Testing the Limits of Large Language Models on Regular Languages

Anonymous ACL submission

Abstract

Large language models (LLMs) have shown strong performance across reasoning benchmarks, yet whether this reflects genuine symbolic reasoning or sophisticated pattern matching remains unresolved. We investigate this through regular languages, the simplest formal language class, providing a fully verifiable testbed where correctness can be evaluated exactly. Applying a staged diagnostic framework to frontier models (GPT-5.2, Grok-4.1, Gemini-2.5) and open-source Qwen 2.5 models (1.5B, 7B, 14B), we identify eleven failure modes spanning semantic interpretation, generative execution, and DFA construction, recurring across all architectures regardless of scale. We introduce a regex-to-DFA benchmark (22,500 training and 1,000 test examples) and investigate mitigation through prompting, fine-tuning, and curriculum learning. Fine-tuning yields substantial recovery, but gaps persist at the highest complexity tier that no training format or data ordering resolves. Mechanistic analysis reveals why: outcomes are prematurely determined in early layers, and inference-time steering recovers only a small fraction of the residual gap. Symbolic reasoning failures in LLMs are structural and partially fixable.

1 Introduction

The past several years have produced a striking convergence of evidence suggesting that large language models may be developing genuine reasoning abilities. Strong performance across arithmetic reasoning, program synthesis, and symbolic benchmarks has reinforced this impression, with transformer-based models appearing to carry out structured inference across increasingly demanding tasks (Lewkowycz et al., 2022; Welleck et al., 2021; Azerbayev et al., 2023; Glazer et al., 2024). Yet a parallel body of work has steadily complicated this picture. Many reasoning benchmarks admit shortcut strategies, pattern matching, surface heuristics, and template memorization that allow high accuracy without genuine symbolic understanding (Mitchell and Krakauer, 2023; Yue et al., 2024; Katz et al., 2025). When evaluated under conditions that require precise symbolic reasoning rather than pattern completion, performance often degrades sharply: BIG-Bench Extra Hard, for instance, reveals systematic failures on tasks that demand sound reasoning beyond surface-level correlations (Kazemi et al., 2025). Statistical fluency does not necessarily translate into structurally grounded competence. This tension raises a question that end-to-end benchmark accuracy alone cannot resolve: *Do LLMs internalize the algebraic structure underlying a task, or do they approximate it through distributional interpolation?* The distinction matters because, as Chollet (2019) and Mirzadeh et al. (2025) argue, genuine reasoning systems must operate over sound formal rules and abstract symbolic operators to generalize reliably beyond their training distributions. Benchmark success is insufficient evidence unless models demonstrate consistent adherence to the underlying formal structure of the task, a property that aggregate accuracy metrics are poorly suited to detect.

Regular languages (RL) (Sipser, 2013) offer a well-suited testbed for probing this distinction. Its operators have unambiguous semantics: concatenation, union, Kleene star, and Brzozowski derivatives admit formal definitions that eliminate interpretive ambiguity. Reasoning even in the context of the least expressive class of formal languages, namely the RLs, requires non–trivial symbolic manipulation, including limited forms of recursion, and global structural constraints. Moreover, correctness can be evaluated exactly making it possible to distinguish failures of language understanding from failures of symbolic execution. Existing Theory-of-Computation benchmarks such as TuringQ (Zahraei and Asgari, 2024) largely assess declarative knowledge, including definitions, clo-

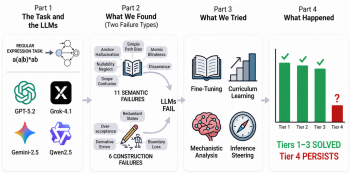


Figure 1: **Overview of our framework.**

sure properties, and well–known results, without testing whether models can apply that knowledge to concrete instances by correctly manipulating symbolic representations (Golesteanu et al., 2024). In particular, they leave unexamined whether models preserve critical invariants such as nullability, reachability, and language equivalence across transformations between regular expressions and automata. This paper addresses that gap directly. We argue that contemporary LLMs exhibit systematic and structural failures in symbolic reasoning, even for RLs and that these failures are not resolved by scale, prompting, or inference-time intervention alone. We substantiate this claim through a staged evaluation protocol applied to three state-of-the-art frontier models, namely GPT-5.2, Grok-4.1, and Gemini-2.5, as well as open-source Qwen 2.5 models at 1.5B, 7B, and 14B parameters, using the regex-to-DFA benchmark dataset introduced in (Shelat et al., 2026). Rather than reporting aggregate accuracy, we decompose performance into four staged probes: natural-language interpretation of the regex; generation of accepted and rejected examples; classification of edge-case strings; and DFA construction via both intuitive reasoning and Brzozowski derivatives. This decomposition allows us to separate what a model can articulate from what it can actually compute. For Qwen 2.5, we further investigate whether these failures can be mitigated through supervised fine-tuning, curriculum learning, and mechanistic intervention, using a four-tier benchmark that scales from simple two-state DFAs to full subset construction. [1]

Our evaluation surfaces four findings not previously characterized in Theory of Computation (ToC)-oriented evaluations. *Models consistently misinterpret operators (concatenation, union, Kleene star) they can correctly define, revealing a dissociation between declarative and procedural knowledge.* They contradict their own interpretations when generating examples, exposing unstable internal representations. During DFA construction, they over-generalize, collapse semantically distinct states, or violate boundary and nullability constraints; these failures persist under chain-of-thought and tree-of-thought prompting and recur across architectures. Fine-tuning substantially recovers performance, yet gaps persist at the highest complexity tier that curriculum restructuring cannot close, as mechanistic analysis traces these failures to weight-level decisions made during pretraining. Inference-time steering recovers only a marginal fraction of the residual gap. Meaningful progress on symbolic reasoning may require models that enforce formal invariants by construction, rather than approximating them through pattern matching (Joishy et al., 2026).

2 Can LLMs Understand Regex?

Understanding the failures requires addressing: *At what stage does the breakdown occur when a model translates a regex into a DFA?* A model may fail due to semantic misinterpretation, specifically misreading nullability, compositional scope, or operator precedence, or due to an inability to execute automaton construction despite a correct interpretation. Conflating these failure modes obscures their remediation. We therefore introduce a staged diagnostic framework that probes reasoning about algebraic properties of the regex before DFA construction is attempted, isolating genuine failures of symbolic execution from superficial success driven by pattern completion or recall.

Cross-Consistency Protocol. As part of this staged diagnostic framework, we introduce a three-task evaluation protocol in which model responses are generated within a **single prompt context** (see Appendix B for example). Asking all three tasks within the same prompt is essential: Only then can contradictions be meaningfully attributed to the model's reasoning rather than differences in how it interpreted separate prompts, effectively separating genuine understanding from pattern matching.

Task 1: Natural-Language Translation. The model is asked to paraphrase the regex as a coherent description of the language, identifying admissible prefixes, suffixes, and mandatory structural blocks. This probes whether the model can construct a faithful mapping from syntax to semantics, which forms the foundation for the later tasks. For example, consider regex R_1: $a(a|b)^*a^*a$: The model responded: "Starts with a, then any repe-

[1] Qwen 2.5 provides a clean scaling ladder (1.5B, 7B, 14B) with explicit pretraining (Qwen Team, 2025), ensuring failures reflect genuine symbolic reasoning limitations.

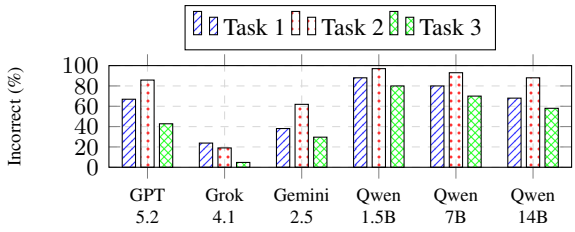


Figure 2: Errors under cross-consistency.

tition of any combinations of a and b, and then **ends with *aa***." This interpretation already contains inconsistencies with the actual semantics, as the true regex requires the string to end only with a single a, not with ***aa***. **Task 2: Rule-Based Execution via Examples.** This task evaluates whether the model can operationalize its understanding of regex from Task 1 by producing concrete instances of the language. The model must generate ten accepted and ten rejected strings. We evaluate (a) *correctness* and (b) *inclusiveness*, by checking whether the examples span structurally distinct branches of the language. This task probes the model's ability to explore the underlying DFA state space, a prerequisite for correct construction. For example, for R_1: *Accepted (model output): aaa*, *aaaa*, *aba* and *Rejected (model output): a*, *baa*, *abab*, ***aa***. Several of these are erroneous (e.g., ***aa*** must be accepted). **Task 3: Symbolic Recognition Verification.** Given a curated set of edge-case strings, the model must provide accept/reject verdicts with brief justifications. This evaluates whether the model can perform stepwise symbolic recognition consistent with the regex given in Task 1. For example, for R_1: The string *aabbaba* was incorrectly classified as *Rejected*, with the explanation that it "does not end with ***aa***".

Results. If an LLM genuinely understands the language, outputs across Tasks 1–3 must be *mutually consistent*. Instead, we observe **operational–descriptive dissonance**: models describe constraints correctly but violate them in generation or recognition. Among frontier models, GPT-5.2 exhibits the highest error rates (85.78% in Task 2), Grok-4.1 performs best, and Gemini-2.5 lies in between. Qwen 2.5 follows the same pattern at higher rates; even at 14B, errors remain at 68%, 88%, and 58% for Tasks 1, 2, and 3. Across all six models, errors peak in Task 2 and are lowest in Task 3, a pattern preserved across architectures and scales, pointing to a structural limitation. See Figure 2.

2.1 Taxonomy of Semantic Failures

Across these six models, we observe **11 systematic failure modes** that arise repeatedly across diverse *regex patterns and structures*. These failures emerge from the Cross-Consistency Protocol, which allows us to isolate errors in interpretation, generation, and symbolic recognition. These are not stochastic irregularities but structural limitations in the models internal representations of RLs.

(Task 1) Semantic Interpretation Errors (i) **Anchor Hallucination**: Models invent prefix or suffix rules that do not exist in the formal specification. For example: Consider regex $(b \mid a(a(bb \mid bbb)^*ba)^*b)^*a(a(bb \mid bbb)^*ba)^*a(bb \mid bbb)^*bba$, LLMs asserted that "all valid strings end with ***bbb***" because it over-focused on the local token cluster ***bbb*** inside the internal block. This ignores the global structure enforcing a literal ***bba*** suffix. Such errors show that LLMs attend to high-frequency substrings rather than the parse-tree semantics. (ii) **Nullability Neglect**: Models confuse 'zero or more' with 'one or more' or treat nullable groups as mandatory. For example: For ba^*, LLMs claimed that "the string must end with **one or more *a***". Similarly, in expressions such as $(aa \mid bb)^*$, models insisted that the language **excludes the empty string.** These errors reveal a failure to preserve nullability. (iii) **Atomic Unit Blindness**: Regex encodes length constraints through atomic block sizes. Models fail to recognize that valid strings must be linear combinations of these lengths. For example: For $(bb \mid bbb)^*$, models generated the string ***b***. While the regex allows strings of length m ($m \in 2\mathbb{N}$ and $m \in 3\mathbb{N}$), i.e., generation of ***b*** is incorrect. The model treats the regex as a "bag of characters" (seeing that b is the only allowed symbol) rather than respecting the indivisible block constraints. (iv) **Scope and Nesting Confusion**: LLMs lose track of which parenthesis scope governs which repetition constraints. For example: Consider $x(ab^*c)^*y$. Here, the leading x and trailing y lie *outside* the repeated block $(ab^*c)^*$. However, LLMs frequently reinterpret the leading or trailing symbols as part of the repeated block, effectively as $(xab^*c)^+$ or $(ab^*cy)^+$. This leads to incorrect claims: **"inner block must appear at least once"**.

