higher on O/P Pred-I (66.8 and 41.6 Pass@5) than O/P Pred (57.6 and 27.6), because the provided decoder function is simpler to invert than the encoder is to simulate directly. This advantage vanishes for LZW and Huffman. More strikingly, all 15 models score 0% on Huffman encoding (O/P Pred, O/P Pred-I) while QwQ-32B reaches 7.9% on Huffman decoding (I/P Pred) and 11.1% on I/P Pred-I. Decoding is a tree traversal over an explicitly provided tree; encoding requires constructing the frequency table, building the Huffman tree, and emitting variable-length codes: a multi-stage hierarchical procedure no model handles correctly. Finally, Edit Similarity remains above zero even at 0% exact match (e.g., Mistral-7B on AE: ES $\approx 8\%$, P@5 $= 0\%$), showing that models produce plausible but imprecise outputs, and that RTCE demands exact symbol-level fidelity.

Tokenization is not the bottleneck. Tokenization is not the root cause of these failures. First, Llama and the Qwen family use fundamentally

Model	Size (B)	Alg.	O/P Pred ($x \xrightarrow{enc} z$)			O/P Pred-I ($x \xrightarrow{dec^{-1}} z$)			I/P Pred ($z \xrightarrow{dec} x'$)			I/P Pred-I ($z \xrightarrow{enc^{-1}} x'$)			Average
			EM	ES	P@5	EM	ES	P@5	EM	ES	P@5	EM	ES	P@5	
Llama-3.2-1B-Instruct	1.0	AE	0.00	0.69	0.00	0.00	1.62	0.00	0.00	0.16	0.00	0.00	1.58	0.00	0.34
		LZW	0.00	0.03	0.00	0.00	0.08	0.00	0.00	0.11	0.00	0.00	0.35	0.00	0.05
		RLE	0.00	0.20	0.00	0.00	0.05	0.00	0.00	0.81	0.00	0.00	0.72	0.00	0.15
		HUFF	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.27	0.00	0.00	0.68	0.00	0.08
DeepSeek_R1_Distill_Qwen_1.5B	1.5	AE	0.72	15.00	2.00	0.00	1.00	0.00	0.00	2.71	0.00	0.00	0.53	0.00	1.83
		LZW	0.00	1.12	0.00	0.00	0.33	0.00	0.00	2.01	0.00	0.00	0.38	0.00	0.32
		RLE	2.80	12.32	7.20	0.40	0.81	2.00	1.60	7.44	3.60	1.84	6.78	4.80	4.30
		HUFF	0.00	0.62	0.00	0.00	0.01	0.00	0.16	5.78	0.40	0.00	0.65	0.00	0.64
Phi_3_mini_128k_instruct	3.8	AE	0.00	6.85	0.00	0.00	0.91	0.00	0.08	15.38	0.40	0.00	7.55	0.00	2.60
		LZW	0.00	9.48	0.00	0.08	8.66	0.40	0.24	12.83	1.20	0.00	10.87	0.00	3.65
		RLE	1.84	41.86	2.80	0.40	18.21	2.00	3.76	31.40	7.20	2.64	26.43	5.60	12.01
		HUFF	0.00	0.66	0.00	0.00	0.93	0.00	0.00	10.04	0.00	0.00	6.88	0.00	1.54
Phi_3.5_mini_instruct	3.8	AE	0.56	24.00	1.20	0.00	1.07	0.00	0.00	5.55	0.00	0.00	1.78	0.00	2.85
		LZW	0.00	16.41	0.00	0.00	7.54	0.00	0.00	15.94	0.00	0.08	17.38	0.40	4.81
		RLE	1.84	53.66	4.00	0.00	0.41	0.00	4.00	45.72	7.20	3.52	37.27	6.40	13.67
		HUFF	0.00	0.51	0.00	0.08	0.26	0.40	0.08	11.29	0.40	0.00	7.37	0.00	1.70
Mistral_7B_Instruct_v0.3	7.2	AE	0.00	7.92	0.00	0.00	7.99	0.00	0.00	7.79	0.00	0.00	0.96	0.00	2.05
		LZW	0.00	9.85	0.00	0.00	5.68	0.00	0.00	9.29	0.00	0.00	2.14	0.00	2.25
		RLE	0.64	44.06	2.00	1.60	32.36	3.60	0.56	35.20	1.20	0.16	16.02	0.80	11.52
		HUFF	0.00	0.62	0.00	0.00	0.39	0.00	0.00	9.22	0.00	0.00	1.62	0.00	0.99
Qwen2.5_7B_Instruct	7.6	AE	0.72	30.68	1.20	0.56	28.27	0.80	0.00	10.56	0.00	0.08	5.34	0.40	6.55
		LZW	0.00	6.31	0.00	0.16	9.98	0.80	0.16	16.36	0.40	0.40	18.20	0.80	4.46
		RLE	2.96	34.74	5.20	2.88	31.51	5.60	4.96	51.88	6.00	3.76	52.81	6.40	17.39
		HUFF	0.00	0.07	0.00	0.00	0.46	0.00	0.00	15.59	0.00	0.16	14.77	0.80	2.65
Llama_3.1_8B_Instruct	8.0	AE	0.72	15.71	0.80	0.40	19.02	0.80	0.32	7.84	0.80	0.08	6.47	0.40	4.45
		LZW	0.00	3.42	0.00	0.00	1.60	0.00	0.00	3.64	0.00	0.00	1.21	0.00	0.82
		RLE	2.08	26.02	4.40	1.60	12.52	3.60	2.72	29.95	5.20	0.24	23.23	0.80	9.36
		HUFF	0.00	0.00	0.00	0.00	0.16	0.00	0.00	10.22	0.00	0.00	3.18	0.00	1.13
codegemma_7b_it	8.5	AE	0.16	13.67	0.80	0.08	6.37	0.40	0.00	5.81	0.00	0.08	2.92	0.40	2.56
		LZW	0.16	41.71	0.80	0.16	11.40	0.80	0.00	9.12	0.00	0.00	3.98	0.00	5.68
		RLE	1.68	54.53	3.20	0.24	5.93	1.20	0.40	14.54	1.20	0.48	3.22	1.60	7.35
		HUFF	0.00	0.98	0.00	0.00	1.11	0.00	0.00	8.97	0.00	0.00	2.50	0.00	1.13
Yi_Coder_9B_Chat	8.8	AE	0.08	6.24	0.40	0.00	0.84	0.00	0.00	11.86	0.00	0.00	7.04	0.00	2.20
		LZW	1.20	14.35	2.40	0.24	1.64	1.20	0.40	15.51	1.20	0.32	10.87	1.20	4.21
		RLE	10.56	37.44	14.00	2.40	5.02	9.20	5.76	32.89	8.00	0.80	13.33	2.80	11.85
		HUFF	0.00	0.13	0.00	0.00	0.00	0.00	0.00	10.64	0.00	0.00	11.63	0.00	1.87
DeepSeek_R1_Distill_Qwen_14B	14.8	AE	4.88	32.12	11.20	5.20	31.05	13.20	0.40	9.99	1.20	0.64	9.92	1.20	10.08
		LZW	8.24	28.78	14.00	7.52	32.60	13.60	2.24	22.38	5.60	2.56	24.40	6.40	14.03
		RLE	16.16	63.25	26.00	16.88	56.03	28.00	8.08	47.25	13.20	4.56	27.39	16.80	26.97
		HUFF	0.00	0.06	0.00	0.00	0.45	0.00	0.68	15.59	1.51	1.13	15.69	2.63	3.15
Codestral_22B_v0.1	22.2	AE	0.00	2.20	0.00	0.00	0.80	0.00	0.00	11.33	0.00	0.00	6.78	0.00	1.76
		LZW	0.88	23.35	1.20	0.64	23.94	1.20	1.36	22.52	2.40	0.32	14.23	1.20	7.77
		RLE	9.92	69.62	13.60	8.32	70.80	13.20	11.60	71.34	20.40	11.04	47.49	20.80	30.68
		HUFF	0.00	0.08	0.00	0.00	0.10	0.00	0.00	6.26	0.00	0.00	11.57	0.00	1.50
QwQ_32B	32.8	AE	12.64	23.06	27.59	18.27	33.81	41.62	1.85	11.01	2.31	2.20	11.32	2.89	15.71
		LZW	15.52	36.67	18.03	16.15	44.62	18.68	9.18	45.08	12.57	9.34	48.46	15.38	24.14
		RLE	32.00	80.23	57.60	36.80	78.62	66.80	26.08	79.49	53.20	34.96	79.76	61.20	57.23
		HUFF	0.00	0.00	0.00	0.00	0.22	0.00	4.34	18.25	7.89	4.71	19.52	11.11	5.50
Qwen2.5_Coder_32B_Instruct	32.8	AE	0.24	34.03	0.80	0.24	37.51	1.20	0.24	15.65	1.20	0.08	9.77	0.40	8.45
		LZW	4.56	63.76	7.60	2.88	66.75	7.20	2.16	46.34	3.60	2.08	41.79	4.00	21.06
		RLE	20.00	83.93	26.40	18.64	79.84	28.40	23.84	77.72	41.60	10.56	53.97	33.20	41.51
		HUFF	0.00	0.00	0.00	0.00	0.41	0.00	1.38	19.31	3.85	0.46	11.57	0.76	3.15
DeepSeek_R1_Distill_Qwen_32B	32.8	AE	9.36	29.77	21.05	12.71	21.55	30.59	1.40	11.20	2.34	1.18	8.16	3.53	12.74
		LZW	12.50	64.39	17.61	13.14	50.77	20.57	5.23	35.09	10.23	6.29	36.23	13.71	23.81
		RLE	23.39	74.52	39.13	19.56	42.14	44.10	15.28	63.93	33.62	11.79	29.62	39.30	36.37
		HUFF	0.00	0.00	0.00	0.00	0.09	0.00	2.81	16.80	7.41	2.84	11.88	5.97	3.98
deepseek_coder_33b_instruct	33.3	AE	0.48	12.00	0.80	0.00	1.55	0.00	0.24	14.87	0.80	0.00	9.29	0.00	3.34
		LZW	0.08	2.86	0.40	0.08	3.52	0.40	0.48	18.85	1.20	0.24	12.40	0.80	3.44
		RLE	7.20	52.72	9.60	0.24	3.79	0.80	4.80	54.46	6.80	1.12	18.62	4.40	13.71
		HUFF	0.00	0.10	0.00	0.00	0.00	0.00	0.00	10.88	0.00	0.00	3.56	0.00	1.21

Table 3: Unified comparison across models present in all four algorithm tables (AE, LZW, RLE, HUFFMAN). The colour-coded "Average" column (red = low, green = high) aggregates all the EM, ES, and Pass@5 (P@5) metrics, enabling quick visual comparison of model performance, strengths, and trends. Across all settings, results show a consistent hierarchy: O/P Pred > O/P Pred-I > I/P Pred > I/P Pred-I. O/P Pred scores highest, and it seems that forward encoding an input string from compression is easier for most LLMs. I/P Pred-I and O/P Pred-I are the most challenging, as they combine strict input and output demands with inverse decoding, adding complexity and compounding errors.

different tokenization regimes (e.g., Llama fuses roughly 3 binary digits per token), yet accuracy remains consistently low across both. This rules out a specific tokenizer's behavior as the explanation. Second, QwQ and Qwen2.5-Coder (both 32B) share the exact same parameter count and tokenizer. Despite this identical representation, QwQ achieves a 15.71 Avg. AE score compared to Qwen2.5-Coder's 8.45. This 1.86× difference is attributable to reasoning-focused training, suggesting the bottleneck is a lack of logical reasoning rather than tokenization.

6.2 RQ2: Multi-turn revision

We apply a multi-turn self-reflection approach, in the spirit of (Madaan et al., 2023; Asai et al., 2024), that refines model answers through structured cycles of critique and revision without regenerating from scratch. Starting from the zero-shot prediction as the initial draft, we run two revision rounds, each comprising two phases. In the **critique phase**, the model reviews its own current draft and produces a structured assessment: a list of correctness and formatting findings, a suggested fix plan, and an explicit `VERDICT: KEEP` or `VERDICT: REVISE` decision. In the **revision phase**, triggered only when the verdict is `REVISE`, a separate editor role rewrites the draft strictly according to the critique feedback, without access to the ground truth answer. Before each scoring step, answers are standardised into a canonical JSON format `{"output": <value>}` to decouple formatting errors from genuine reasoning failures. To ensure all prompts fit within the model's context window, the system estimates token usage and applies context budgeting, trimming inputs and adjusting generation lengths as needed. Iteration terminates early as soon as an exact match with the ground truth is produced. If no exact match is achieved after all rounds, fallback policies are applied: the final draft is retained as is, optionally annotated to indicate the lack of confirmed correctness, or replaced with a neutral placeholder that signals prediction uncertainty without revealing the ground truth. We show the exact working configuration in Appendix E.