(Task 2) Generative Execution and Coverage Failures (v) **Pseudo-Structural Hallucination**: Generated strings mimic the 'texture' of the regex (correct alphabet and substrings) but violate strict

structural boundaries, such as mandatory suffixes or group atomicity. For example: For $(ab)^*cd$, models generated '***ababcdab***'. This string violates the global structure by allowing the repetition to 'leak' past the mandatory terminator. This indicates that the model is predicting the next token based on local probability (seeing *ab* makes *ab* more likely again). (vi) **Simple-Path Bias**: Models overwhelmingly generate examples from the simplest branch of $(A \mid B)$ and ignore deeper paths. For example: In $(a^*b \mid b^*aa(aa \mid bb)^*a)ba^*$, the accepted list almost exclusively follows the a^*b branch. The complex second branch is rarely explored, showing that LLMs do not traverse the full nondeterministic state space. (vii) **Complexity Aversion**: Valid strings are incorrectly rejected due to heuristic biases against length, irregularity, or perceived 'messiness'. For example: Strings such as ***baabba***, valid under several tested regexes, were rejected without syntactic justification. This suggests classification relies on similarity to simple prototypes rather than formal membership tests.

(Task 3) Execution and Reasoning Errors These failures operationally demonstrate the presence of reasoning errors in LLMs. (viii) **Trace Fabrication**: Models produce execution traces containing nonexistent substring matches. For example: For *aaababbb*, LLMs claimed that the substring (index starts with position 0) contained a segment at positions 5–7 matching ***bba***; the actual substring was ***bbb***. The reasoning is generated post-hoc and does not reflect a causal simulation of a finite automaton. (ix) **Greedy Parsing Failures**: Models emulate greedy but non-backtracking regex engines. For example: For the regex $(a \mid ab)^*b$ and the input string *abab*, classical automata accept by segmenting it as $\boldsymbol{ab} \cdot \boldsymbol{a} \cdot \boldsymbol{b}$, where the final b satisfies the required suffix. However, LLMs often consume the entire string inside $(a \mid ab)^*$ by choosing the greedy segmentation $\boldsymbol{ab} \cdot \boldsymbol{ab}$, leaving no trailing b for the final literal. This reflects a lack of backtracking. (x) **Indexing and Positional Drift**: Models lose track of indices during step-by-step reasoning. For example: LLMs miscounted the length of ***abbaabba*** as 7 **rather than** 8, skipping a repeated b. This behaviour closely mirrors well-documented *counting failures* in LLMs, where models struggle to maintain exact positional or cardinality information (Kalai et al., 2025). These errors stem partly from unstable positional representations during multistep reasoning and partly from tokenization effects, since models operate over subword tokens rather than raw characters, leading to index drift and loss of exact string boundaries. (xi) **Descriptive–Operational Dissonance**: The constraints stated in Task 1 do not match recognition or reasoning behavior in Tasks $2-3$. For example: After asserting that "the language forbids strings ending in ***b***", LLMs generated multiple accepted examples ending in ***b***. This shows that natural-language explanations and symbolic execution are produced by separate, weakly correlated processes.

These failures reveal a fragmented reasoning pipeline: semantic errors propagate into generation and recognition rather than being contained. Full raw outputs are in Appendix F.

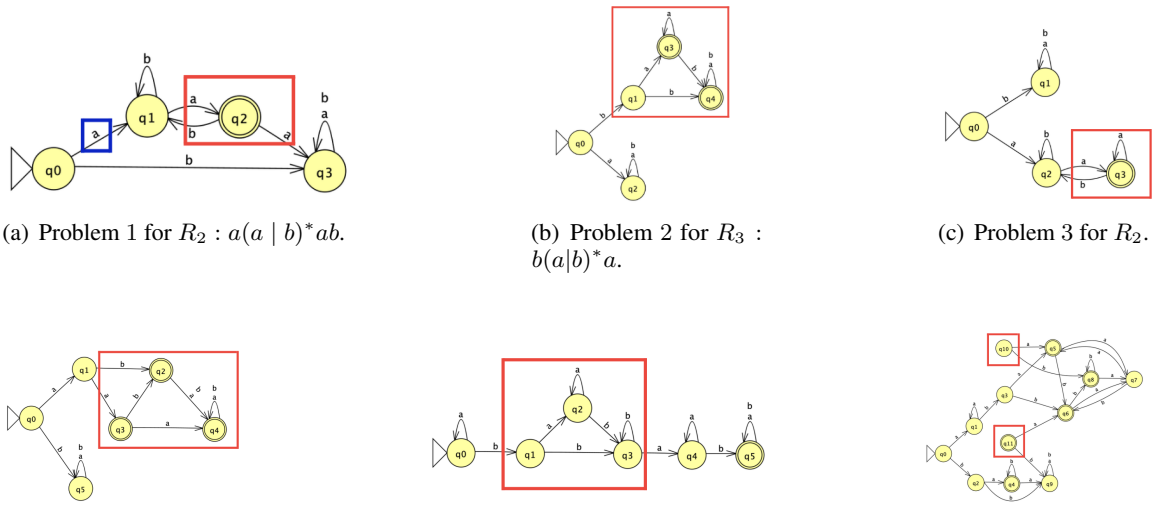
3 Failure Modes in DFA Construction

The semantic failures outlined in Section 2 reveal that LLMs often misinterpret core operators, collapse structural distinctions, or generate incorrect examples. One central question emerges: *Can contemporary LLMs successfully construct a correct DFA from a regex, and how do their semantic weaknesses manifest during this final stage of synthesis?* If LLMs lack a symbolic representation of RL, then their construction attempts, whether intuitive or procedural, should exhibit deeper structural failures, including incorrect state boundaries, misapplied derivatives, improper normalization, and over-acceptance of strings that violate global constraints. We analyze DFAs produced under different prompting strategies not only to determine *whether* models succeed, but also *why* and *where* their symbolic reasoning pipelines break down.

Experimental Setup. We evaluate DFA construction using the 180-problem benchmark of Shelat et al. (2026) under Tree-of-Thought prompting (Yao et al., 2023). We probe two strategies: **Intuitive Construction**, where the model builds the DFA without any procedural template, as a student might in a traditional Theory of Computation course, and **Derivative-Based Construction**, where it follows Brzozowski derivatives (Sipser, 2013). Thompson Construction is deliberately excluded as it bypasses the symbolic reasoning steps we aim to evaluate. Prompts are in Appendix D.

3.1 DFA Construction Errors

We note **six recurrent classes of DFA construction errors**. These are not isolated or accidental; rather, they reveal weaknesses in the symbolic processes required for correct DFA derivation.



(a) Problem 1 for $R_2 : a(a \mid b)^*ab$.

(b) Problem 2 for $R_3 : b(a|b)^*a$.

(c) Problem 3 for R_2.

(d) Problem 4 for $R_4 : (ab^+a) \mid (aba^+b)$.

(e) Problem 5 for $R_5 : (a^*ba^+)(b^*ab)$.

(f) Problem 6 (redundant states).

Figure 3: Common DFA construction failures across Problems 1-6. Red shows errors; blue shows correct paths.

Problem 1: Misinterpreting Kleene-Star Structure. When constructing automata from the regex, LLMs often fail to preserve the semantics of $(regex)^*$, treating repeated segments as if they were obligatory or permitting a single expansion. Figure 3(a) illustrates a typical pattern after considering the regex $a(a \mid b)^*ab$, the model respects the starting condition (state with a (in blue)) but collapses for the repeated block $(a \mid b)^*$ (in red), yielding a machine that accepts strings ending only in ***a***. This error indicates a failure to reason about unbounded repetition and nullable components.

Problem 2: Errors in Brzozowski-Derivative. Under derivative-based prompting, LLMs must manipulate regex symbolically. We frequently observe failures to normalize equivalent residuals or to propagate nullability correctly. This problem directly corresponds to the '*Nullability Neglect*' failure identified in Section 2. In Figure 3(b), the model attempts to take the derivative of the regex $R_3 = b(a \mid b)^*a$ but commits a critical substitution error: it replaces the nullable component ε with the universal language $(a \mid b)^*$, marked in red. Specifically, the derivative of the residual $(a \mid b)^*a$ with respect to a should yield $(a \mid b)^*a \cup \varepsilon$. Instead, the model outputs $(a \mid b)^*a \cup (a \mid b)^*$, which simplifies to $(a \mid b)^*$. This substitution creates an overly broad residual that accepts any subsequent input, thereby erasing the constraint (i.e., end with ***a***).

Problem 3: Incorrect Pre-Minimization Structure. For the regex R_2, the model incorrectly designates q_3 as the accepting state, even though q_3 corresponds to reading only the first ***a*** of the final ***ab*** suffix. The transition $q_3 \xrightarrow{b} q_2$ is also wrong: this ***b*** should lead to a *new, distinct accepting state* representing the completed suffix. These incorrect assignments and transitions, highlighted in red in Figure 3(c), constitute the primary pre-minimization errors. The structure of the initial DFA *before minimization* directly determines the correctness of the minimized machine. When the model assigns inconsistent state semantics or incorrect transitions at this early stage, minimization merges states that should remain distinct, propagating the initial structural error into the final DFA.

Problem 4: Over-Acceptance of Non-Language Strings. LLMs collapse dissimilar residual expressions into a single state, producing DFAs that accept strings outside the target language. For regex $R_4 : (ab^+a) \mid (aba^+b)$ in Figure 3(d), merging semantically incompatible components accepts strings such as abb (in red). The model identifies the shared prefix ab but conflates the residual requirements (ending in a vs. ending in b) into a single 'sink-acceptor' state.

Problem 5: Loss of Boundary Conditions Under Concatenation. When languages are composed of multiple constrained sub-expressions, LLMs often apply correct local transformations (e.g., handling each sub-expression independently) but fail to enforce the global constraints that arise under concatenation. For example, in the language

$R_5 : (a^*ba^+)(b^*ab)$, the final a^+ in the first block acts as a boundary; block 2 may begin only after the automaton has consumed the entire run of a**'s** mandated by a^+ . Interestingly, when prompted with the sub-expressions a^*ba^+ and b^*ab in isolation, the model is able to construct correct DFAs for each block. However, as shown in Figure 3(e), it prematurely enters the b^* region of block 2 during construction of R_5 . This boundary violation (marked in red) collapses the required ordering of the concatenated expression. This highlights that the model can correctly handle *atomic* components, yet fails to maintain the required constraints once those components are concatenated.

Problem 6: Creation of Redundant States. Many LLM-generated DFAs contain states that contribute nothing to language recognition. Figure 3(f) shows an example for R_6 : $(a^*a(b|ba)^*(bb|aa)|a^*ba)b^*$ in which an entire subgraph is unreachable or unrelated to any accepting path, revealing a lack of semantic pruning and incomplete reasoning about reachability. This failure is closely related to **Scope and Nesting Confusion** (iv) and **Descriptive–Operational Dissonance** (xi) (see Section 2).

Errors Across Construction Methods. Table 1 reports failure mode frequencies; Grok-4.1 is excluded as it did not produce complete DFAs within budget. Under intuitive construction, over-acceptance (P4) dominates, at 27% for GPT-5.2 and 40% for Qwen-1.5B, with redundant states (P6) as the second major failure (22–30%). Derivative construction reduces over-acceptance but exposes deeper fragility: derivative violations (P2) become dominant at 35–40% in frontier models and 45–60% in Qwen 2.5 variants, with the 14B model remaining above frontier rates (45% vs. 35–40%), indicating scale alone does not close the gap. Pre-minimization errors (P3) remain stable across all scales and methods, suggesting a representational limitation that neither scale nor strategy resolves. Derivative prompting trades one failure mode for another, a trade-off that holds universally.

4 Mitigating Symbolic Reasoning Failures

We investigate mitigation through prompting, supervised fine-tuning, curriculum learning, and mechanistic intervention on the Qwen 2.5 family (1.5B, 7B, 14B) as a reproducible testbed spanning an order of magnitude in scale. All experiments use a regex-to-DFA benchmark introduced in this

Mode	GPT-5.2		Gemini-2.5		Qwen-1.5B		Qwen-7B		Qwen-14B	
	I	D	I	D	I	D	I	D	I	D
P1	12	18	14	20	28	25	22	20	18	17
P2	5	35	6	40	7	60	6	50	5	45
P3	13	15	12	14	18	18	15	15	12	12
P4	27	8	25	11	40	15	35	12	30	10
P5	15	18	16	20	25	25	22	22	18	18
P6	22	12	24	14	30	20	25	18	22	15

Table 1: Failure (%): intuitive (I) vs. derivative (D).

work, consisting of 22,500 training and 1,000 test examples, each pairing a regular expression with its ground-truth DFA verified programmatically (Appendix C details dataset, tiers, and formats). We construct this dataset rather than reusing the diagnostic benchmark from Sections 2–3, which contains too few examples for fine-tuning and lacks structured difficulty tiers. Unlike Section 3, the mitigation pipeline uses the complete formal construction procedure, enabling scalable generation of verified training examples with full intermediate traces. The dataset is partitioned into four tiers: Tier 1 (symbol mapping, 2-state DFAs), Tier 2 (concatenation, 3–4 states), Tier 3 (alternation, 4–6 states), and Tier 4 (full subset construction, 6+ states). We focus on Tier 4 as the only tier requiring end-to-end NFA-to-DFA subset construction, where a single error cascades through all downstream states, making it the most discriminating indicator of genuine procedural competence. Each model is evaluated every 200 training steps on 400 held-out examples (200 in-distribution, 200 out-of-distribution), distinguishing genuine learning from memorization. For training details, see Appendix D.

4.1 Prompting and Fine-Tuning

Zero-shot and Few-shot Baselines. Zero-shot and few-shot prompting yield near-zero accuracy across all model sizes and tiers (see Appendix F), establishing that fine-tuning is necessary.

Supervised Fine-Tuning. We fine-tune all three model sizes under two supervision formats differing only in output: CoT, where each example includes full intermediate construction steps before the final DFA, and No-CoT, where the model directly produces the final DFA. Models are fine-tuned via LoRA (Hu et al., 2022) with base weights frozen (full hyperparameters in Appendix D).

Results. Both CoT and No-CoT fine-tuning recover dramatically from the 0% prompting baseline, achieving near-perfect accuracy on Tiers 1–3

Training	Model	Overall	T1	T2	T3	T4
CoT SFT	1.5B	96.0%	100%	100%	100%	80.5%
	7B	96.5%	100%	100%	100%	82.9%
	14B	96.3%	100%	100%	100%	82.1%
No-CoT SFT	1.5B	95.0%	100%	100%	96.3%	92.0%
	7B	98.0%	100%	100%	100%	95.4%
	14B	98.0%	100%	100%	97.5%	97.7%

Table 2: Accuracy across training conditions and tiers.

Condition	Overall	T4
Nat.: 1→2→3→4→5	43.5%	39.0%
Rev.: 5→4→3→2→1	26.0%	29.3%
Hard-first: 5→1→2→3→4	39.0%	34.1%
Mid-out: 3→2→4→1→5	43.5%	39.0%
Random: 2→5→1→4→3	37.0%	31.7%

Table 3: Phase-wise curriculum results (7B).

across all model sizes. The divergence appears exclusively at Tier 4: CoT accuracy is 80.5%, 82.9%, and 82.1% at 1.5B, 7B, and 14B, nearly flat across a $10\times$ increase in scale, suggesting a ceiling that capacity alone cannot break. No-CoT improves consistently to 92.0%, 95.4%, and 97.7% with no sign of saturation. This scale-invariant CoT ceiling suggests that additional parameters are directed toward generating coherent reasoning traces rather than improving symbolic execution.

Curriculum Learning. Fine-tuning recovers performance substantially, yet Tier 4 gaps persist. We investigate whether restructuring training order can help through phase-wise curriculum learning (Bengio et al., 2009), dividing training into five phases progressing from Tier 1 to Tier 4 while revisiting simpler distributions to prevent catastrophic forgetting. Experiments are conducted on the 7B model as a representative middle point of our scaling ladder. As Table 3 shows, no phase ordering approaches joint training performance. The best ordering (natural easy-to-hard) achieves 43.5% overall and 39.0% on Tier 4, far below joint CoT SFT (96.5%, 82.9%). The worst (reverse hard-first) collapses to 26.0%, confirming that early exposure to hard examples is detrimental regardless of what follows. The Tier 4 gap persists regardless of training order, motivating a deeper investigation into the model internals. Curriculum details: Appendix C.

4.2 Mechanistic Analysis

Tier 4 gaps persist regardless of supervision format or training order. This raises a concrete question: *why does the model still fail on Tier 4, even after seeing thousands of correct examples during training?* We investigate by looking inside the model.

What are we looking for? Consider a concrete example. Given the regex (ab)*c, a model must track, at every step, which NFA states are simultaneously active; this is exactly what subset construction does. If the model has genuinely learned this procedure, we would expect to find specific internal components, namely neurons or layers, that "light up" consistently whenever subset construction is being performed, regardless of the surface details of the regex. If no such components exist, the model is not executing the procedure in any localized, reliable way. We test this directly.

Is there a part of the model responsible for subset construction? We examine the activations of individual neurons across all 28 transformer layers of the CoT-trained 7B model, evaluated on 285 Tier 4 test examples (baseline accuracy 85.3%). For each of the 56,000 neuron–layer pairs sampled (2,000 neurons per layer from the 18,944-dimensional MLP activations), we ask: Does this neuron fire more strongly when the model produces a correct DFA than when it fails? (Li et al., 2026) We call this Activation Difference Scoring (ADS).

Finding neurons that correlate with correctness, however, is not enough. A neuron might fire strongly on correct outputs simply because correct outputs tend to be longer, or more structured, or involve a particular alphabet. To isolate neurons that specifically encode subset construction, and not just surface properties of the task, we test each neuron against three carefully designed contrast inputs: (i) **V1**: The same DFA, but with alphabet renamed (e.g., a→0, b→1). A neuron encoding the construction procedure should still fire strongly here, since the underlying task is identical. (ii) **V2**: A different regex that produces a structurally isomorphic DFA. Again, a construction-specific neuron should fire strongly. (iii) **V3**: A regex that *looks* complex but actually produces a trivial 2-state DFA, a decoy. A neuron genuinely encoding subset construction should fire weakly here, since despite surface complexity the regex collapses to a trivial 2-state DFA upon minimisation. A neuron that is truly tracking subset construction should score high on V1 and V2 but low on V3. We capture this with a specificity score: $\text{Specificity}(l, i) = \frac{\text{V1}+\text{V2}}{2} - \text{V3}$.

The results are clear: **no individual neuron encodes the subset construction procedure.** The most specific neurons concentrate in Layer 27, but even the best candidate neuron fires at 91% of its peak activation on the decoy input V3, mean-

Condition	Tier 4 Acc.	Δ	Level
Baseline (CoT 7B)	85.3%	—	—
VGNS (4-round) [ours]	87.7%	+2.4	Combined
SADI	86.3%	+1.0	Representation
Good neurons (×1.5)	86.0%	+0.7	Neuron
Functional antagonism	86.0%	+0.7	Neuron
ActAdd	85.6%	+0.3	Representation
Bad neurons (×0.5)	85.6%	+0.3	Neuron
Random neurons (control)	85.6%	+0.3	Control
Probe direction	84.9%	−0.4	Representation

Table 4: Steering intervention results on Tier 4.

ing it cannot distinguish "I am performing subset construction" from "this input looks complicated." Across all 56,000 candidates, the V3 bleed ratio ranges from 0.73 to 0.91. The subset construction computation is distributed across many neurons, with no small, identifiable set responsible for it.

Where in the model is the outcome decided? Since no individual neuron is in charge, we zoom out and ask a different question: at what point in the model's 28 layers does it effectively "commit" to a correct or incorrect answer? To test this, we train a linear classifier at each layer to predict, using only the model's internal state at that layer, the validity of the final output.

The answer is striking. **By Layer 10, before the model has completed even half of its computation, the internal state already predicts the final outcome with 87.1% accuracy.** The remaining 18 layers, where the actual DFA construction mechanics appear to run (layers 20–26 show denser concentrations of construction-relevant neurons, peaking at Layer 27), are largely executing on a trajectory already set. Think of it like a race where the winner is effectively decided at the halfway mark; the second half still runs, but the result rarely changes. This reveals the structure of the failure: errors are not introduced late in the computation, during the symbolic execution steps. They are *already present* in the model's early representations, before execution even begins. This raises a natural follow-up question: If the outcome is encoded this early, can we intervene on those early representations, without any retraining, to steer the model toward correct answers? **Can we fix failures by intervening at inference time?** We evaluate a range of targeted interventions at inference time, testing at two levels: individual neurons and full layer representations. All results are in Table 4. (i) *Neuron-level interventions.* Amplifying top correct neurons by $1.5\times$ gains $+0.7$ pp; suppressing failure-correlated neurons to $0.5\times$ gains $+0.3$ pp; applying both yields only $+0.7$ pp, meaning suppression adds nothing. Applying the same operations to randomly chosen neurons yields the same $+0.3$ pp. Targeted interventions provide only 0.4 pp over random manipulation, confirming that no neuron specifically encodes the subset construction procedure and neuron-level manipulation cannot provide meaningful causal control. (ii) *Representation-level steering.* Since individual neurons lack causal leverage, we move to the full hidden state at Layer 10, where 87.1% of outcome information is linearly encoded. The simplest approach, ActAdd (Turner et al., 2023), adds a fixed vector pointing from "incorrect" toward "correct" outputs and yields only $+0.3$ pp. This fails because incorrect examples do not form a single compact cluster in representation space; a uniform global shift cannot address their diverse failure modes. A more principled approach, steering along the direction a linear probe learned to distinguish correct from incorrect outputs, actually *hurts* performance (-0.4 pp). This reveals a subtlety: The direction that best *predicts* correctness from hidden states is not the same as the direction that *causes* correctness. Correlation in representation space does not imply causal control. Finally, SADI (Wang et al., 2025), which sparsifies the steering vector and scales intervention strength adaptively per example, achieves the best single-pass result at $+1.0$ pp (86.3%), yet this still recovers less than 10% of the total gap. (iii) VGNS: Verification-Guided Neuron Steering. Since single-pass interventions consistently fall short, we ask whether *iterative* escalation can do better: if a gentle nudge does not correct the model, try a stronger one. Our proposed **VGNS** framework applies progressively stronger interventions to examples that remain unsolved after each round, starting with no intervention, then functional antagonism, then SADI, then both at $2\times$ strength, reaching **87.7%** ($+2.4$ pp), the best result across all conditions. Yet 35 examples fail all four rounds. These are not borderline cases but problems for which the model never learned the required representations; no inference-time intervention can supply what was never learned. The VGNS ceiling is therefore not a ceiling on inference-time methods in general, but on what they can recover when weight-level knowledge is absent.

Concluding Remarks. Symbolic reasoning failures in LLMs are structural and partly remediable: fine-tuning helps simpler tiers, but highest-complexity failures persist due to training limits.