Figure 3 shows performance across revision rounds on AE. The first revision round yields measurable gains, particularly on inversion tasks ($z \xrightarrow{\text{enc}^{-1}} x'$, $x \xrightarrow{\text{dec}^{-1}} z$), which suffer disproportionately from shallow reasoning errors that a structured critique can surface. However, subsequent rounds exhibit sharply diminishing returns, with performance plateauing well below the ceiling. This rapid saturation reveals a fundamental limitation of unaided self-critique: while it can recover from minor formatting and surface-level reasoning errors, it cannot resolve deeper failures such as systematic state-tracking errors or incorrect inversion logic. These require external execution feedback or stronger compositional reasoning abilities (Ni et al., 2025; Kohli et al., 2025), neither of which self-reflection alone provides.

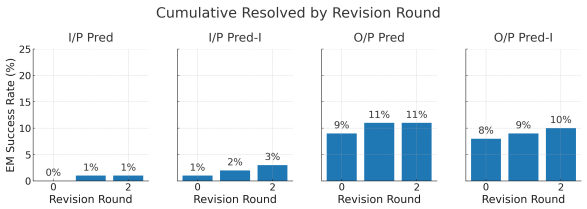


Figure 3: Multi-turn revision for AE on a subset of the dataset. Overall, tasks requiring inversion show lower initial accuracy but larger gains with revision across the two rounds.

6.3 RQ3: Fine-tuning

Our SFT pipeline proceeds in five stages: execution trace generation, trace filtering, natural language translation, supervised data construction, and LoRA fine-tuning.

Trace generation. We instrument each reference solution by injecting a `@snoop` decorator at a depth equal to the number of helper functions defined in the file, enabling fine-grained step-by-step variable tracing. The instrumented script is executed in an isolated subprocess (new process group) with a strict five-second wall-clock timeout; upon expiry, the process tree is terminated via `SIGTERM`, followed by `SIGKILL`. Execution traces are captured from `stderr` and must terminate with the sentinel line `<< Return value from main_solution: <value>`, which simultaneously verifies successful completion and records the ground-truth output.

Trace filtering. A trace is accepted only if it (i) contains no runtime error markers, (ii) ends with the sentinel line as its final entry, and (iii) does not exceed 3,000 lines. Applied to the 250-input RTCE pool, this yields 87–98 validated examples per algorithm (87 for RLE, 98 for AE), each associating a problem description and input with a clean, deterministic execution trace.

Natural language translation. Validated traces are translated into step-by-step natural-language reasoning using Qwen3-32B, deployed offline via vLLM with sampling parameters: temperature 0.6, top-p 0.95, top-k 20, and a maximum of 8,192 output tokens. Memory addresses are stripped from traces before prompting. The model is instructed to reproduce the exact intermediate values visible in the trace, while writing as if reasoning through the code independently, without citing the trace as a source. The ground-truth answer is parsed from the sentinel line to serve as the evaluation label. Prompts exceeding 32,768 characters are discarded prior to generation.

SFT data construction. Each translated example is cast as a two-turn chat instance aligned with the QwQ-32B instruction template: the user message contains the problem description, reference solution code, and input; the assistant message contains the natural-language reasoning chain. Sequences that exceed 32,768 characters after applying the chat template are filtered out, yielding a compact, high-quality supervised fine-tuning dataset.

LoRA fine-tuning. We fine-tune QwQ-32B using Low-Rank Adaptation (LoRA; Hu et al. 2022) with DeepSpeed ZeRO-3 across four GPUs. Rank-8 adapters are applied to all linear modules (`lora_target: all`). Training runs for three epochs with a per-device batch size of 1 and gradient accumulation of 8 (effective batch size 32), a peak learning rate of 1×10^{-4} under a cosine schedule with 10% linear warmup, a sequence cutoff of 2,048 tokens, and BF16 mixed precision with SDPA flash attention (Dao et al., 2022). One adapter checkpoint is produced per algorithm. Full hyperparameter details are given in Table 5 (Appendix D).

Table 4 reports fine-tuning results for QwQ-32B, the strongest zero-shot model. SFT substantially lifts forward task performance on RLE (80.0%), LZW (87.5%), and AE (78.6%), yet inverse task gains are uneven. RLE holds up well (76.5%), while AE collapses to 30.8% and Huffman encoding remains at 0% despite decoder improvement (35.0%). This asymmetry reveals that exposure to correct forward-execution steps is not sufficient: the adapters overfit to the surface form of trace-derived reasoning chains without internalising the bijective state-transition structure required for inversion.

This confirms our central diagnostic argument: We hypothesise the bottleneck is not exposure to correct reasoning steps but the model's inability to generalise the algorithm's bijective invariants to unseen inputs. Likewise, the rapid saturation of self-reflection after the first revision round (RQ2) indicates that the model lacks the capacity for deep semantic repair rather than merely surface-level error correction. Together, both results converge on the same conclusion: current LLMs fail on RTCE because they cannot maintain and invert evolving algorithmic state (Dziri et al., 2023), a limitation that neither trace-supervised fine-tuning nor iterative self-critique can overcome.

algo	temp	I/P Pred	I/P Pred-I	O/P Pred	O/P Pred-I
ae	0.2	30.77	23.08	78.57	84.62
	0.8	15.00	20.00	70.00	84.21
huffman	0.2	35.00	50.00	0.00	0.00
	0.8	36.36	50.00	0.00	0.00
lzw	0.2	62.50	62.50	87.50	87.50
	0.8	77.42	75.00	78.13	80.65
rle	0.2	76.47	86.00	80.00	86.00
	0.8	80.65	86.89	80.33	86.89

Table 4: Pass@5 after fine-tuning Qwen/QwQ-32B with LoRA across two different temperature settings. Taken together, models get better at one direction or the inversion variant, yet they still fail to maintain high accuracy simultaneously across all four tasks.

7 Conclusion

We introduce RTCE to assess round-trip consistency in Code-LLMs, highlighting a new test for model robustness. Achieving round-trip bijection is a stringent objective, requiring to internalise deep semantic and structural code properties, including variable dependencies, control flow, type constraints, and program invariants. When tasked with both encoding (forward execution) and decoding (inverse reconstruction), LLMs consistently display systematic yet non-obvious inconsistencies. These manifest as information loss, the introduction of hallucinated details, or violations of semantic constraints, preventing exact input recovery and expose brittle, non-compositional reasoning. LLMs, even when assisted with advanced prompts, fine-tuning, or self-reflection, do not fully demonstrate the internal semantic coherence required for trustworthy, compositional code reasoning. This opens a rich landscape for future work that could span new architectures, training objectives, and evaluation methodologies aimed at building models capable of reliable code reasoning.

Limitations

In our analysis, we predominantly use open-source LLMs, as they offer advantages such as ease of fine-tuning, self-reflection, and iterative reasoning capabilities, which are often limited or inaccessible in closed-source models. Additionally, the design of RTCE centres on the compression-decompression paradigm, which may overlook other forms of invertibility, such as refactoring, decompilation, or symbolic manipulation, that are equally important for code inversion. Within the RTCE framework, the current implementation focuses exclusively on Python, which may limit the generalisability of insights into model capabilities across other programming languages, though extending to other languages is straightforward. While 1,000 exact-match instances are sufficient to expose fundamental inversion failures, the small pool limits fine-grained per-category statistical reliability and may underestimate the variance in difficulty across input distributions. Finally, the evaluation framework is static and execution-free, thus unable to assess how models handle runtime phenomena such as side effects, concurrency, exceptions, or external dependencies, which are common challenges in code correctness and invertibility.

Ethical Considerations

RTCE is built from synthetic inputs and deterministic reference implementations; it involves no human subjects, crowdsourced annotations, or personally identifiable data. We evaluate only publicly released models, making our results fully reproducible without access to proprietary APIs. RTCE probes a specific reasoning ability: whether LLMs can faithfully execute and invert lossless compression algorithms. This is a diagnostic tool for understanding model limitations, not a system that produces harmful outputs or enables misuse. Large-scale inference and fine-tuning carry a computational cost, which we mitigated by batching requests and reusing completed inference files. The dataset, code, and model outputs are released under permissive licences to support reproducibility.

Acknowledgements

We thank the reviewers and the area chair for their useful comments. NM was supported by the UKRI Centre for Doctoral Training (CDT) in Natural Language Processing through the UKRI grant (EP/S022481/1). The authors acknowledge the use of resources provided by the Isambard-AI National AI Research Resource (AIRR; McIntosh-Smith et al. 2024). Isambard-AI is operated by the University of Bristol and is funded by the UK Government's Department for Science, Innovation and Technology (DSIT) via UK Research and Innovation; and the Science and Technology Facilities Council [ST/AIRR/I-A-I/1023]. AV was supported by the "UNREAL: Unified Reasoning Layer for Trustworthy ML" project (EP/Y023838/1) selected by the ERC and funded by UKRI EPSRC.

References

Miltiadis Allamanis, Sheena Panthaplackel, and Pengcheng Yin. 2024. Unsupervised evaluation of code LLMs with round-trip correctness. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 1050–1066. PMLR.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2024. Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *ArXiv preprint*, abs/2108.07732.

Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2024. Reasoning runtime behavior of a program with LLM: How far are we? *ArXiv preprint*, abs/2403.16437.

Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2025. Reasoning runtime behavior of a program with LLM: How far are we? In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1869–1881, Los Alamitos, CA, USA. IEEE Computer Society.

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.

Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Re. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter

West, Chandra Bhagavatula, Ronan Le Bras, Jena D. Hwang, Soumya Sanyal, Xiang Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. 2023. Faith and fate: Limits of transformers on compositionality. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Solomon W. Golomb. 1966. Run-length encodings. *IEEE Transactions on Information Theory*, 12(3):399–401.

Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2024. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations*.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. CRUXEval: A benchmark for code reasoning, understanding and execution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 16568–16621. PMLR.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and 1 others. 2025. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638.

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. *ArXiv preprint*, abs/2401.14196.

Yancheng He, Shilong Li, Jiaheng Liu, Weixun Wang, Xingyuan Bu, Ge Zhang, Z.y. Peng, Zhaoxiang Zhang, Zhicheng Zheng, Wenbo Su, and Bo Zheng. 2025. Can large language models detect errors in long chain-of-thought reasoning? In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 18468–18489, Vienna, Austria. Association for Computational Linguistics.

Ashish Hooda, Mihai Christodorescu, Miltiadis Allamanis, Aaron Wilson, Kassem Fawaz, and Somesh Jha. 2024. Do large code models understand programming concepts? counterfactual analysis for code predicates. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

David A. Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-Coder technical report. *ArXiv preprint*, abs/2409.12186.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*.

Weisen Jiang, Han Shi, Longhui Yu, Zhengying Liu, Yu Zhang, Zhenguo Li, and James Kwok. 2024. Forward-backward reasoning in large language models for mathematical verification. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 6647–6661, Bangkok, Thailand. Association for Computational Linguistics.

Harsh Kohli, Sachin Kumar, and Huan Sun. 2025. GroundCocoa: A benchmark for evaluating compositional & conditional reasoning in language models. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 8280–8295, Albuquerque, New Mexico. Association for Computational Linguistics.

Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. 2025. CodeI/O: Condensing reasoning patterns via code input-output prediction. *ArXiv preprint*, abs/2502.07316.

Changshu Liu, Yang Chen, and Reyhaneh Jabbarvand. 2025. Assessing coherency and consistency of code execution reasoning by large language models. *ArXiv preprint*, abs/2510.15079.

Changshu Liu and Reyhaneh Jabbarvand. 2025. A tool for in-depth analysis of code execution reasoning of large language models. *ArXiv preprint*, abs/2501.18482.

Changshu Liu, Shizhuo Dylan Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. 2024. CodeMind: A framework to challenge large language models for code reasoning. *ArXiv preprint*, abs/2402.09664.

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. *ArXiv preprint*, abs/2402.19173.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon,

Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. *Preprint*, arXiv:2303.17651.

Simon McIntosh-Smith, Sadaf R Alam, and Christopher Woods. 2024. Isambard-AI: a leadership class supercomputer optimised specifically for Artificial Intelligence. *ArXiv preprint*, abs/2410.11199.

Marcus J. Min, Yangruibo Ding, Luca Buratti, Saurabh Pujar, Gail Kaiser, Suman Jana, and Baishakhi Ray. 2024. Beyond accuracy: Evaluating self-consistency of code large language models with IdentityChain. In *The Twelfth International Conference on Learning Representations*.

Ruikang Ni, Da Xiao, Qingye Meng, Xiangyu Li, Shihui Zheng, and Hongliang Liang. 2025. Benchmarking and understanding compositional relational reasoning of LLMs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(18):19703–19711.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations (ICLR)*.

J. J. Rissanen. 1976. Generalized kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Preprint*, arXiv:2303.11366.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*.

Zhepeng Wang, Runxue Bao, Yawen Wu, Jackson Taylor, Cao Xiao, Feng Zheng, Weiwen Jiang, Shangqian Gao, and Yanfu Zhang. 2024. Unlocking memorization in large language models with dynamic soft prompting. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 9782–9796, Miami, Florida, USA. Association for Computational Linguistics.

Welch. 1984. A technique for high-performance data compression. *Computer*, 17(6):8–19.

Ruiyang Xu, Jialun Cao, Yaojie Lu, Ming Wen, Hongyu Lin, Xianpei Han, Ben He, Shing-Chi Cheung, and Le Sun. 2025. CRUXEVAL-X: A benchmark for multilingual code reasoning, understanding and execution. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 23762–23779, Vienna, Austria. Association for Computational Linguistics.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, and 23 others. 2024. Qwen2.5 technical report. *ArXiv preprint*, abs/2412.15115.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. OpenCodeInterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 12834–12859, Bangkok, Thailand. Association for Computational Linguistics.

A Concrete task example

Figure 4 illustrates the four prediction tasks applied to a single RLE input, showing both the forward (encoding) and backward (decoding with inversion) directions through the compression pipeline.

Concrete Task Example

Consider a simple compression transformation using run-length encoding (RLE), where repeated characters are encoded as character-count pairs:

$$\mathsf{enc}(x) = \texttt{RLE\_compress}(x),$$

$$\mathsf{dec}(z) = \texttt{RLE\_decompress}(z).$$

Given the input string $x =$ "aaabbbcc" (8 characters), the encoder produces:

$$z = \mathsf{enc}(x) = \texttt{"a3b3c2"} \quad \text{(6 characters)}.$$

The four evaluation settings are instantiated as follows:

- **Output Prediction (forward).** Given (x, enc), the LLM simulates the encoder and predicts:

$$\hat{z} = \mathsf{enc}(x) = \texttt{"a3b3c2"}.$$

- **Input Prediction with Inversion.** Given (z, enc), the LLM must mentally invert enc to act as the decoder and predict:

$$\hat{x}' = \mathsf{enc}^{-1}(z) = \mathsf{dec}(z) = \texttt{"aaabbbcc"}.$$

- **Output Prediction with Inversion.** Given (x, dec), the LLM must mentally invert dec to act as the encoder and predict:

$$\hat{z} = \mathsf{dec}^{-1}(x) = \mathsf{enc}(x) = \texttt{"a3b3c2"}.$$

- **Input Prediction (forward).** Given (z, dec), the LLM simulates the decoder and predicts:

$$\hat{x}' = \mathsf{dec}(z) = \texttt{"aaabbbcc"}.$$

In the two *with inversion* tasks (2 and 3), the LLM must mentally invert the provided function to solve the task correctly, even though the inverse function is never explicitly given. This tests whether the model understands the bidirectional relationship between encoding and decoding.

Figure 4: A concrete task example outlining the workflow across the four prediction tasks using run-length encoding.

B Compression algorithms

Listing 1: LZW encoding reference.

```
def main_solution(uncompressed):
    dict_size = 256
    dictionary = {chr(i): i for i in range(
     dict_size)}
    w = ""
    result = []
    for c in uncompressed:
        wc = w + c
        if wc in dictionary:
            w = wc
        else:
            result.append(dictionary[w])
            dictionary[wc] = dict_size
            dict_size += 1
            w = c
    if w:
        result.append(dictionary[w])
    return result
```

Listing 2: LZW decoding reference.

```
def main_solution(compressed):
    dict_size = 256
    dictionary = {i: chr(i) for i in range(
     dict_size)}
    result = []
    w = chr(compressed.pop(0))
    result.append(w)
    for k in compressed:
        if k in dictionary:
            entry = dictionary[k]
        elif k == dict_size:
            entry = w + w[0]
        else:
            raise ValueError("Bad compressed k:
     %s" % k)
        result.append(entry)
        dictionary[dict_size] = w + entry[0]
        dict_size += 1
        w = entry
    return "".join(result)
```

Listing 3: AE encoding reference (using per-item freq).

```
def main_solution(uncompressed):
    freq = FREQ_DICT  # injected per item,
     includes 'EOF'
    total = sum(freq.values())
    symbols = sorted(freq.keys())
    cum_counts, running = {}, 0
    for sym in symbols:
        cum_counts[sym] = running
        running += freq[sym]
    low, high = 0.0, 1.0
    for c in list(uncompressed) + ['EOF']:
        width = high - low
        high = low + width * (cum_counts[c] +
     freq[c]) / total
        low  = low + width *  cum_counts[c] /
     total
    return (low + high) / 2
```

Listing 4: AE decoding reference (using per-item freq).

```
def main_solution(compressed):
    freq = FREQ_DICT  # injected per item,
     includes 'EOF'
    total = sum(freq.values())
    symbols = sorted(freq.keys())
    cum_counts, running = {}, 0
    for s in symbols:
        cum_counts[s] = running
        running += freq[s]
    low, high = 0.0, 1.0
    result = []
    while True:
```

```
        width = high - low
        scaled = (compressed - low) / width *
    total
        for s in symbols:
            if cum_counts[s] <= scaled <
    cum_counts[s] + freq[s]:
                symbol = s
                break
        if symbol == 'EOF':
            break
        result.append(symbol)
        high = low + width * (cum_counts[symbol]
     + freq[symbol]) / total
        low  = low + width *  cum_counts[symbol]
     / total
    return ''.join(result)
```

Listing 5: RLE encoding reference.

```
def main_solution(uncompressed):
    if not uncompressed:
        return []
    result = []
    prev_char = uncompressed[0]
    count = 1
    for c in uncompressed[1:]:
        if c == prev_char:
            count += 1
        else:
            result.append((prev_char, count))
            prev_char = c
            count = 1
    result.append((prev_char, count))
    return result
```

Listing 6: RLE decoding reference.

```
def main_solution(compressed):
    result = []
    for char, count in compressed:
        result.append(char * count)
    return ''.join(result)
```

Listing 7: Huffman encoding reference.

```
def main_solution(uncompressed):
    from collections import namedtuple
    import heapq
    freq = FREQ_DICT  # injected per item
    Node = namedtuple('Node', ['freq','symbol','
    left','right'])
    Node.__lt__ = lambda a,b: a.freq < b.freq
    heap = [Node(f, sym, None, None) for sym, f
     in freq.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        l = heapq.heappop(heap); r = heapq.
    heappop(heap)
        heapq.heappush(heap, Node(l.freq + r.
    freq, None, l, r))
    codebook = {}
    def walk(node, prefix):
        if node.symbol is not None:
            codebook[node.symbol] = prefix or '0
    '
        else:
            walk(node.left, prefix + '0')
            walk(node.right, prefix + '1')
    walk(heap[0], '')
    bitstr = ''.join(codebook[c] for c in
    uncompressed)
    padding = (-len(bitstr)) % 8
    bitstr += '0' * padding
    encoded_bytes = [int(bitstr[i:i+8], 2) for i
     in range(0, len(bitstr), 8)]
    return encoded_bytes, codebook, padding
```

Listing 8: Huffman decoding reference.

```
def main_solution(compressed):
    from collections import namedtuple
    import heapq
    encoded_bytes, codebook, padding =
    compressed
    freq = FREQ_DICT  # same dict used to
    rebuild the tree
    Node = namedtuple('Node', ['freq','symbol','
    left','right'])
    Node.__lt__ = lambda a,b: a.freq < b.freq
    heap = [Node(f, sym, None, None) for sym, f
    in freq.items()]
    heapq.heapify(heap)
    while len(heap) > 1:
        l = heapq.heappop(heap); r = heapq.
    heappop(heap)
        heapq.heappush(heap, Node(l.freq + r.
    freq, None, l, r))
    root = heap[0]
    bitstr = ''.join(f'{b:08b}' for b in
    encoded_bytes)
    if padding:
        bitstr = bitstr[:-padding]
    result, node = [], root
    for bit in bitstr:
        node = node.left if bit == '0' else node.
    right
        if node.symbol is not None:
            result.append(node.symbol)
            node = root
    return ''.join(result)
```

C Model inference details

All experiments employed the `vLLM` inference engine. Decoding was performed with temperatures $\{0.2, 0.8\}$ and nucleus sampling ($p = 0.95$), producing up to 16,384 tokens per completion. Unless specified otherwise, each prompt generated $n = 5$ completions. Models were initially loaded in `bfloat16` precision with the configured tensor parallelism (`tp_size`); on GPU memory exhaustion, inference fell back to 4-bit NF4 quantisation via `bitsandbytes` with double quantisation and `float16` compute. A fixed random seed (42) ensured reproducibility during both data curation and model inference. The stop sequence `[/ANSWER]` was enforced with inclusion in the output. For robustness, execution time was logged, and each request was retried up to two times upon failure with a 5 s backoff. The outputs were stored in JSONL format, containing model metadata (name, size, category), inference parameters, and generated content.

Our experiments were conducted using four

NVIDIA GH200 GPUs, each featuring 120GB of memory. For each model and temperature setting, we estimate an average inference time of approximately 14 hours to produce predictions of 5000 input samples.

D Fine-tuning hyperparameters

Table 5 shows the configured values for the set of hyperparameters used in the fine-tuning experiments.

Hyperparameter	Value
Base model	Qwen/QwQ-32B
Fine-tuning type	LoRA
LoRA rank	8
LoRA target	all
Stage	SFT
Epochs	3.0
Learning rate	1×10^{-4}
Batch size (per device)	1
Gradient accumulation	8
Cutoff length	2048
Precision	FP16

Table 5: Key fine-tuning hyperparameters for Qwen/QwQ-32B with LoRA.

E Multi-turn revision configurations

Environment and hardware. Experiments were run under SLURM using a Singularity container `e4s-cuda90-aarch64-25.06.sif`. We used Python 3.10 in a Conda environment created with `requirements.txt`. Jobs requested 4 GPUs, 32 CPUs, and 128 GB RAM, though each run was launched with `-num_gpus 1`. Offline mode was enabled (`-hf_offline`), with the Hugging Face cache stored at `~/.cache/huggingface/hub`.

Models. All experiments used the `Qwen/QwQ-32B` model. We applied RoPE scaling with the following JSON configuration:

```
{"rope_type":"yarn","factor":16.0,
 "original_max_position_embeddings":8192}
```

Data. Algorithms evaluated were: `ae`, `lzw`, `rle`, and `huffman`. For each, the input file was of the form:

```
processed_datasets_test/${ALGO}/
  ${MODEL}_temp_${TEMP}_n5_verified.jsonl
```

The ground truth field was `res.actual`. Temperature was parsed from the filename (e.g. `_temp_0.2_` $\rightarrow$ 0.2).

Core settings. We used 2 reflection rounds (`-reflection_rounds 2`), critique style B (`-critique_style B`), and early stopping on EM (`-gt_stop_on em`). The mismatch policy was `-on_mismatch annotate`, which keeps the draft and attaches a status note (no ground truth leak). Additional context and generation settings were:

```
--max_model_len 65536
--model_ctx 8192
--gen_tokens 512
--max_tokens 1024
--safety_margin 64
--truncate_hard_chars 16000
--chars_per_token 1.5
--gpu_memory_utilization 0.9
```

Answer schema and canonicalization. Outputs were required in the form `{"output": <value>}`, optionally wrapped in `[ANSWER]...[/ANSWER]` tags (`-force_answer_tags`). The canonicalizer strips code fences, scans up to 20k characters for the first valid JSON object, and flattens nested forms such as `{"output":{"return":x}}` $\rightarrow$ `x`. If no valid JSON is found, a format-only repair prompt rewrites the output into the required schema.

Scoring. Exact match (EM) was computed as follows: numbers are compared using `isclose` with relative and absolute tolerance 10^{-3}; strings are case-folded, trimmed, and unquoted if necessary.