5 Limitations

Scope of Formal Languages. This work focuses exclusively on regular languages — the lowest level of the Chomsky hierarchy. Regular languages are well-suited as a diagnostic testbed precisely because their operators carry unambiguous semantics and correctness can be verified exactly. However, the failure modes we identify — including nullability misinterpretation, boundary condition violations, and derivative instability — may manifest differently on context-free or context-sensitive languages where structural complexity is substantially higher. Our findings should not be interpreted as general statements about LLM symbolic reasoning across all formal language classes, and extending this diagnostic framework to higher levels of the Chomsky hierarchy remains an important direction for future work.

Model Coverage in Mitigation. Our mitigation experiments are conducted exclusively on the Qwen 2.5-Instruct family at three scales (1.5B, 7B, 14B). While this provides a clean and reproducible scaling ladder, it represents a single model family with a shared pretraining distribution, tokenisation scheme, and instruction-tuning methodology. It is possible that other architectures — such as decoder-only models with different attention mechanisms, or models with significantly larger parameter counts — respond differently to the same supervision formats. In particular, our finding that CoT fine-tuning hits a scale-invariant Tier 4 ceiling may not generalise to all architectures, and the mechanistic findings specific to Qwen 2.5-7B's layer structure may not transfer to models with different depth or width configurations. Cross-architectural replication is needed before these conclusions can be treated as universal.

Benchmark Distribution and Generalisation. The regex-to-DFA fine-tuning benchmark introduced in this work is generated programmatically from a specific distribution of regular expressions controlled by tier-level operator weights, recursion depth limits, and state count bounds. While we design this distribution to cover diverse structural patterns across four difficulty tiers, and supplement it with 44 handcrafted examples covering canonical textbook patterns, the distribution is not exhaustive. Models fine-tuned on this benchmark may exploit distributional regularities — such as characteristic regex lengths, operator frequencies, or DFA topology patterns — rather than acquiring fully general procedural competence. Our out-of-distribution evaluation on 200 held-out OOD examples partially addresses this concern, but cannot rule out subtler forms of overfitting to the generation process itself. Evaluation on independently constructed benchmarks with different structural biases would strengthen the generalisation claims.

Mechanistic Analysis Scope. Our mechanistic analysis is conducted on a single model (CoT-trained Qwen 2.5-7B) on a single task (Tier 4 DFA construction) using 285 held-out test examples. While the findings — distributed representation of subset construction, Layer 10 outcome encoding at 87.1% predictive accuracy, and Layer 27 as the execution peak hosting 22 of the top 25 ADS neurons — are internally consistent and supported by multiple independent analyses (ADS scoring, ADFA variant contrast, and representation probing), they reflect the internal organisation of one specific model under one specific training regime. We cannot determine whether these structural properties are shared across other model sizes, architectures, or training formats without repeating the full mechanistic pipeline. Additionally, our analysis identifies *where* failures are encoded but does not establish *why* — whether the root cause lies in the transformer attention mechanism, the MLP structure, the pretraining distribution, or the absence of formal inductive biases during training. Disentangling these factors requires controlled architectural ablations that are beyond the scope of this work.

Inference-Time Steering Ceiling. Our steering interventions, including VGNS, recover at most 2.4 percentage points of the Tier 4 gap, leaving 35 examples permanently unsolvable across all four intervention rounds. While we evaluate a comprehensive set of intervention strategies — neuron amplification ($\times 1.5$), suppression ($\times 0.5$), ActAdd, probe direction steering, SADI adaptive steering, and multi-round VGNS — these represent a specific subset of possible inference-time approaches. Methods such as test-time compute scaling, chain-of-thought with verifier feedback, or constraint injection at the decoding level are not evaluated here and may recover additional performance. The ceiling we report should therefore be interpreted as a ceiling for the class of interventions we test, not as an absolute upper bound on inference-time recovery.

Diagnostic Coverage of Frontier Models. The cross-consistency protocol and DFA construction evaluation are applied to GPT-5.2, Grok-4.1, and Gemini-2.5 purely as diagnostic tools. We do not conduct fine-tuning or mechanistic analysis on these models due to the absence of open weights and the prohibitive cost of fine-tuning via API. As a result, the diagnostic and mitigation components of this paper use different model families: frontier models establish the breadth and universality of the failure modes, while Qwen 2.5 serves as the controlled testbed for mitigation. While we argue this separation is justified — the eleven failure modes and their ordering by frequency are consistent across all evaluated models, including both frontier and open-source families — it means we cannot directly verify whether the mechanistic findings from Qwen 2.5 transfer to frontier model internals.

Evolving Model Behavior. Our diagnostic evaluations of frontier models (GPT–5.2, Grok–4.1, and Gemini–2.5) were conducted at a specific point in time. Like many large language models, these systems undergo continuous updates. Based on our experience, performance on individual examples can improve substantially between versions as specific bugs are patched, even while the broader structural failure modes identified in this work persist across architectures and scales. This dynamic nature means that the concrete error instances reported here for frontier models may no longer appear in the exact same form in future releases. As Edsger Dijkstra famously noted, "Program testing can be used to show the presence of bugs, but never to show their absence." Consequently, our findings should be interpreted as evidence of systematic weaknesses at the time of evaluation rather than permanent, unfixable defects of specific model snapshots.

Limitations and Scope of Evaluation. A limitation of our work is that we have not tested new, proprietary, math-oriented models such as Gemini DeepThink (Luong and Lockhart, 2025; Woodruff et al., 2026), which are not readily available to the general public. Rather, our study evaluates whether current-generation models demonstrate rule-governed reasoning over formal-language concepts or if their outputs stem from pattern completion through extensive pretraining. For heavily pretrained closed models, achieving high accuracy on complex instances is plausible, but interpretation remains challenging. A correct answer does not confirm that a model has used the definition of the Kleene star or applied Brzozowski derivatives appropriately; it may simply indicate statistical familiarity with similar mathematical constructs. Notably, even Claude Sonnet 4.6 showed significant reasoning errors in limited testing (jumbled reasoning). While its inclusion could have been valuable, sustained evaluation was impractical due to API costs. By concentrating on transparent open models, this study offers a more controlled framework for assessing the limits of formal reasoning and whether models can follow the logical structure of automata-theoretic procedures from first principles instead of merely mimicking surfacial patterns.

6 Ethical Considerations

Data Ethics and Content Safety

This work examines the formal reasoning abilities of large language models using exclusively synthetic or researcher-authored formal structures: regular expressions, deterministic finite automata, and mechanistic analysis contrast examples. No human-generated text, personal data, or sensitive information appears in either the diagnostic benchmark or the fine-tuning datasets. The 180-problem diagnostic benchmark from Shelat et al. (2026) contains only formal task descriptions authored by its creators and does not include personal, demographic, or user-submitted material. Our regex-to-DFA fine-tuning benchmark and curriculum phase datasets are fully synthetic, produced by an automated symbolic generator that samples regular expressions and constructs canonical DFAs via Thompson's construction, subset construction, and Hopcroft's minimisation. There are no concerns regarding privacy, consent, copyright, or data provenance for any dataset used in this work.

Model Training, Safety, and Bias

All fine-tuning experiments use the openly available Qwen 2.5-Instruct family at 1.5B, 7B, and 14B parameters, and the training corpus consists solely of deterministic, machine-generated formal structures. Because the underlying tasks involve only symbolic reasoning over regular languages, the models are not exposed to harmful, biased, or offensive content at any stage of training. The models cannot memorise or leak private information, as none is present in the training data. The tasks studied here — regex interpretation, DFA construction,

subset construction, and string membership — are strictly formal in nature. We emphasise that the resulting fine-tuned models should not be viewed as substitutes for certified formal verification systems. Although improvements in symbolic reasoning capability may support future tooling, our models do not guarantee soundness or completeness beyond the experimental domain studied in this paper.

Human Oversight and Evaluation Fairness

Human evaluation in this work is limited to two components. In Section 2, annotators assess model outputs across three tasks of the Cross-Consistency Protocol for semantic correctness and cross-task consistency. In Section 3, annotators classify incorrect DFA outputs according to the failure mode taxonomy (P1–P6), with access to the ground-truth DFA and regex for reference. In both cases, all annotators were fully blinded to model identity and output origin, reducing potential evaluator bias. Evaluation criteria are objective and rooted in formal correctness: string membership, nullability, operator scope, transition correctness, and consistency between tasks. This eliminates subjective interpretation and ensures that reported error rates and failure mode frequencies faithfully reflect actual reasoning failures rather than annotator preference. Disagreements were resolved through adjudication by a senior reviewer in Section 2, and by discussion among annotators in Section 3. All Section 4 evaluations are fully automatic and require no human judgment.

Environmental Impact

The environmental impact of this work is limited. All fine-tuning experiments use parameter-efficient adaptation via LoRA, which updates only small low-rank matrices rather than full model weights, substantially reducing memory and compute requirements relative to full fine-tuning. All training jobs run on single NVIDIA H100 80GB GPU nodes with a maximum wall time of 48 hours per job. Frontier model evaluations (GPT-5.2, Grok-4.1, Gemini-2.5) are conducted via API, incurring no local compute cost. The mechanistic analysis and steering evaluation scripts operate on the already-trained 7B model with no additional gradient computation. Overall, the carbon footprint of this work is limited compared to large-scale pretraining or reinforcement learning pipelines.

Transparency, Reproducibility, and Research Integrity

We place strong emphasis on transparency and reproducibility. All code used for dataset generation, model training, inference, Tree-of-Thought evaluation, mechanistic analysis, steering vector extraction, and DFA correctness verification is released in full in Appendix G and in the artifact repository. We additionally provide all prompting templates, training hyperparameters, SLURM scripts, and data-processing pipelines, enabling other researchers to replicate every experiment exactly. Since all datasets used or produced in this work are either publicly available or fully synthetic, there are no legal or ethical barriers to releasing complete research artifacts. This commitment to openness supports responsible model development and promotes methodological clarity within the study of symbolic reasoning in large language models.

Use of AI Assistance

In accordance with responsible research guidelines, we note that AI tools were used only for limited and appropriate purposes. Specifically, AI assistants were employed to help with routine tasks such as code refactoring, formatting of LaTeX sections, and proofreading of non-scientific text. All conceptual contributions, theoretical formulations, experimental design decisions, dataset construction logic, mechanistic analysis methodology, and substantive results originate entirely from the authors. No AI system generated or influenced the scientific ideas, analyses, or findings presented in the paper.

In summary, this research presents minimal ethical risk. It operates entirely within controlled symbolic environments, involves no human data, employs energy-efficient parameter-efficient training, and is designed to maximise transparency and reproducibility. The work advances understanding of formal reasoning in large language models without introducing new safety, privacy, or societal concerns.

References

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward William Ayers, Dragomir Radev, and Jeremy Avigad. 2023. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *Preprint*, arXiv:2302.12433v1. ArXiv preprint.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 41–48. ACM.

François Chollet. 2019. On the measure of intelligence. *Preprint*, arXiv:1911.01547v2. ArXiv preprint.

Elliot Glazer, Ege Erdil, Tamay Besiroglu, Diego Chicharro, Evan Chen, Alex Gunning, Caroline Falkman Olsson, Jean-Stanislas Denain, Anson Ho, Emily de Oliveira Santos, Olli Järviniemi, Matthew Barnett, Robert Sandler, Matej Vrzala, Jaime Sevilla, Qiuyu Ren, Elizabeth Pratt, Lionel Levine, Grant Barkley, and 5 others. 2024. Frontiermath: A benchmark for evaluating advanced mathematical reasoning in ai. *Preprint*, arXiv:2411.04872v1. ArXiv preprint.

Matei A. Golesteanu, Garrett B. Vowinkel, and Ryan E. Dougherty. 2024. Can chatgpt pass a theory of computing course? In *Proceedings of the 2024 ACM Virtual Global Computing Education Conference*, pages 33–38, Virtual Event, USA. Association for Computing Machinery.

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. *International Conference on Learning Representations*.