Prompts (verbatim). Critique-B (system):

```
You are a structured reviewer.
Provide actionable findings
and a fix plan. Do not reveal or
approximate any expected value.
```

Critique-B (user):

```
Conversation:
{conversation}

Draft answer:
{draft}

Write sections 'Findings:' and 'Fix:'
in bullet points.
Do not include or infer the expected value.
End with VERDICT: KEEP or VERDICT: REVISE.
```

Revision (no-leak, system):

```
You are a careful editor. Revise the draft
strictly according to the feedback.
```

```
Do not include analysis.
Provide only the improved final answer.
```

Revision (no-leak, user):

```
Conversation:
{conversation}

Draft answer:
{draft}

Feedback:
{feedback}

Now produce the corrected final answer only.
```

Format-repair (system):

```
You must output ONLY:
[ANSWER]
{"output": <value>}
[/ANSWER]
No other text, code, or explanations.
```

Format-repair (user):

```
Rewrite into the required format.
If it already contains the
needed value, keep it.
------
{raw_text}
------
```

F Prompt templates

We show the Prompt Templates employed in our experiments, capturing the structural variations in how tasks were framed for the models. Figures 5 and 7 illustrate the prompt templates for input prediction and output prediction, respectively. In addition, Figures 6 and 8 present the corresponding prompt templates that incorporate inversion, demonstrating how the task formulation changes under this modification.

G Model URLs

Table 6 lists the HuggingFace model URLs for the remaining evaluated models.

H Pass@5 Radial Plots

Figures 9–12 show per-model Pass@5 radial plots across all four I/O prediction tasks for each compression algorithm. Each axis corresponds to one task, and each line represents a model, allowing direct visual comparison of strengths and weaknesses across tasks and model families.

I Input length vs. pass@5

Figures 13–16 show the relationship between input length and Pass@5 for all four compression algorithms. Each subplot corresponds to a model, with Pass@5 on the y-axis and input-length buckets on the x-axis. Across algorithms, longer inputs consistently yield lower Pass@5, confirming that input complexity is a key difficulty driver.

J Easy and difficult inputs for LLMs

We present examples of the top-5 inputs that appear to be easy or difficult for LLMs across different compression algorithms. Specifically, Tables 7 and 8 report results for AE; Tables 9 and 10 for Huffman; Tables 11 and 12 for LZW; and Tables 13 and 14 for RLE.

K Prompt difficulty for different data source categories

We use the Prompt Difficulty plot to show how challenging different data source categories are, with the x-axis indicating the number of models achieving Pass@5 and the y-axis showing the number of input strings. This breakdown by category highlights which prompt types are widely solved versus those that consistently challenge models. Figure 17 shows for AE, Figure 18 shows for LZW, Figure 19 shows for RLE, and Figure 20 shows for Huffman.

L Real-world significance

Here are some real-world scenarios related to RTCE.

- **Code obfuscation and de-obfuscation:** If the round-trip fails, the de-obfuscated code will contain errors, breaking the original program and demonstrating a failure to understand the underlying logic of the transformations.
- **Data serialisation and deserialisation:** If the LLM misunderstands even minor aspects of the schema (field names, integer vs. string encoding, escape sequences), the round-trip will not reconstruct the original data.

Input prediction prompt template

System. You are a helpful programming assistant designed to execute code. You must verify your output via a round-trip check and self-correct before returning the final JSON.

User. You are given a Python function and an input. Return a *literal* output (no unsimplified expressions, no function calls), enclosed within `[ANSWER]` and `[/ANSWER]` tags. Do *not* include any additional text.

The input and output requirements are as follows:
Input: `s` (`str`): The input string to be duplicated and wrapped.
Output: `return` (`str`): A string starting with "b", followed by two copies of `s`, and ending with "a".

Given the following input:

`"hi"`

Given the following function:

```
[PYTHON]
def main_solution(s):
    s = s + s
    return "b" + s + "a"
[/PYTHON]
```

Can you predict the output without writing any code? Do not include any explanations, reasoning, or extra text. Put your final answer in the following JSON format: `"output": <your output>`, where `<your output>` must strictly match the output requirement specified above.

[THOUGHT]
Let's execute the code step by step:
1. The function `main_solution` is defined, which takes a single argument `s`.
2. The function is called with the argument `"hi"`, so within the function, `s` is initially `"hi"`.
3. Inside the function, `s` is concatenated with itself, so `s` becomes `"hihi"`.
4. The function then returns a new string that starts with `"b"`, followed by the value of `s` (which is now `"hihi"`), and ends with `"a"`.
5. The return value of the function is therefore `"bhihia"`.
[/THOUGHT]

[ANSWER]
{"output": "bhihia"}
[/ANSWER]

The input and output requirements are as follows:
Input: `uncompressed` (`str`): The input string to be compressed.
Output: `return` (list of tuple): A list of `(char, count)` tuples representing the RLE-compressed string.

Given the following output:

`<output>`

Given the following function:

`<decoding_function>`

Can you predict the output without writing any code? Do not include any explanations, reasoning, or extra text. Put your final answer in the following JSON format: `"output": <your output>`, where `<your output>` must strictly match the specified output requirement.

[THOUGHT]

Figure 5: Input prediction prompt template for RLE algorithm.

Input prediction with inversion prompt template

System. You are a helpful programming assistant designed to execute code. You must verify your output via a round-trip check and self-correct before returning the final JSON.

User. You are given a Python function and an input. Return a *literal* output (no unsimplified expressions, no function calls), enclosed within `[ANSWER]` and `[/ANSWER]` tags. Do *not* include any additional text.

The input and output requirements are as follows:
Input: `s` (`str`): The input string to be duplicated and wrapped.
Output: `return` (`str`): A string starting with "b", followed by two copies of `s`, and ending with "a".

Given the following input:

`"hi"`

Given the following function:

```
[PYTHON]
def main_solution(s):
    s = s + s
    return "b" + s + "a"
[/PYTHON]
```

Can you predict the output without writing any code? Do not include any explanations, reasoning, or extra text. Put your final answer in the following JSON format: `"output": <your output>`, where `<your output>` must strictly match the output requirement specified above.

[THOUGHT]
Let's execute the code step by step:
1. The function `main_solution` is defined, which takes a single argument `s`.
2. The function is called with the argument `"hi"`, so within the function, `s` is initially `"hi"`.
3. Inside the function, `s` is concatenated with itself, so `s` becomes `"hihi"`.
4. The function then returns a new string that starts with `"b"`, followed by the value of `s` (which is now `"hihi"`), and ends with `"a"`.
5. The return value of the function is therefore `"bhihia"`.
[/THOUGHT]

[ANSWER]
{"output": "bhihia"}
[/ANSWER]

The input and output requirements are as follows:
Input: `uncompressed` (`str`): The input string to be compressed.
Output: `return` (list of tuple): A list of `(char, count)` tuples representing the RLE-compressed string.

Given the following output:

`<output>`

Given the following function:

`<encoding_function>`

The function `main_solution` performs encoding. You must use the inverse logic to implement the decoding function, `main_solution_inverse`, to infer your answer, not run or duplicate it directly.

Can you predict the output based on `main_solution_inverse`? Do not include any explanations, reasoning, or extra text. Put your final answer in the following json format: "output": <your output>, where <your output> should strictly match the output requirement as specified.

[THOUGHT]

Figure 6: Input prediction with inversion prompt template for RLE algorithm.

Output prediction prompt template

System. You are a helpful programming assistant designed to execute code. You must verify your output via a round-trip check and self-correct before returning the final JSON.

User. You are given a Python function and an input. Return a *literal* output (no unsimplified expressions, no function calls), enclosed within `[ANSWER]` and `[/ANSWER]` tags. Do *not* include any additional text.

The input and output requirements are as follows:
Input: `s` (`str`): The input string to be duplicated and wrapped.
Output: `return` (`str`): A string starting with "b", followed by two copies of `s`, and ending with "a".

Given the following input:

`"hi"`

Given the following function:

```
[PYTHON]
def main_solution(s):
    s = s + s
    return "b" + s + "a"
[/PYTHON]
```

Can you predict the output without writing any code? Do not include any explanations, reasoning, or extra text. Put your final answer in the following JSON format: `"output": <your output>`, where `<your output>` must strictly match the output requirement specified above.

[THOUGHT]
Let's execute the code step by step:
1. The function `main_solution` is defined, which takes a single argument `s`.
2. The function is called with the argument `"hi"`, so within the function, `s` is initially `"hi"`.
3. Inside the function, `s` is concatenated with itself, so `s` becomes `"hihi"`.
4. The function then returns a new string that starts with `"b"`, followed by the value of `s` (which is now `"hihi"`), and ends with `"a"`.
5. The return value of the function is therefore `"bhihia"`.
[/THOUGHT]

[ANSWER]
{"output": "bhihia"}
[/ANSWER]

The input and output requirements are as follows:
Input: `uncompressed` (`str`): The input string to be compressed.
Output: `return` (list of tuple): A list of `(char, count)` tuples representing the RLE-compressed string.

Given the following input:

`<input>`

Given the following function:

`<encoding_function>`

Can you predict the output without writing any code? Do not include any explanations, reasoning, or extra text. Put your final answer in the following JSON format: `"output": <your output>`, where `<your output>` must strictly match the specified output requirement.

[THOUGHT]

Figure 7: Output prediction prompt template for RLE algorithm.

Output prediction with inversion prompt template

System. You are a helpful programming assistant designed to execute code. You must verify your output via a round-trip check and self-correct before returning the final JSON.

User. You are given a Python function and an input. Return a *literal* output (no unsimplified expressions, no function calls), enclosed within `[ANSWER]` and `[/ANSWER]` tags. Do *not* include any additional text.

The input and output requirements are as follows:
Input: `s` (`str`): The input string to be duplicated and wrapped.
Output: `return` (`str`): A string starting with "b", followed by two copies of `s`, and ending with "a".

Given the following input:

`"hi"`

Given the following function:

```
[PYTHON]
def main_solution(s):
    s = s + s
    return "b" + s + "a"
[/PYTHON]
```

Can you predict the output without writing any code? Do not include any explanations, reasoning, or extra text. Put your final answer in the following JSON format: `"output": <your output>`, where `<your output>` must strictly match the output requirement specified above.

[THOUGHT]
Let's execute the code step by step:
1. The function `main_solution` is defined, which takes a single argument `s`.
2. The function is called with the argument `"hi"`, so within the function, `s` is initially `"hi"`.
3. Inside the function, `s` is concatenated with itself, so `s` becomes `"hihi"`.
4. The function then returns a new string that starts with `"b"`, followed by the value of `s` (which is now `"hihi"`), and ends with `"a"`.
5. The return value of the function is therefore `"bhihia"`.
[/THOUGHT]

[ANSWER]
{"output": "bhihia"}
[/ANSWER]

The input and output requirements are as follows:
Input: `uncompressed` (`str`): The input string to be compressed.
Output: `return` (list of tuple): A list of `(char, count)` tuples representing the RLE-compressed string.

Given the following input:

`<input>`

Given the following function:

`<decoding_function>`

The function `main_solution` performs decoding. You must use the inverse logic to implement the encoding function, `main_solution_inverse`, to infer your answer, not run or duplicate it directly.

Can you predict the output based on `main_solution_inverse`? Do not include any explanations, reasoning, or extra text. Put your final answer in the following json format: "output": <your output>, where <your output> should strictly match the output requirement as specified.

[THOUGHT]

Figure 8: Output prediction with inversion prompt template for RLE algorithm.