Anish R Joishy, Ishwar B Balappanawar, Vamshi Krishna Bonagiri, Manas Gaur, Krishnaprasad Thirunarayan, and Ponnurangam Kumaraguru. 2026. Flying pigs, far and beyond: Evaluating llm reasoning in counterfactual worlds. *Preprint*, arXiv:2505.22318.

Adam Tauman Kalai, Ofir Nachum, Santosh S. Vempala, and Edwin Zhang. 2025. Why language models hallucinate. *Preprint*, arXiv:2509.04664v1. ArXiv preprint.

Michael Katz, Harsha Kokel, and Sarath Sreedharan. 2025. Seemingly simple planning problems are computationally challenging: The countdown game. *Preprint*, arXiv:2508.02900v1. ArXiv preprint.

Mehran Kazemi, Bahare Fatemi, Hritik Bansal, John Palowitch, Chrysovalantis Anastasiou, Sanket Vaibhav Mehta, Lalit K. Jain, Virginia Aglietti, Disha Jindal, Peter Chen, Nishanth Dikkala, Gladys Tyen, Xin Liu, Uri Shalit, Silvia Chiappa, Kate Olszewska, Yi Tay, Vinh Q. Tran, Quoc V. Le, and Orhan Firat. 2025. Big-bench extra hard. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 26473–26501, Vienna, Austria. Association for Computational Linguistics.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. 2022. Solving quantitative reasoning problems with language models. In *Advances in Neural Information Processing Systems 35*, pages 3843–3857. Curran Associates, Inc.

Wenjie Li, Guansong Pang, Hezhe Qiao, Debin Gao, and David Lo. 2026. Identifying good and bad neurons for task-level controllable LLMs. *Preprint*, arXiv:2601.04548.

Thang Luong and Edward Lockhart. 2025. Advanced version of gemini with deep think officially achieves gold-medal standard at the international mathematical olympiad. Google DeepMind Blog.

Seyed Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2025. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. In *Proceedings of the 13th International Conference on Learning Representations*, Virtual Event.

Melanie Mitchell and David C. Krakauer. 2023. The debate over understanding in ai's large language models. *Proceedings of the National Academy of Sciences*, 120(13):e2215907120.

Qwen Team. 2025. Qwen2.5 technical report. Technical report, Alibaba Group.

Shlok Shelat, Jay Raval, Souvik Roy, and Manas Gaur. 2026. Beyond memorization: Testing llm reasoning on unseen theory of computation tasks. *Preprint*, arXiv:2601.13392v1. ArXiv preprint.

Michael Sipser. 2013. *Introduction to the Theory of Computation*, 3rd edition. Cengage Learning, Boston, MA, USA.

Alexander Matt Turner, Lisa Thiergart, Gavin Leech, David Udell, Juan J. Vazquez, Ulisse Mini, and Monte MacDiarmid. 2023. Activation addition: Steering language models without optimization. *Preprint*, arXiv:2308.10248.

Weixuan Wang, Jingyuan Yang, and Wei Peng. 2025. Semantics-adaptive activation intervention for LLMs via dynamic steering vectors. In *The Thirteenth International Conference on Learning Representations*.

Sean Welleck, Jiacheng Liu, Ronan Le Bras, Hanna Hajishirzi, Yejin Choi, and Kyunghyun Cho. 2021. Naturalproofs: Mathematical theorem proving in natural language. In *Proceedings of the Thirty-Fifth Conference on Neural Information Processing Systems, Datasets and Benchmarks Track (Round 1)*, Virtual Event.

David P. Woodruff, Vincent Cohen-Addad, Lalit Jain, Jieming Mao, Song Zuo, MohammadHossein Bateni, Simina Branzei, Michael P. Brenner, Lin Chen, Ying Feng, Lance Fortnow, Gang Fu, Ziyi Guan, Zahra Hadizadeh, Mohammad T. Hajiaghayi, Mahdi JafariRaviz, Adel Javanmard, Karthik C. S., Ken ichi Kawarabayashi, and 17 others. 2026. Accelerating

scientific research with gemini: Case studies and common techniques. *Preprint*, arXiv:2602.03837.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik R. Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, volume 37, pages 21444–21460. NeurIPS 2023 Conference Paper.

Xiang Yue, Yuansheng Ni, Kai Zhang, Tianyu Zheng, Ruoqi Liu, Ge Zhang, Samuel Stevens, Dongfu Jiang, Weiming Ren, Yuxuan Sun, Cong Wei, Botao Yu, Ruibin Yuan, Renliang Sun, Ming Yin, Boyuan Zheng, Zhenzhu Yang, Yibo Liu, Wenhao Huang, and 3 others. 2024. Mmmu: A massive multi-discipline multimodal understanding and reasoning benchmark for expert agi. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9556–9567, Vancouver, Canada. IEEE/CVF.

Pardis Sadat Zahraei and Ehsaneddin Asgari. 2024. Turingq: Benchmarking ai comprehension in theory of computation. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 12267–12280, Miami, Florida, USA. Association for Computational Linguistics.

A Formal Background

This appendix summarizes the formal machinery underlying regular expressions, deterministic finite automata, and the procedural constructions used across this work. Definitions in A.1–A.4 underpin the diagnostic evaluation in Sections 2 and 3; definitions in A.5–A.7 underpin the fine-tuning pipeline in Section 4. All definitions follow Sipser (2013).

A.1 Regular Expressions

Let Σ be a finite alphabet. The set of *regular expressions* over Σ is defined inductively:

- $\emptyset$ denotes the empty language.
- ε denotes the language $\{\varepsilon\}$.
- For each $a \in \Sigma$, the symbol a denotes $\{a\}$.
- If R and S are regular expressions, then so are:
  - (RS) denoting concatenation $L(R) \cdot L(S)$,
  - $(R \mid S)$ denoting union $L(R) \cup L(S)$,
  - (R^*) denoting Kleene closure $\bigcup_{i \geq 0} L(R)^i$.

These operators carry unambiguous formal semantics, eliminating interpretive ambiguity and making it possible to evaluate correctness exactly. This property is what makes regular languages an ideal testbed for probing symbolic reasoning — there is no room for debate about whether a string belongs to the language.

Operator Precedence. By convention, Kleene star binds most tightly, followed by concatenation, followed by union. Thus $ab^* \mid c$ parses as $(a(b^*)) \mid c$. Scope and precedence confusion is one of the eleven failure modes documented in Section 2.

Example. For $R = (a \mid b)^*ab$, the language $L(R)$ consists of all strings over $\{a, b\}$ ending with `ab`. For instance: `ab`, `bab`, `aaab` are in $L(R)$, whereas `aba`, `b`, and ε are not.

A.2 Deterministic Finite Automata

A *deterministic finite automaton* (DFA) is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of states,
- Σ is a finite input alphabet,
- $\delta : Q \times \Sigma \to Q$ is the total transition function,
- $q_0 \in Q$ is the unique start state,
- $F \subseteq Q$ is the set of accepting states.

The DFA processes a string $w = w_1 w_2 \cdots w_n$ by starting in q_0 and applying δ for each symbol in sequence. It *accepts* w if the final state reached

is in F, and *rejects* otherwise. Because δ is total, every string has a unique run. A DFA is *complete* if $\delta(q, a)$ is defined for every state q and symbol a; completeness is enforced in our benchmark by adding an explicit dead (trap) state for all undefined transitions.

Example. The following DFA recognizes all strings over $\{a, b\}$ ending with ab:

	a	b
$\rightarrow q_0$	q_1	q_0
q_1	q_1	q_2
$*q_2$	q_1	q_0

State q_0 is the start state (no relevant suffix seen), q_1 records that the last symbol was a, and q_2 records that the last two symbols were ab and is the sole accepting state.

A.3 Nullability

The *nullability* predicate $\mathsf{Null}(R)$ determines whether $\varepsilon \in L(R)$, i.e., whether the empty string is accepted by R:

$$\begin{aligned}
\mathsf{Null}(\emptyset) &= \text{false} \\
\mathsf{Null}(\varepsilon) &= \text{true} \\
\mathsf{Null}(a) &= \text{false} \quad \text{for any } a \in \Sigma \\
\mathsf{Null}(RS) &= \mathsf{Null}(R) \wedge \mathsf{Null}(S) \\
\mathsf{Null}(R \mid S) &= \mathsf{Null}(R) \vee \mathsf{Null}(S) \\
\mathsf{Null}(R^*) &= \text{true}
\end{aligned}$$

Nullability is critical in two places in this paper. In Section 2, models frequently confuse nullable and non-nullable subexpressions — treating a^* (nullable) as mandatory or a^+ (non-nullable) as optional, a failure documented as *Nullability Neglect*. In Section 3, nullability propagation is a required step in Brzozowski derivative computation (Section A.4), and failures to propagate it correctly are documented as Problem 2 in DFA construction.

A.4 Brzozowski Derivatives

The *Brzozowski derivative* of a regular expression R with respect to a symbol $a \in \Sigma$ is:

$$D_a(R) = \{\, w \in \Sigma^* : aw \in L(R) \,\}.$$

Intuitively, $D_a(R)$ is the residual language after consuming the first symbol a from any string in $L(R)$. It is itself a regular expression, defined inductively:

$$\begin{aligned}
D_a(\emptyset) &= \emptyset \\
D_a(\varepsilon) &= \emptyset \\
D_a(b) &= \begin{cases} \varepsilon & b = a \\ \emptyset & b \neq a \end{cases} \\
D_a(RS) &= \begin{cases} D_a(R)S \mid D_a(S) & \text{if } \mathsf{Null}(R) \\ D_a(R)S & \text{otherwise} \end{cases} \\
D_a(R \mid S) &= D_a(R) \mid D_a(S) \\
D_a(R^*) &= D_a(R)R^*
\end{aligned}$$

The case for concatenation RS is the most subtle: if R is nullable, then a string beginning with a may match R as the empty string and then have a consumed by S, contributing $D_a(S)$ to the residual. If R is not nullable, the first symbol must be consumed by R alone. This is precisely where LLMs fail in Section 3 (Problem 2): they replace the nullable ε contribution with the universal language $(a \mid b)^*$, producing an overly broad residual that accepts any subsequent input and erases the original constraint.

Derivative-based DFA construction. Starting from a regex R, one can construct a DFA whose states are the distinct derivatives of R:

$$\begin{aligned}
Q &= \{\text{all distinct derivatives of } R\}, \\
q_0 &= R, \\
F &= \{D \in Q : \mathsf{Null}(D)\}, \\
\delta(D, a) &= D_a(D).
\end{aligned}$$

This construction terminates because regular languages have finitely many distinct left quotients. In Section 3, models are asked to follow this procedure under Tree-of-Thought prompting as a diagnostic probe of algebraic manipulation capability.

Example. For $R = ab$ over $\Sigma = \{a, b\}$:

$$\begin{aligned}
D_a(ab) &= D_a(a) \cdot b = \varepsilon \cdot b = b, \\
D_b(ab) &= D_b(a) \cdot b = \emptyset \cdot b = \emptyset.
\end{aligned}$$

Then $D_b(b) = \varepsilon$ (nullable, so accepting) and $D_a(b) = \emptyset$. The derivative DFA has states $\{ab, b, \varepsilon, \emptyset\}$ corresponding to positions in parsing the string, with ε as the sole accepting state.

A.5 Thompson's NFA Construction

Thompson's construction converts a regular expression into an NFA by structural induction. An NFA

extends the DFA by allowing ε-transitions and multiple transitions on the same symbol. Each construction step produces a *fragment* with exactly one start state and one accept state.

Base cases.

- **Symbol** a: two states s and t with $s \xrightarrow{a} t$.
- **Epsilon** ε: two states s and t with $s \xrightarrow{\varepsilon} t$.

Inductive cases.

- **Concatenation** RS: connect the accept state of $N(R)$ to the start state of $N(S)$ via an ε-transition. New start: $N(R)$'s start. New accept: $N(S)$'s accept.
- **Union** $R \mid S$: new start state s with ε-transitions to both $N(R)$'s and $N(S)$'s start states; new accept state t reached from both their accept states via ε-transitions.
- **Kleene star** R^*: new start s and accept t; add ε-transitions $s \to N(R)$'s start, $s \to t$ (skip entirely), $N(R)$'s accept $\to$ $N(R)$'s start (loop), and $N(R)$'s accept $\to t$ (exit).

Example. For $R = ab$, Thompson's construction produces:

$$s_0 \xrightarrow{a} s_1 \xrightarrow{\varepsilon} s_2 \xrightarrow{b} s_3$$

where s_0 is the start state and s_3 is the accept state. Each training example in the CoT fine-tuning dataset (Section 4) records every Thompson step as a separate JSON object in the construction trace, supervising the model to execute this procedure symbol by symbol.

A.6 ε-Closure and Subset Construction

ε-closure. The ε-closure of a set of NFA states S is the set of all states reachable from any state in S via zero or more ε-transitions:

$$\varepsilon\text{-closure}(S) = \{q \mid q \text{ reachable from some } s \in S \text{ via } \varepsilon\text{-transitions alone}\}$$

Subset construction. Subset construction converts an NFA into an equivalent DFA by treating sets of NFA states as single DFA states:

1. DFA start state: $D_0 = \varepsilon\text{-closure}(\{q_0\})$.
2. For each DFA state D and symbol a: $\delta_{\text{DFA}}(D, a) = \varepsilon\text{-closure}\Big(\bigcup_{q \in D} \delta_{\text{NFA}}(q, a)\Big)$.
3. D is accepting iff $D \cap F_{\text{NFA}} \neq \emptyset$.
4. Repeat until no new DFA states are generated.
5. Add a dead state for any undefined transitions.

Subset construction is the central procedural step in our fine-tuning pipeline and the defining challenge of Tier 4 in our benchmark, where each DFA state is a set of NFA states computed via reachability. A single error in one DFA state cascades through all downstream states derived from it, which is why Tier 4 is the most discriminating indicator of genuine procedural competence.

Example. For the NFA for $R = ab$ with states s_0, s_1, s_2, s_3:

- $D_0 = \varepsilon\text{-closure}(\{s_0\}) = \{s_0\}$. On a: $\{s_1\}$, ε-closure $= \{s_1, s_2\} = D_1$. On b: $\emptyset \to D_{\text{dead}}$.
- $D_1 = \{s_1, s_2\}$. On b: $\{s_3\}$, ε-closure $= \{s_3\} = D_2$ (accepting). On a: $\emptyset \to D_{\text{dead}}$.

The resulting DFA has 3 live states, exactly tracking position within `ab`.

A.7 Hopcroft's Minimisation Procedure

A DFA produced by subset construction may contain redundant states that are *indistinguishable*: no input string can tell them apart. Hopcroft's procedure finds and merges all such states, producing the unique minimum-state DFA for the language.

Distinguishability. States p and q are *distinguishable* if there exists a string w such that exactly one of $\delta^*(p, w)$ and $\delta^*(q, w)$ is accepting. States that are not distinguishable are *equivalent* and can be safely merged.

Procedure.

1. Initialise partition $P = \{F,\ Q \setminus F\}$.
2. Maintain a worklist $W = P$.
3. While W is non-empty, pick block A. For each $a \in \Sigma$, compute $X = \{q \in Q \mid \delta(q, a) \in A\}$. For each block B with $B \cap X \neq \emptyset$ and $B \setminus X \neq \emptyset$, split B into $B \cap X$ and $B \setminus X$, updating W.
4. Each final equivalence class becomes one state in the minimised DFA.

The minimised DFA is canonical — the unique smallest DFA accepting the language, up to state renaming. All CoT training examples in Section 4 include the full Hopcroft partition refinement trace, recording each split step as a separate JSON object.

A.8 Relevance to This Paper

Table 5 summarises which formal constructions are exercised in each part of the paper.

Construction	Used in	Role
Regular expressions (A.1)	Sections 2, 3, 4	Task inputs throughout
DFA (A.2)	Sections 2, 3, 4	Target output throughout
Nullability (A.3)	Sections 2, 3	Failure mode (Nullability Neglect, Problem 2)
Brzozowski derivatives (A.4)	Section 3	Derivative-based construction diagnostic
Thompson's construction (A.5)	Section 4	CoT trace generation pipeline
Subset construction (A.6)	Section 4	Tier 4 benchmark; CoT trace pipeline
Hopcroft minimisation (A.7)	Section 4	Final minimised DFA; CoT trace pipeline

Table 5: Formal constructions and their roles across the paper.

B Appendix B: Cross-Consistency Protocol Example

We include here the model outputs for all three tasks of the Cross-Consistency Protocol for the regular expression:

$$(b \mid a(a(bb \mid bbb)^*ba)^*b)^* \\ a(a(bb \mid bbb)^*ba)^* \\ a(bb \mid bbb)^*\, bba.$$

Task 1: Natural-Language Interpretation

The language consists of strings over $\{a, b\}$ that are built from an initial repetition of either a single b or a longer block that starts with a, ends with b, and may contain repeated internal segments. After that initial repetition, every string must contain a, then another a, and then finish with a fixed tail that ends in bba. The ending is always of the form $(bb \mid bbb)^*$ followed by bba, so every accepted string ends with at least three b's, with the last three symbols exactly bba. The middle region between the two required a's consists of zero or more blocks of the form $a(bb \mid bbb)^*ba$. These same chunk patterns may also appear inside the initial repeated part but wrapped so that the full alternative begins with a and ends with b. Because of the mandatory aa in the final concatenation, no string in the language can be empty or end in a. All occurrences of a arise from the structured block patterns rather than arbitrary placements. The final tail enforces that the last a must be followed by a sequence of bb/bbb groups and then exactly bba, forbidding endings such as $\ldots bab$ or $\ldots b$.

Task 2: Example-Based Reasoning

Task 2: Example-Based Reasoning

Accepted strings:
$\{aabba,\ aaabbba,\ baaabbba,\ bbaabbbba,\ aaababbba,\ aabbaabbbba,\ baababbba,\ abbbaabbbba,\ babaaabbba,\ ababbbaababbba\}$

Rejected strings:
$\{\ a,\ b,\ aa,\ aabb,\ abba,\ baaba,\ abab,\ bbb,\ aababa,\ babbabaab\ \}$

Task 3: Symbolic Execution on Edge Cases

- **aaababbb** $\rightarrow$ **Rejected.** The string ends in bbb, but the regex requires the final suffix to be $(bb \mid bbb)^*$ followed by bba. It cannot be decomposed to end with the mandatory bba tail.
- **babbabaab** $\rightarrow$ **Rejected.** The regex forces the string to end with bba; however, this string ends with aab, which violates the required final structure.
- **ababbbaab** $\rightarrow$ **Rejected.** Although the last symbol is b, the final three characters are aab, not bba. The required tail structure is missing.

C Dataset Details

C.1 Diagnostic Benchmark (Sections 2–3)

The diagnostic evaluation in Sections 2 and 3 uses the **180 regex-to-DFA construction problems** introduced in Shelat et al. (2026). These problems are manually curated to emphasize structural diversity and specifically target the failure modes documented in this paper. Each problem pairs a regular expression with a ground-truth DFA represented as a complete transition table, enabling exact correctness verification independent of linguistic ambiguity.

Structural Coverage. The benchmark is designed to stress-test symbolic reasoning by including expressions with:

- nested alternation and deeply scoped repetition operators,
- variable atomic block sizes that impose length constraints (e.g., $(bb \mid bbb)^*$),
- nullable vs. non-nullable subexpressions in critical positions,
- long-range concatenation constraints where boundary conditions span multiple sub-blocks,

- mandatory prefix and suffix enforcement that requires global structural awareness rather than local pattern matching.

These properties are chosen precisely because they expose the failure modes documented in Section 2 (nullability neglect, anchor hallucination, scope confusion) and Section 3 (Kleene-star misinterpretation, derivative errors, boundary loss under concatenation). A model that relies on surface-level pattern matching rather than genuine symbolic reasoning will fail systematically on this benchmark.

Ground-Truth DFAs. Each problem is annotated with the minimal ground-truth DFA, allowing exact correctness checks via language equivalence verification. The DFAs range in complexity from simple 2–3 state machines to automata with composite state encodings (e.g., $q_2q_3q_4$) arising from subset construction, making them suitable for fine-grained diagnosis across both interpretation and execution stages.

Example Benchmark Problem. The following regex appears in the benchmark and illustrates the structural complexity typical of harder problems:

$$R = (b \mid a(a(bb \mid bbb)^*ba)^*b)^* \\ a(a(bb \mid bbb)^*ba)^* \\ a(bb \mid bbb)^*\, bba.$$

The ground-truth DFA transition table for this regex is shown in Table 6, and the corresponding state diagram is shown in Figure 4. The composite state labels (e.g., $q_2q_3q_4$) reflect the subset construction origin of the DFA: each such state corresponds to a set of NFA states that must be tracked simultaneously, requiring the model to reason about reachability across multiple parallel paths.

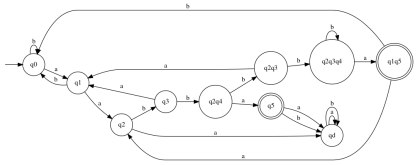


Figure 4: State diagram for the example benchmark regex R. Red transitions indicate paths to the dead state; composite state labels indicate NFA state sets from subset construction.

Usage in This Paper. This benchmark is used exclusively for controlled evaluation in Sections 2 and 3. In Section 2, each of the 180 regexes serves as input to the Cross-Consistency Protocol across three tasks. In Section 3, the same problems are used to evaluate DFA construction under both intuitive and derivative-based prompting strategies under Tree-of-Thought. The benchmark is not used for training at any point. In contrast to the fine-tuning benchmark introduced in Section 4 (which is procedurally generated at scale), these 180 problems function as a small but high-precision evaluation suite targeting specific structural failure modes and ambiguity-sensitive constructions.

State	a	b
$\rightarrow q_0$	q_1	q_0
q_1	q_2	q_0
q_2	q_d	q_3
q_3	q_1	q_2q_4
q_2q_4	q_5	q_2q_3
q_2q_3	q_1	$q_2q_3q_4$
$q_2q_3q_4$	q_1q_5	$q_2q_3q_4$
$q_1q_5^*$	q_2	q_0
q_5^*	q_d	q_d
q_d	q_d	q_d

Table 6: Ground-truth DFA transition table for the example benchmark regex R. States marked with * are accepting. q_d is the dead (trap) state. Composite state labels such as $q_2q_3q_4$ reflect the subset construction origin of the DFA.

C.2 Regex-to-DFA Fine-Tuning Benchmark (Section 4)

The mitigation experiments in Section 4 use a new regex-to-DFA benchmark introduced in this work, distinct from the diagnostic benchmark in C.1. This dataset is designed for supervised fine-tuning and provides the structured difficulty tiers required to study how mitigation effectiveness varies with task complexity. It consists of **22,500 training examples** and **1,000 test examples**, each pairing a regular expression with its ground-truth minimised DFA represented as a complete transition table, verified programmatically with no partial credit.

Generation Pipeline. Each example is generated through the following pipeline:

1. **Regex generation.** A random regular expression AST is sampled over a randomly chosen alphabet from {*binary*, *ab*, *abc*, *abcd*, *digits*}. The AST is built by a tiered random generator that controls operator distribution, recursion depth, and structural complexity according to the target tier. The generator supports all core operators (concatenation, union, Kleene star) plus syntactic sugar (plus, optional) that is desugared before construction.