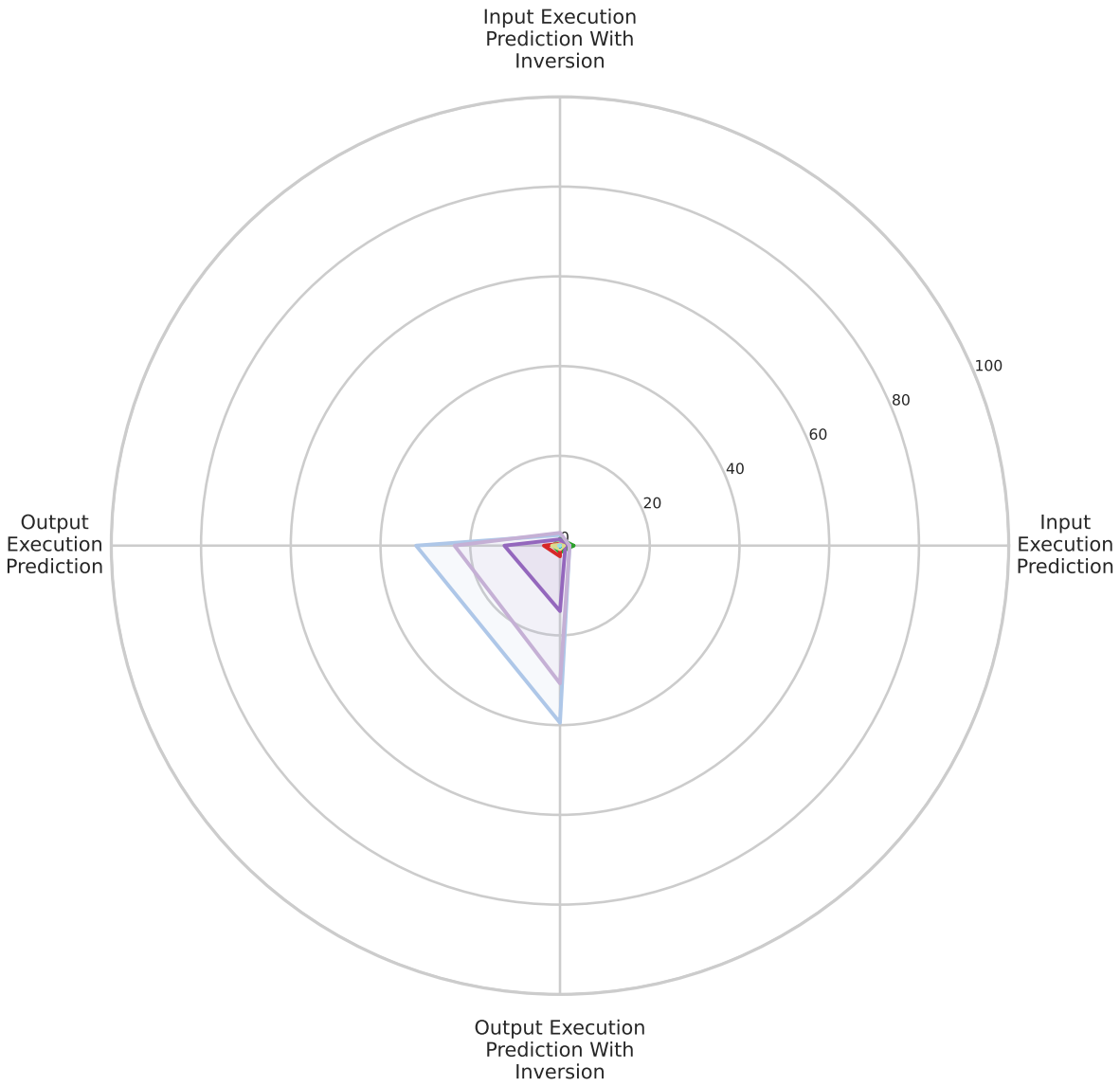


Figure 9: Pass@5 radial plot for AE: models achieve moderate scores on output prediction but collapse on inversion tasks, with larger models showing a more balanced polygon and smaller models nearly flat across all axes.

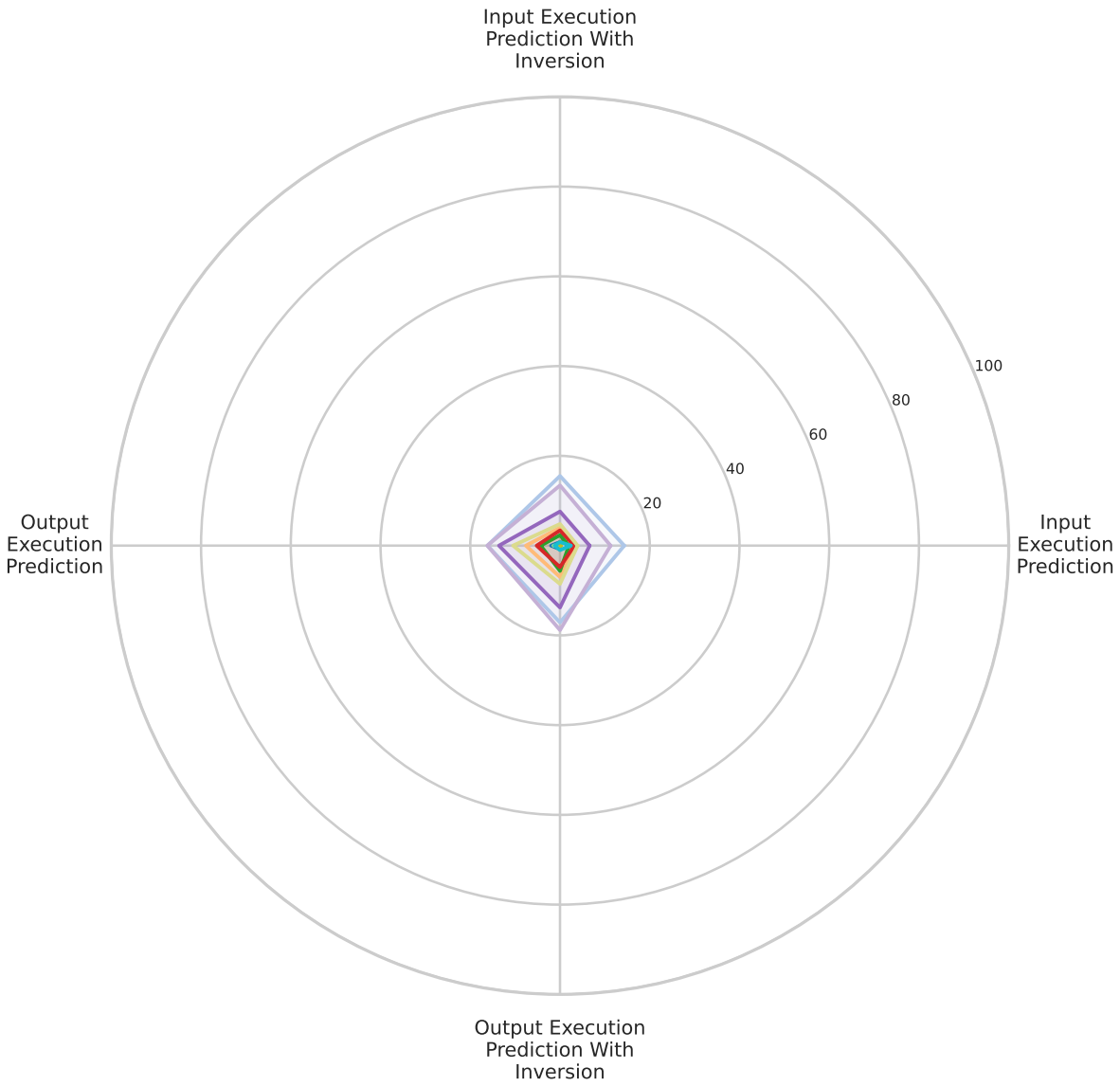


Figure 10: Pass@5 radial plot for LZW: dictionary-based encoding produces a more asymmetric profile than AE, with output prediction axes scoring higher than input prediction axes across nearly all model families.

Figure 11: Pass@5 radial plot for RLE: the most filled polygons of all algorithms, confirming RLE is the most tractable; reasoning-focused models (QwQ-32B, DeepSeek-R1) show notably larger and more symmetric shapes.

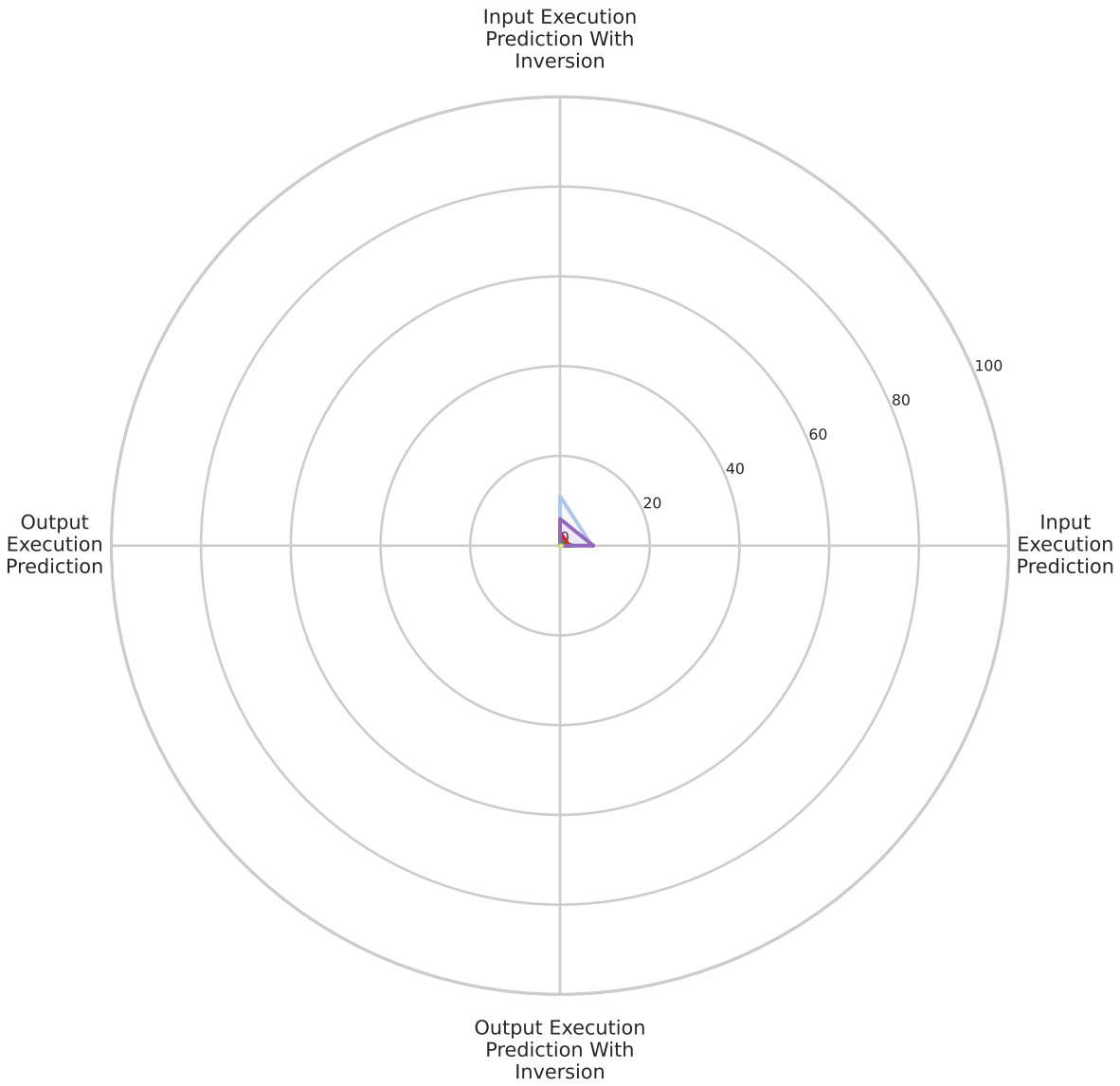


Figure 12: Pass@5 radial plot for Huffman: almost all models collapse to near-zero on every axis, reflecting the extreme difficulty of Huffman encoding; only the largest reasoning models show any measurable area.