2. **DFA construction.** The regex AST is converted to a minimised DFA via the full procedural pipeline: Thompson's construction (Appendix A.5) $\rightarrow$ ε-closure and subset construction (Appendix A.6) $\rightarrow$ Hopcroft's minimisation (Appendix A.7).
3. **Verification.** The NFA, pre-minimisation DFA, and minimised DFA are cross-verified on all strings up to length 7, confirming $L(\text{NFA}) = L(\text{DFA}) = L(\text{minDFA})$. Any example failing verification is discarded.
4. **Deduplication.** Examples are deduplicated by an MD5 hash of the regex string and alphabet, ensuring no two training examples represent the same language.
5. **Formatting.** Each verified example is formatted into either a CoT trace (for CoT SFT) or a direct transition table (for No-CoT SFT), as described below.

The dataset also includes a curated set of 44 handcrafted examples covering classic textbook regexes (e.g., $(a \mid b)^*abb$, $(0 \mid 1)^*101$) and known structurally interesting patterns, ensuring coverage of canonical cases that random generation may underrepresent.

Difficulty Tiers. The dataset is partitioned into four tiers of increasing structural complexity, controlled by the generator's operator weights, recursion depth limits, and minimum/maximum DFA state counts:

- **Tier 1** ($\approx$15% of data): direct symbol mapping and simple concatenation/union; produces 2-state minimised DFAs.
- **Tier 2** ($\approx$25%): introduces Kleene star, plus, and optional operators; produces small looping DFAs with 2–4 states.
- **Tier 3** ($\approx$35%): combined operators with longer concatenations and star-of-union patterns; produces mid-size DFAs with 3–6 states.
- **Tier 4** ($\approx$25%): deeply nested closures of concatenations; produces larger DFAs with 4–18 states requiring full end-to-end subset construction.

Tier 4 is the primary focus of analysis in Section 4 because it is the only tier where a single error cascades through all downstream states, making it the most discriminating indicator of genuine procedural competence versus surface pattern matching.

CoT Format. In the CoT supervision format, each example includes the full construction trace as the target output, delimited by *TRACE BEGIN* and *TRACE END* markers, followed by the final DFA delimited by *FINAL DFA BEGIN* and *FINAL DFA END* markers. The trace records every intermediate step as a structured JSON object, including:

- the original regex and its postfix token sequence,
- every Thompson construction step (symbol fragments and concatenation/union/star fragment merges),
- the complete NFA state and transition table,
- every subset construction step mapping NFA state sets to DFA states,
- every Hopcroft partition refinement step,
- the minimised DFA before and after state renaming,
- the final renamed DFA in the output transition table format.

The system prompt for CoT training instructs the model: *"You are a DFA construction assistant. You must output the TRACE and FINAL DFA exactly in the same format as training data. Do not output any explanation outside the TRACE markers."*

Sample Entry (CoT format, truncated). The following shows a truncated CoT training example for the regex $babaa(abaababa)(baaabaaa)(baaaaaab)$ over $\Sigma = \{a, b\}$, illustrating the trace structure:

```
=== TRACE BEGIN ===
### Original Regex
{"regex": "babaa(abaababa)(baaabaaa)(baaaaaab)"}

### Regex After Concat Insertion
{"regex_with_concat":
 "b.a.b.a.a.(a.b.a.a.b.a.b.a). ..."}

### Thompson Step (Symbol)
{"token": "b",
 "created_fragment": {"start": "s0",
                      "accept": ["s1"]}}

### Thompson Step (Symbol)
{"token": "a",
 "created_fragment": {"start": "s2",
                      "accept": ["s3"]}}

### Thompson Step (Concat)
{"frag1_start": "s0", "frag1_accepts": ["s1"],
 "frag2_start": "s2", "frag2_accepts": ["s3"],
 "new_start": "s0",   "new_accepts": ["s3"]}

... [Thompson steps continue] ...

### Subset Construction Step
{"dfa_state": "D0", "nfa_subset": ["s0"],
 "transitions": {"a": "D30", "b": "D1"}}

### Subset Construction Step
{"dfa_state": "D1", "nfa_subset": ["s1","s2"],
 "transitions": {"a": "D2", "b": "D30"}}

... [Subset construction steps continue] ...

### Hopcroft Refinement
{"symbol": "b",
 "split_block": ["D0","D1",...,"D9"],
 "inter": ["D0","D1",...,"D27"],
```

```
 "diff": ["D28"]}

... [Hopcroft refinement steps continue] ...

### Final Renamed DFA
{"states": ["q0","q1",...,"q30"],
 "start_state": "q0",
 "accept_states": ["q22"],
 "transitions": {"q0": {"a":["q24"],
                        "b":["q1"]}, ...}}
=== TRACE END ===

=== FINAL DFA BEGIN ===
states: q0 q1 q2 ... q30
alphabet: a b
start: q0
final: q22
transitions:
q0 a q24
q0 b q1
...
=== FINAL DFA END ===
```

No-CoT Format. In the No-CoT supervision format, each example contains only the final minimised DFA as the target output with no intermediate steps. The system prompt instructs the model: *"You are a DFA construction assistant. You must output ONLY the DFA in the exact required format. Do not provide any explanation, extra text, or commentary."*

Sample Entry (No-CoT format). The following is a representative No-CoT example for the regex 00 over $\Sigma = \{0, 1\}$:

```
Prompt:
You are given a regular expression over
alphabet  = {0, 1}. Construct a COMPLETE
deterministic finite automaton (DFA) that
recognizes EXACTLY the same language.

OUTPUT FORMAT (STRICT):
states: q0 q1 q2 ...
alphabet: 0 1
start: q0
final: qX qY ...
transitions:
q0 0 q1
q0 1 q2
...

CONSTRAINTS:
1. DFA must be complete.
2. DFA must be deterministic.
3. Use only q0,q1,q2,... naming.
4. No explanation.

REGEX: 00

Output:
states: q0 q1 q2 q3
alphabet: 0 1
start: q0
final: q2
transitions:
q0 0 q1
q0 1 q3
q1 0 q2
q1 1 q3
q2 0 q3
q2 1 q3
q3 0 q3
q3 1 q3
```

The DFA has 4 states: q_0 (start, no relevant prefix seen), q_1 (first `0` seen), q_2 (accepting, `00` matched), and q_3 (dead state for all invalid continuations).

Train/Validation/Test Splits. The dataset is split into training (90%), validation (5%), and test (5%) sets by random shuffle with seed 42. The validation set is further divided into 200 in-distribution (IID) examples drawn from the same regex generation distribution as training, and 200 out-of-distribution (OOD) examples generated with structural patterns and operator combinations not seen during training. Reporting both IID and OOD performance ensures that improvements reflect genuine procedural learning rather than memorization of training-set patterns.

C.3 Curriculum Phase Datasets (Section 4.2)

The phase-wise curriculum learning experiments in Section 4.2 use a reorganised version of the fine-tuning benchmark described in C.2, divided into five phase-specific dataset pairs (*phase1_train* through *phase5_train*, each with a corresponding validation file). These phase files are used exclusively for the curriculum experiments and are not used in the main CoT or No-CoT SFT experiments.

Phase Structure. Each phase corresponds to a tier or tier mixture of increasing complexity:

- **Phase 1**: Tier 1 examples only — simple concatenation and union, producing 2-state DFAs.
- **Phase 2**: Tier 2 examples — introduces Kleene star, plus, and optional operators.
- **Phase 3**: Tier 3 examples — combined operators with longer concatenations and branching structure.
- **Phase 4**: Balanced mixture of all tiers — designed to prevent catastrophic forgetting by reintroducing simpler distributions after harder ones have been seen.
- **Phase 5**: Tier 4 examples only — full subset construction, the hardest and most discriminating tier.

Format. Phase files use the 8-step markdown CoT format, which differs structurally from the JSON-trace CoT format used in the main CoT SFT experiments. Each example follows the Qwen ChatML message format with system, user, and assistant roles. Approximately 70% of each phase file consists of construction examples (*type: construction*) and 30% consists of theory Q&A examples (*type: theory*) covering conceptual questions about subset construction, DFA properties, nullability, and Hopcroft's procedure. The theory examples

are included to prevent the model from overfitting to construction mechanics alone and to maintain conceptual grounding throughout training. The assistant response for construction examples follows an 8-step markdown structure:

- **Step 1**: Analyse the regex and identify the alphabet.
- **Step 2**: Apply Thompson's construction rules.
- **Step 3**: NFA transition table with start and accept states.
- **Step 4**: Compute ε-closure of the start state.
- **Step 5**: Subset construction table mapping NFA state sets to DFA states.
- **Step 6**: Hopcroft minimisation with partition reduction count.
- **Step 7**: Final minimised DFA transition table.
- **Step 8**: Verification on representative accepted and rejected strings with state traces.

Sample Entry (Phase 1, curriculum format). The following shows two consecutive entries from *phase1_train.jsonl* — one construction example and one theory Q&A example — illustrating the 70/30 interleaving and the exact 8-step markdown format used throughout all phase files:


```
{"messages": [
  {"role": "system", "content": "You are an expert
     in formal
   language theory and automata. When given a
     regular expression,
   convert it to a minimised DFA using Thompson's
     construction,
   subset construction, and Hopcroft's
     minimisation. Show all
   steps."},
  {"role": "user", "content": "Use Thompson's
     construction and
   subset construction to build a minimised DFA
     for '0(0|0)|1'
   over Sigma = {0, 1}."},
  {"role": "assistant", "content":
   "## Step 1: Analyse the Regex\nRegex: 0(0|0)|1
     | Sigma={0,1}
    ## Step 2: Thompson's Construction\nBuild NFA
     using rules.
    ## Step 3: NFA Transition Table\nStart: q11 |
     Accept: q12
    | State | Symbol | Next |
    | q1 | 0 | {q2} | ... | q11 | eps | {q1, q9} |
    ## Step 4: eps-closure of Start State
    eps-closure(q11) = {q1, q9, q11} -> D0
    ## Step 5: Subset Construction
    | State | NFA States | 0 | 1 | Accept? |
    | D0 | {q1,q9,q11} | D1 | D2 |  |
    | D1 | {q2,q3,q5,q7} | D3 | empty |  |
    | D2 | {q10,q12} | empty | empty | Y |
    | D3 | {q4,q6,q8,q12} | empty | empty | Y |
    ## Step 6: Minimisation (Hopcroft's Procedure)
    Reduced 4 -> 3 states.
    ## Step 7: Minimised DFA
    States: {D0,D1,D2} | Start: D0 | Accept: {D2}
    | D0 | D1 | D2 |  |
    | D1 | D2 | -- |  |
    | D2 | -- | -- | Y |
    ## Step 8: Verification
    | \"1\" | Y | D0->D2 | | \"00\" | Y |
     D0->D1->D2 |
    | \"eps\" | N | D0 |  | \"0\" | N | D0->D1 |
    3 states | Accept: {D2} | Examples: \"1\",
     \"00\""}],
 "metadata": {"type": "construction", "phase": 1,
              "regex": "0(0|0)|1", "alphabet":
    ["0","1"],
              "min_dfa_states": 3, "hash":
    "e7bc95d8"}}

{"messages": [
  {"role": "system", "content": "You are an expert
     in formal
   language theory and automata. Explain the
     theory behind
   regular expressions and DFA construction
     clearly and
   precisely."},
  {"role": "user", "content": "How does subset
     construction
   convert an NFA to a DFA?"},
  {"role": "assistant", "content": "Subset
     construction
   converts NFA N=(Q,S,dN,q0,F) to DFA D where
     each DFA state
   represents a SET of NFA states. Steps: (1)
     Start DFA state =
   eps-closure({q0}). (2) For each DFA state S and
     symbol a,
   compute move(S,a) = eps-closure(union{dN(q,a)|q
     in S}).
   (3) A DFA state S is accepting iff S intersect
     F != empty.
   (4) Repeat until no new states. Worst case:
     2^|Q| states."}],
 "metadata": {"type": "theory", "phase": 1}}
```


The construction entry shows the full 8-step markdown CoT trace with NFA table, subset construction table, Hopcroft reduction count, minimised DFA transition table, and verification strings. The theory entry shows the shorter Q&A format used for the 30% theory examples, which share the same ChatML structure but use a different system prompt and contain no construction trace.