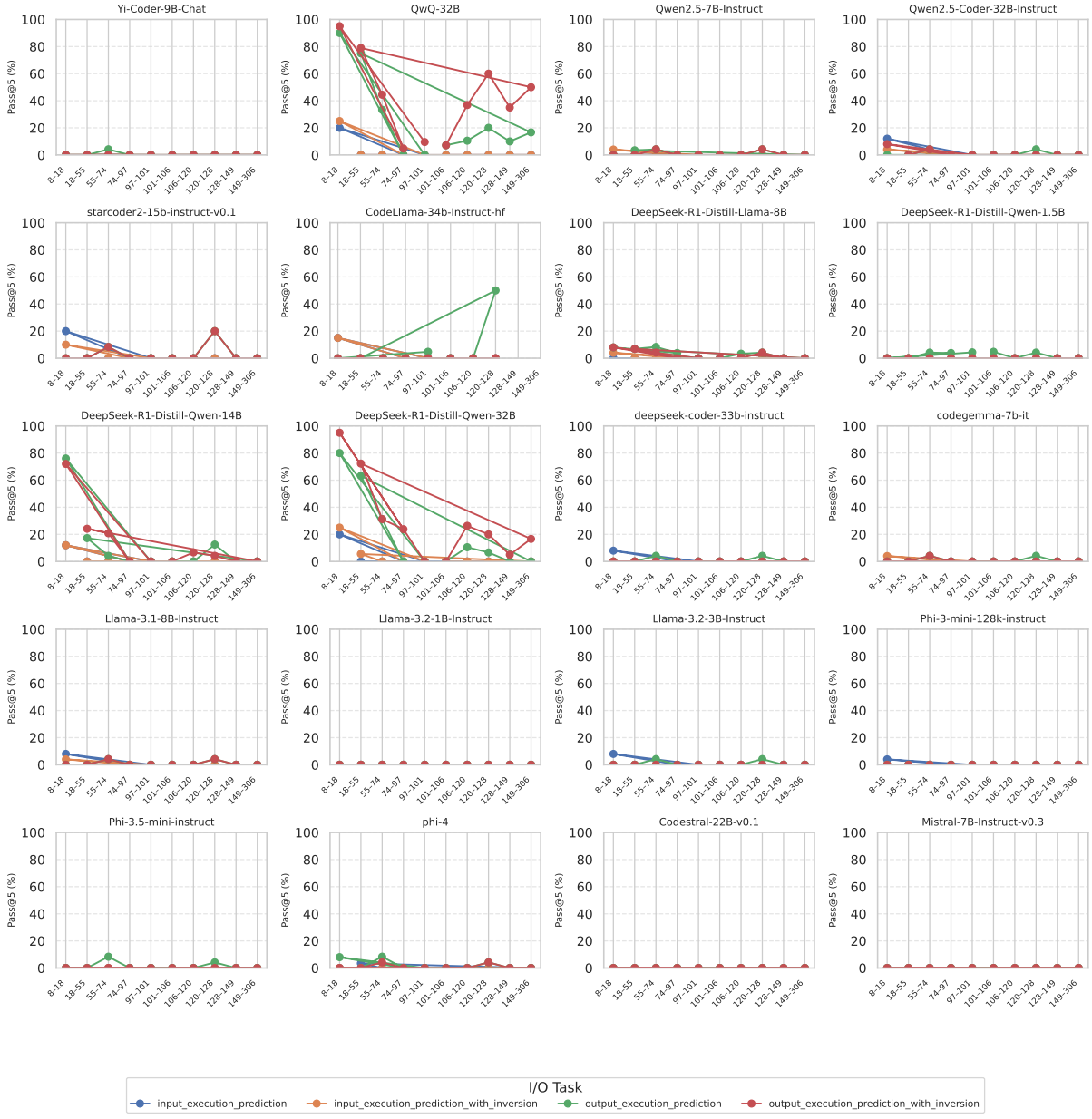


Figure 13: Input length vs. Pass@5 for AE: Pass@5 drops sharply beyond short inputs, with nearly all models failing on strings longer than 100 characters.

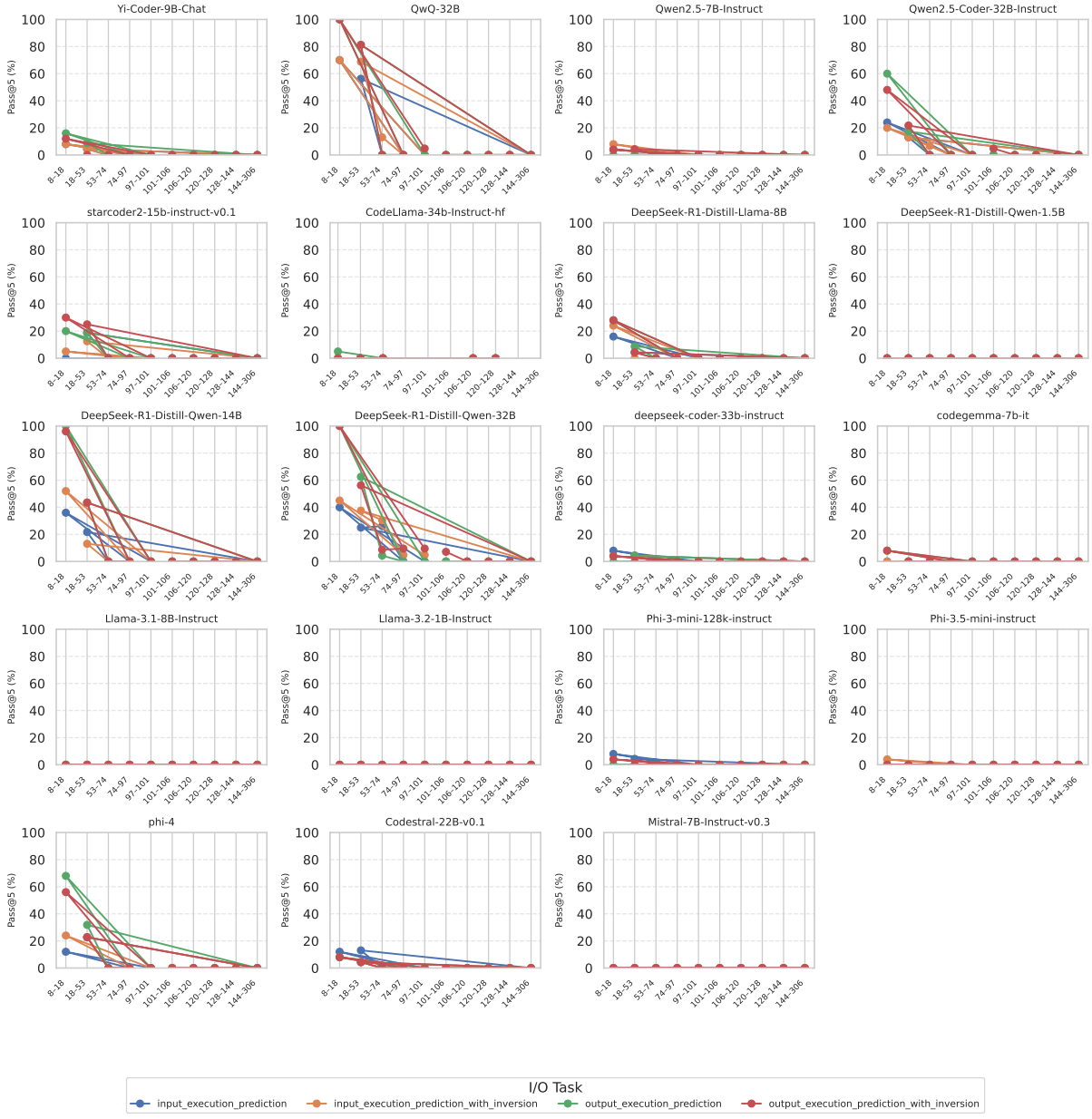


Figure 14: Input length vs. Pass@5 for LZW: the length-performance degradation is steeper than AE, consistent with LZW's growing dictionary making longer inputs exponentially harder to trace.

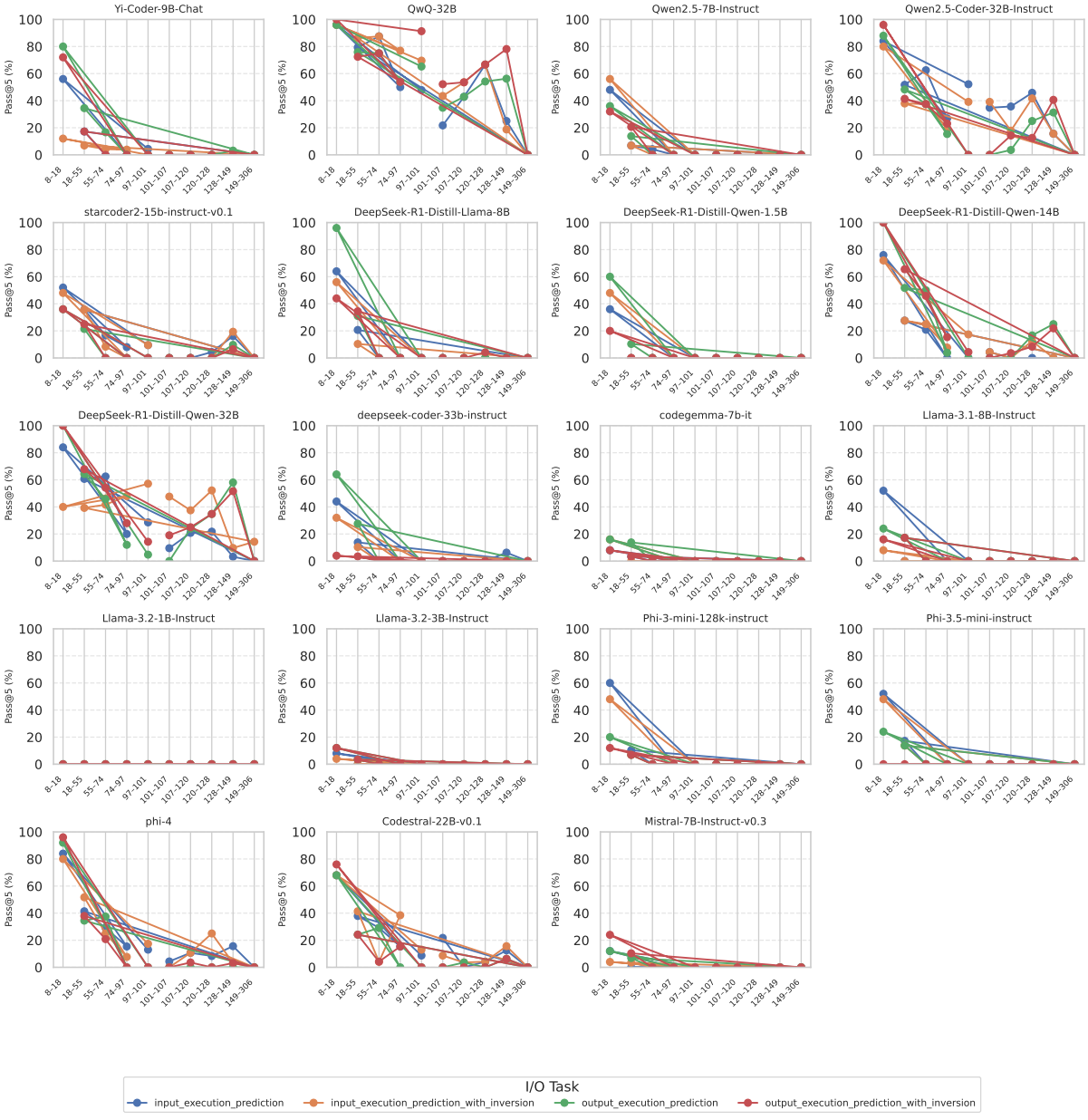


Figure 15: Input length vs. Pass@5 for RLE: performance is the most robust to length among all algorithms, yet still degrades for long inputs, particularly those dominated by non-repetitive character patterns.

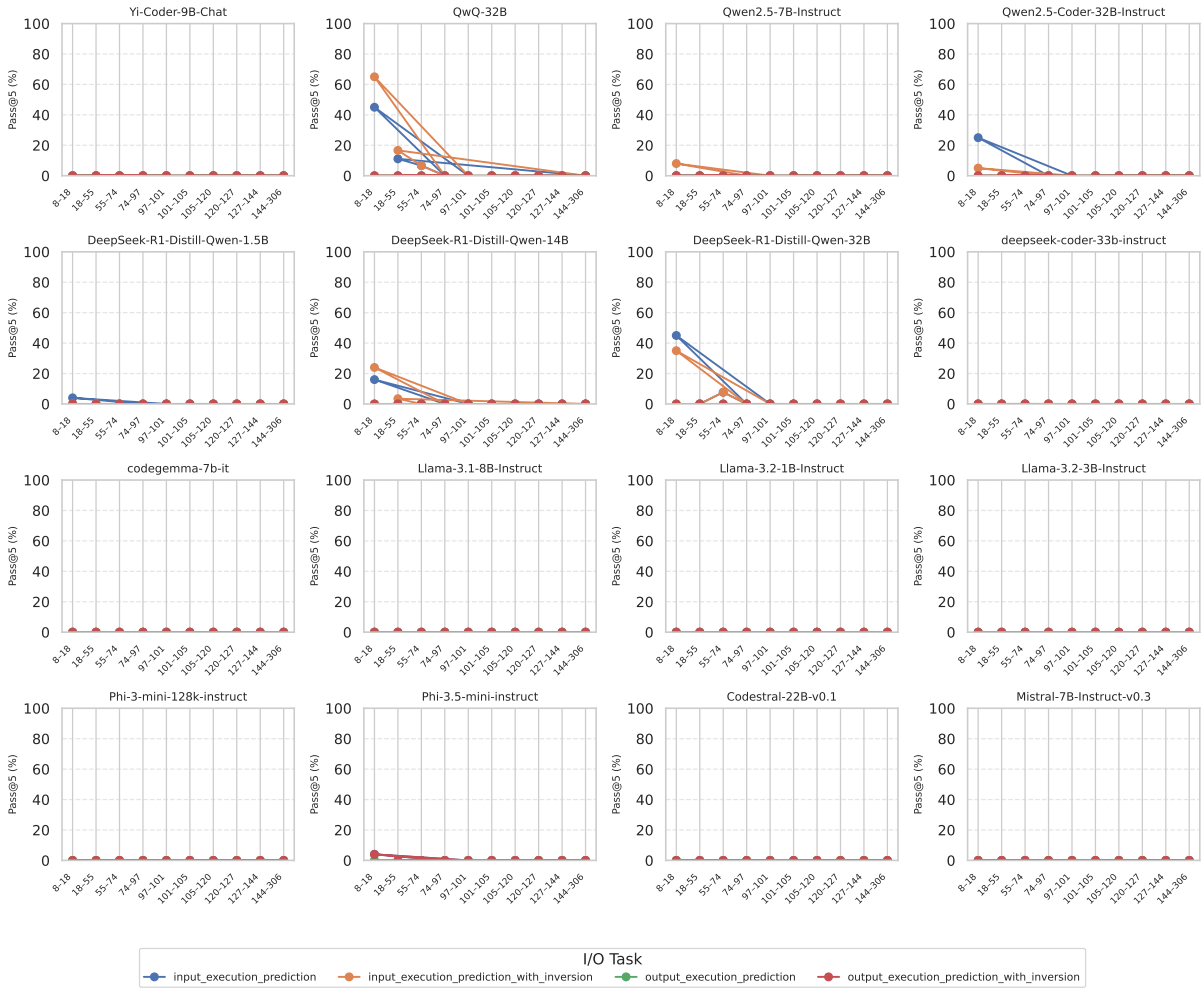
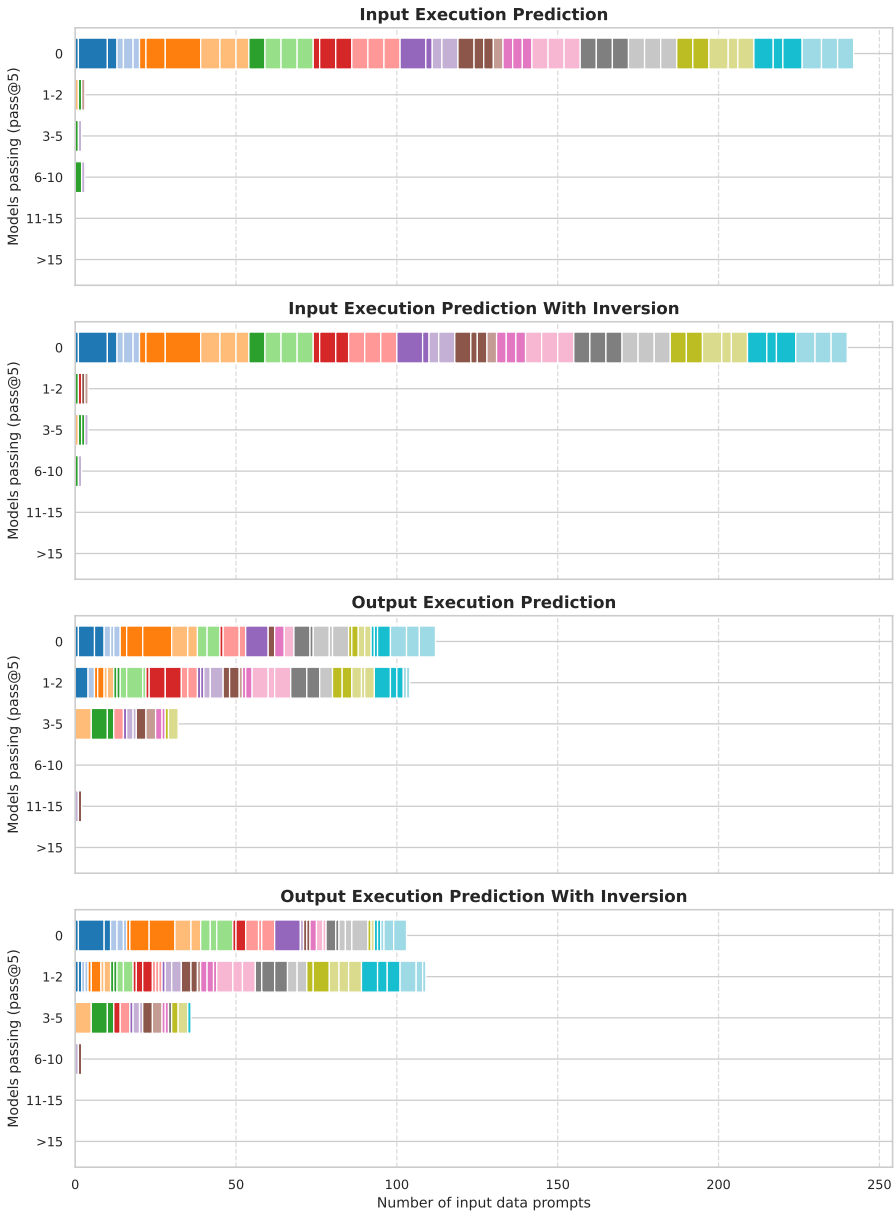


Figure 16: Input length vs. Pass@5 for Huffman: Pass@5 remains near zero across all length buckets, confirming that Huffman difficulty is driven by algorithmic complexity rather than input length alone.



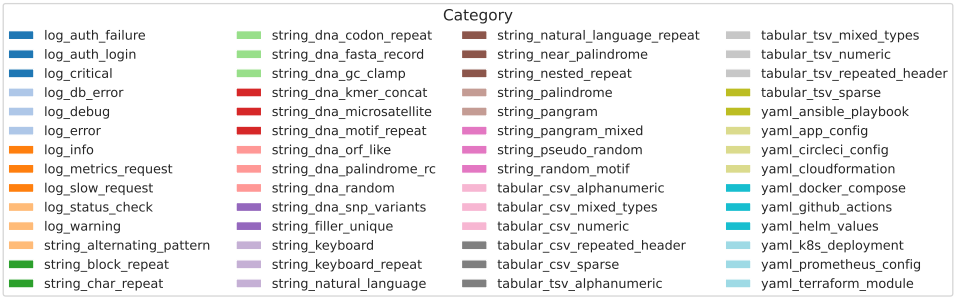
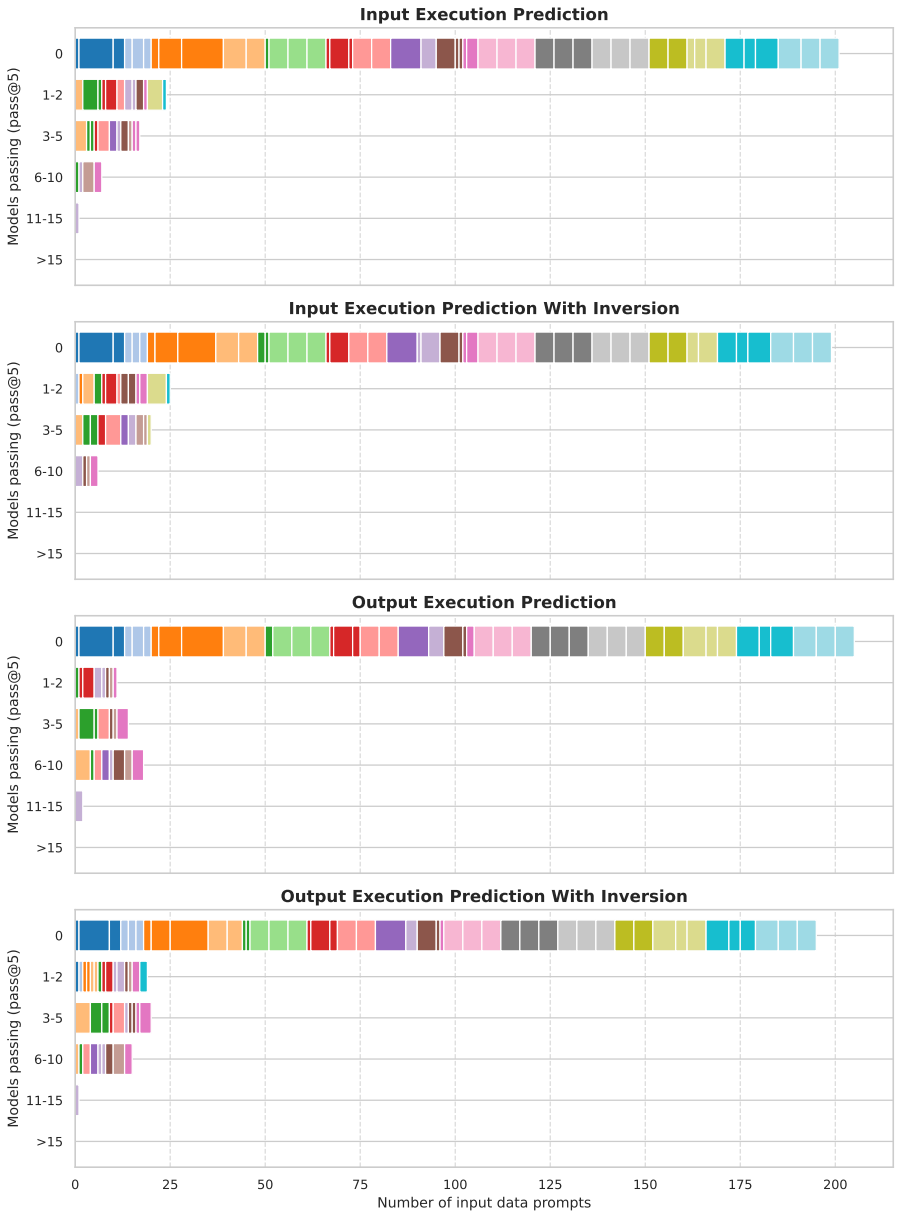


Figure 17: Prompt difficulty for AE: most prompts are solved by zero models, with short repetitive strings (e.g., keyboard sequences) forming the small solvable tail, while structured multi-line inputs such as YAML configs are universally failed.



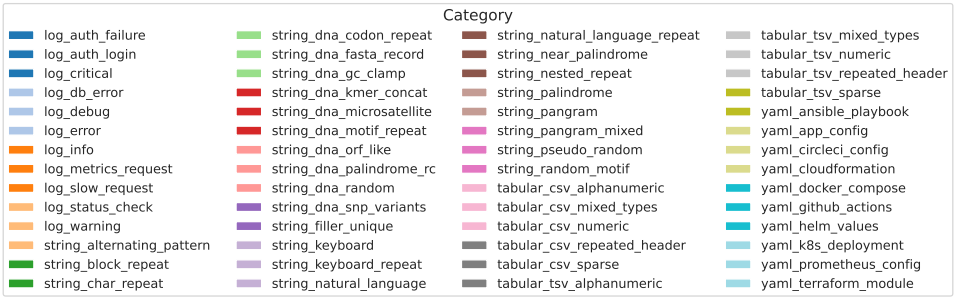


Figure 18: Prompt difficulty for LZW: dictionary-dependent encoding makes nearly all categories hard; only short, low-entropy strings like keyboard sequences are solved by a meaningful fraction of models, while log lines and config files remain universally unsolved.

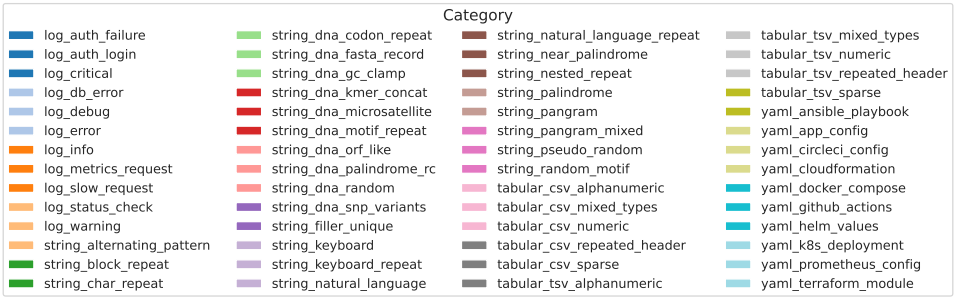
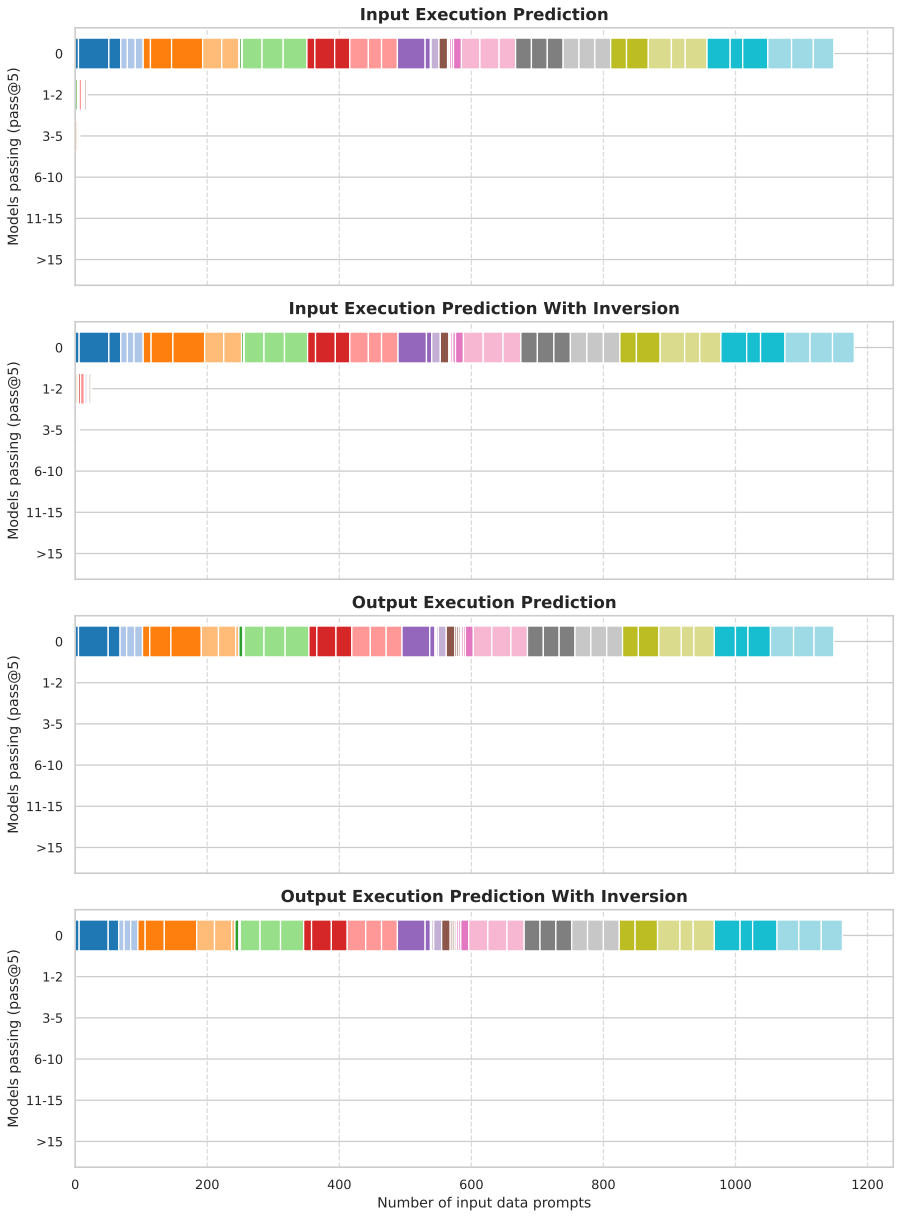


Figure 19: Prompt difficulty for RLE: the most solvable algorithm overall, with a larger right-heavy tail driven by simple repetitive strings; however, long DNA sequences and random character strings remain solved by very few models.



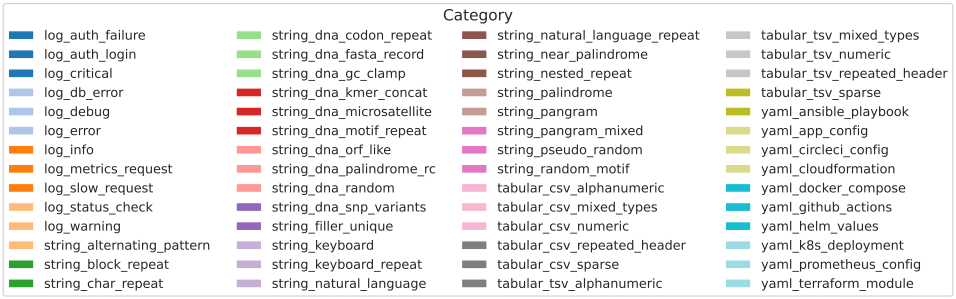


Figure 20: Prompt difficulty for Huffman: frequency-table construction makes this the hardest algorithm; nearly all prompts across all categories are solved by zero models, with only a small subset of short, high-repetition strings reaching partial success.

Model	Size (B)	HuggingFace URL
Yi-Coder-9B-Chat	8.80	https://huggingface.co/01-ai/Yi-Coder-9B-Chat
codegemma-7b-it	8.50	https://huggingface.co/google/codegemma-7b-it
Qwen3-4B	4.00	https://huggingface.co/Qwen/Qwen3-4B
Qwen3-8B	8.00	https://huggingface.co/Qwen/Qwen3-8B
Qwen3-32B	32.80	https://huggingface.co/Qwen/Qwen3-32B
starcoder2-15b-instruct-v0.1	15.00	https://huggingface.co/bigcode/starcoder2-15b-instruct-v0.1
CodeLlama-70b-Python-hf	70.00	https://huggingface.co/codellama/CodeLlama-70b-Python-hf
CodeLlama-34b-Instruct-hf	34.00	https://huggingface.co/codellama/CodeLlama-34b-Instruct-hf
Phi-3-mini-128k-instruct	3.80	https://huggingface.co/microsoft/Phi-3-mini-128k-instruct
Phi-3.5-mini-instruct	3.80	https://huggingface.co/microsoft/Phi-3.5-mini-instruct
Llama-3.1-8B-Instruct	8.03	https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct
Llama-3.1-70B-Instruct	70.00	https://huggingface.co/meta-llama/Llama-3.1-70B-Instruct
Llama-3.2-1B-Instruct	1.00	https://huggingface.co/meta-llama/Llama-3.2-1B-Instruct
Llama-3.2-3B-Instruct	3.00	https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct
phi-2	2.78	https://huggingface.co/microsoft/phi-2
phi-4	14.70	https://huggingface.co/microsoft/phi-4
Codestral-22B-v0.1	22.20	https://huggingface.co/mistralai/Codestral-22B-v0.1
Mistral-7B-Instruct-v0.3	7.24	https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.3
DeepSeek-R1-Distill-Llama-8B	8.03	https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Llama-8B
DeepSeek-R1-Distill-Qwen-1.5B	1.50	https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B
DeepSeek-R1-Distill-Qwen-14B	14.80	https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-14B
DeepSeek-R1-Distill-Qwen-32B	32.80	https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-32B
DeepSeek-R1-Distill-Llama-70B	70.60	https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Llama-70B
DeepSeek-R1-0528-Qwen3-8B	8.19	https://huggingface.co/deepseek-ai/DeepSeek-R1-0528-Qwen3-8B
deepseek-coder-33b-instruct	33.30	https://huggingface.co/deepseek-ai/deepseek-coder-33b-instruct
Qwen2.5-7B-Instruct	7.62	https://huggingface.co/Qwen/Qwen2.5-7B-Instruct
Qwen2.5-72B-Instruct	72.00	https://huggingface.co/Qwen/Qwen2.5-72B-Instruct
Qwen2.5-Coder-14B-Instruct	14.80	https://huggingface.co/Qwen/Qwen2.5-Coder-14B-Instruct
Qwen2.5-Coder-32B-Instruct	32.80	https://huggingface.co/Qwen/Qwen2.5-Coder-32B-Instruct
QwQ-32B	32.80	https://huggingface.co/Qwen/QwQ-32B
gpt-4o-mini	8.00	https://huggingface.co/gpt-4o-mini
WizardLM-70B-V1.0	70.00	https://huggingface.co/WizardLMTeam/WizardLM-70B-V1.0

Table 6: Model URLs used in our experiments.

Input	**models (total)**	**models (correct)**	**correct_rate**
QWERTYUIOP	20	10	0.50
UUUUUUUU	20	9	0.45
IIIIIIIIIII	20	7	0.35
POIUYTREWQ	20	5	0.25
ABABABAB	20	4	0.20

Table 7: AE: top-5 easiest inputs.

Input	models (total)	models (correct)	correct_rate

`# Default values for mychart`
`---`
`replicaCount: 2`
`image:`
`  repository: nginx`
`  tag: stable-0`
	20	0	0.00

`# Default values for mychart`
`---`
`replicaCount: 2`
`image:`
`  repository: nginx`
`  tag: stable-1`
	20	0	0.00

`# Default values for mychart`
`---`
`replicaCount: 2`
`image:`
`  repository: nginx`
`  tag: stable-2`
	20	0	0.00

`# Default values for mychart`
`---`
`replicaCount: 4`
`image:`
`  repository: nginx`
`  tag: stable-0`
	20	0	0.00

`- hosts: all`
`  tasks:`
`  - name: ensure git0 installed`
`  apt:`
`  name: git0`
`  state: present`
	20	0	0.00

Table 8: AE: top-5 hardest inputs.

Input	models (total)	models (correct)	correct_rate
`UUUUUUUU`	16	7	0.44
`ABABABAB`	16	6	0.38
`QWERTYUIOP`	16	6	0.38
`ABCDABCDABCDABCD`	16	4	0.25
`ABCDEFABCDEFABCDEF`	16	3	0.19

Table 9: Huffman: top-5 easiest inputs.

Input	models (total)	models (correct)	correct_rate

`# Default values for mychart`
`---`
`replicaCount: 2`
`image:`
`  repository: nginx`
`  tag: stable-0`
	16	0	0.00

`# Default values for mychart`
`---`
`replicaCount: 2`
`image:`
`  repository: nginx`
`  tag: stable-1`
	16	0	0.00

`# Default values for mychart`
`---`
`replicaCount: 4`
`image:`
`  repository: nginx`
`  tag: stable-0`
	16	0	0.00

`- hosts: all`
`  tasks:`
`    - name: ensure git1 installed`
`      apt:`
`        name: git1`
`        state: present`
	16	0	0.00

`- hosts: all`
`  tasks:`
`    - name: ensure git2 installed`
`      apt:`
`        name: git2`
`        state: present`
	16	0	0.00

Table 10: Huffman: top-5 hardest inputs.

Input	models (total)	models (correct)	correct_rate
`QWERTYUIOP`	20	16	0.80
`POIUYTREWQ`	20	14	0.70
`FOX BROWN LAZY QUICK JUMPS`	20	10	0.50
`ITBVUUVBTI`	20	10	0.50
`MHYRFFRYHM`	20	10	0.50

Table 11: LZW: top-5 easiest inputs.

Input	models (total)	models (correct)	correct_rate
L(?OD4XezFnsc!ufFt-1A.-f3BDm:{K:wqA`^@&+qVC% jbmfx J0qA4~XfzVg!.)x3e$Ojf2W0%GN%M:w?u-MgTo3>46 T#6~JTu~?B4y-	20	0	0.00
PESSEPPESPESSEPPESPESSEPPES	20	0	0.00
[2025-01-01 00:00:00] CRITICAL - DELETE /status by user:92f5df7b in 2.123s from 192.168.52.6 (server)	20	0	0.00
[2025-01-01 00:00:00] CRITICAL - GET /api/v1/ order by user:b31022f0 in 0.792s from 192.168.111.238 (cache)	20	0	0.00
[2025-01-01 00:00:00] DEBUG - GET /metrics by user:018267c4 in 1.229s from 192.168.194.244 (worker)	20	0	0.00

Table 12: LZW: top-5 hardest inputs.

Input	models (total)	models (correct)	correct_rate
POIUYTREWQ	19	18	0.95
QUICK JUMPS THE OVER LAZY	19	18	0.95
QWERTYUIOP	19	18	0.95
UUUUUUUU	19	18	0.95
CCCGGGCCCGGG	19	17	0.89

Table 13: RLE: top-5 easiest inputs.

Input	models (total)	models (correct)	correct_rate
?88dY1V7C[?)Xf,&qocnnse!+>5T^];S'5P QAZDVy}gt {^38E;'j~e'H'&j]`c4'a*7(l(v>S/h?jl%o*UtjhbHA:u{H >AR0urFZH)!Z	19	0	0.00
AAACTCCGAATACCGATCCAGAGAGAGAGAGAGAGAGAGAGAGATC GACTGTCGCCGATGCGAT	19	0	0.00
ATATACTTGGTCGGCGATCGAATAATAATAATAATAATAATAATA ATAATAATAATAATAATAATCCGTCAATTAGGATCGCGTC	19	0	0.00
ATGCCTGTCGTAGTAGTGGTCGTAGTCGTACGGTCAGCGGCAGCAGCC CCTCGGAAACCGGCTCCTTCACGGTCCCCTCGAGTGCCTGTTCCAGTG TCGTAG	19	0	0.00
ATGCGGCCATCTCGGGCTGTTCCTGAGAAAGTCCCATCTGTCGCTCGG CCACCAGTACGGGTTGCTTCAGTAAAAGTTGCCGTGTCGTCATCGTCG CGAGTCCCAAAAGCCGCCCCGTCTCCGCGATCATCAGTGGTATAA	19	0	0.00

Table 14: RLE: top-5 hardest inputs.