D Model and Training Details

D.1 Qwen 2.5 Architecture

All fine-tuning and mechanistic analysis experiments in Section 4 use the Qwen 2.5-Instruct family (Qwen Team, 2025) at three scales. The exact architectural parameters, sourced from the official model configuration files, are:

Parameter	1.5B	7B	14B
Hidden layers	28	28	48
Hidden dimension	1,536	3,584	5,120
MLP intermediate	8,960	18,944	13,824
Attention heads (Q)	12	28	40
Attention heads (KV)	2	4	8
Activation	SiLU	SiLU	SiLU

Table 7: Qwen 2.5-Instruct architecture parameters used in this work.

All models use RMSNorm for layer normalisation and Rotary Position Embeddings (RoPE). The mechanistic analysis in Section 4.3 targets the SiLU activation output (MLP intermediate activations) for ADS neuron scoring, and the full hidden state at each layer output for representation probing.

Specifically, for the 7B model: 28 layers, 18,944-dimensional MLP intermediate activations (56,000 neuron–layer pairs sampled at 2,000 per layer), and 3,584-dimensional hidden states for representation probing. All models are loaded in bfloat16 precision with automatic device mapping.

D.2 LoRA Fine-Tuning Configuration

All models are fine-tuned using parameter-efficient adaptation via LoRA (Hu et al., 2022) with base weights frozen. The following configuration is identical across all model sizes and supervision formats (CoT, No-CoT, and curriculum):

- **Rank**: $r = 32$
- **Alpha**: $\alpha = 64$
- **Dropout**: 0.05, applied to attention and MLP projections
- **Target modules**: *q_proj*, *k_proj*, *v_proj*, *o_proj*, *gate_proj*, *up_proj*, *down_proj*
- **Bias**: none
- **Task type**: causal language modelling

Optimizer and Scheduler.

- **Optimizer**: AdamW (fused implementation)
- **Learning rate**: 2×10^{-4}
- **Scheduler**: cosine decay
- **Warmup ratio**: 0.03
- **Weight decay**: 0.01
- **Epochs**: 2
- **Per-device batch size**: 1
- **Gradient accumulation steps**: 8 (effective batch size 8)
- **Precision**: bfloat16 with TF32 enabled
- **Gradient checkpointing**: enabled
- **Maximum sequence length**: 8,192 tokens

Evaluation. Evaluation is performed every 200 training steps on a held-out validation set of 400 examples (200 IID, 200 OOD). Training uses a fixed random seed of 42 with full determinism enforced across Python, NumPy, PyTorch, and CUDA.

Curriculum Training. For the phase-wise curriculum experiments, the same LoRA configuration is applied sequentially across phases with early stopping patience of 5 evaluation steps. Each phase trains on its phase-specific dataset and the trained adapter from each phase initialises the next phase, implementing the sequential fine-tuning chain described in Section 4.2.

D.3 Prompting Templates

D.3 Prompting Templates

We use three categories of prompting templates depending on the experiment. All frontier model experiments (Sections 2 and 3) use zero-shot prompting with temperature 0.0. For the baseline experiments in Section 4, we additionally evaluate zero-shot and few-shot (3 demonstrations) prompting on all Qwen 2.5 model sizes before fine-tuning; full results are reported in Appendix F. No system-level fine-tuning is applied to any frontier model.

(1) Intuitive DFA Construction. This prompt instructs the model to construct a DFA without any procedural template, as a student might in a traditional Theory of Computation course:

> *You are an expert in formal languages and finite automata. Output ONLY a valid JSON object containing the DFA transition table. No reasoning or explanations.*

The output schema fixes the fields for states, alphabet, start state, accept set, and a complete transition function. This prompt is sourced from Shelat et al. (2026).

(2) Derivative-Based Construction. This prompt enforces symbolic computation via Brzozowski derivatives, requiring each DFA state to correspond to a residual of the input regex:

> *Internally construct a DFA using Brzozowski derivatives. Each state must correspond to a residual of the regex. OUTPUT ONLY the final DFA JSON. No reasoning text.*

This tests nullability propagation, normalization of equivalent residuals, and derivative rule application. It is the prompt used for derivative-based construction in Section 3 and is sourced from Shelat et al. (2026).

(3) Cross-Consistency Protocol (Tasks 1–3). The Cross-Consistency evaluation uses a custom prompt designed specifically for this work to force structured semantic reasoning across three distinct components within a single prompt context:

> *You are required to produce output in EXACTLY THREE sections:*
>
> *=== TASK 1 ===*
> *Provide a 6–12 sentence semantic description of the language, identifying admissible prefixes, suffixes, and mandatory structural blocks.*
>
> *=== TASK 2 ===*
> *Generate exactly 10 accepted strings and 10 rejected strings. Ensure the accepted strings span structurally distinct branches of the language.*

=== TASK 3 ===
For each of the following three strings, provide an accept/reject verdict with a 1–2 sentence justification consistent with your Task 1 description: [edge-case strings].

This prompt was constructed specifically for this work to systematically probe semantic interpretation ability, positive and negative string generation, and consistency of membership judgment with explanation. The strict three-section formatting requirement makes cross-task inconsistencies directly observable.

D.4 Tree-of-Thought Configuration

Section 3 evaluates DFA construction under Tree-of-Thought prompting (Yao et al., 2023). Our implementation uses a multi-branch strategy with two independent reasoning branches. No multi-step depth expansion, beam search, or branch scoring is performed.

Parameters.

- **Branches**: 2 — intuitive construction (D.3.1) and derivative-based construction (D.3.2)
- **Depth**: 1 (single-shot generation per branch)
- **Retries**: up to 3 per branch on JSON parse failure
- **Temperature**: 0.0 (deterministic)
- **Validation**: strict JSON parsing; invalid or malformed output marks the branch as failed
- **Selection**: each branch output is saved independently; no branch scoring is performed

Execution Pipeline. For each regex in the benchmark, the ToT driver:

1. Loads both prompt templates.
2. Executes one model call per branch.
3. On invalid JSON, appends *"OUTPUT ONLY the DFA JSON"* and retries up to 3 times.
4. Saves the raw output, metadata (finish reason, token usage), parsed DFA JSON if valid, and a structured error object if invalid.

Both branches produce one result entry regardless of success or failure, enabling failure mode analysis across both construction strategies.

Example Execution Trace. For a representative benchmark regex:

- **Intuitive branch**: Attempt 1: non-JSON text → retry. Attempt 2: invalid JSON fragment → retry. Attempt 3: well-formed DFA JSON → saved.
- **Derivative branch**: Attempt 1: malformed JSON → retry. Attempt 2: malformed again → retry. Attempt 3: still invalid → *GENERATION_FAILED* (error object saved).

API Invocation. The ToT driver implements retry logic, automatic JSON extraction, output sanitization, and rate limiting, ensuring consistent evaluation across all frontier models (GPT-5.2, Grok-4.1, Gemini-2.5). Full driver code is provided in Appendix G.

E Evaluation Protocols

This appendix describes the evaluation methodology for all experiments in this paper. Section 2 and 3 evaluations involve both human annotation and automatic verification. Section 4 evaluations are fully automatic with programmatic correctness checking.

E.1 Human Evaluation (Sections 2–3)

E.1.1 Cross-Consistency Protocol (Section 2)

The Cross-Consistency Protocol evaluation requires human assessment of three tasks per model output. All annotators are model-blinded and operate under fixed rubrics. For each model output, evaluators independently check:

- **Task 1 (Semantic description):** correctness of start and end conditions, mandatory structural blocks, forbidden patterns, repetition structure, scope and nesting, and nullability of the described language.
- **Task 2 (Accepted/rejected strings):** correctness of all 20 strings against the actual regex semantics, and coverage of structurally distinct branches of the language. A string list is marked incorrect if any single string is misclassified or if the accepted list fails to span all structural branches.
- **Task 3 (Edge-case classification):** correct ACCEPT/REJECT label with a coherent 1–2 sentence justification that is consistent with the Task 1 description.

A response is marked correct only if every component of every task is correct. Cross-task inconsistencies — where a model contradicts its own Task 1 description in Tasks 2 or 3 — are recorded separately as instances of operational–descriptive dissonance and used to populate the failure taxonomy in Section 2.1. Disagreements between annotators are resolved through adjudication by a senior reviewer.

Error rate computation. For Figure 1, the incorrect percentage for each task and model is computed as the fraction of regex instances for which the model's output for that task contains at least one error, as defined above. Errors are not mutually exclusive across tasks: a model may fail Task 1, Task 2, and Task 3 on the same instance, and each failure is counted independently.

All human annotation was performed by members of the research team, not paid crowdworkers. No external recruitment platform was used. Annotators were fully blinded to model identity and evaluated outputs against formal correctness criteria under fixed rubrics described above. Disagreements were resolved through adjudication by a senior reviewer.

E.1.2 DFA Construction Evaluation (Section 3)

For the DFA construction experiments in Section 3, generated DFAs are evaluated for correctness against the ground-truth DFA from the 180-problem benchmark. Evaluation proceeds in three stages:

Stage 1: Structural validity. The generated DFA must be:

- deterministic: at most one transition per state per symbol,
- complete: exactly one transition per state per symbol (missing transitions are filled by an explicit dead state),
- well-formed: states, alphabet, start state, and accept set are all present and consistently defined.

Any structural violation immediately marks the output as incorrect.

Stage 2: Language equivalence. Valid DFAs are tested for language equivalence against the ground-truth DFA using the validation pipeline from Shelat et al. (2026), which tests membership consistency by running both the model-generated DFA and the ground-truth DFA on all strings up to length 6 over the relevant alphabet, plus a large-scale random sample of longer strings. Two DFAs are considered equivalent if they agree on every string in this test set.

Failure mode annotation. For the failure mode analysis in Section 3.1 and Table 1, each incorrect DFA is additionally annotated by human reviewers with the failure modes (P1–P6) from the taxonomy. Failure mode labels are not mutually exclusive — a single DFA may exhibit multiple failure modes simultaneously, as reflected in Table 1. Reviewers are provided with the ground-truth DFA and the regex for reference. Disagreements between annotators are resolved by discussion with a senior reviewer.

E.2 Automatic Evaluation (Section 4)

All Section 4 experiments use fully automatic evaluation with no human annotation. Correctness is determined by programmatic exact match verification.

E.2.1 DFA Exact Match Verification

A model output is considered correct if and only if all four of the following conditions are jointly satisfied:

1. **Parse success.** The output can be parsed into a valid DFA representation with states, alphabet, start state, accept states, and transition function.
2. **Completeness.** The DFA has exactly one transition per state per symbol over the full alphabet.
3. **Determinism.** No state has more than one transition on any symbol.
4. **Language equivalence.** The DFA is language-equivalent to the ground-truth minimised DFA, verified by testing all strings up to length 6 over the relevant alphabet. Two DFAs are equivalent iff they agree on every string in this test set.

No partial credit is awarded. A DFA that is correct on 99% of test strings but fails on one is marked incorrect. This strict criterion is deliberately chosen because any single error in a Tier 4 DFA cascades through all downstream states derived from it, making partial correctness meaningless for the subset construction procedure.

String test set for equivalence. For a DFA over alphabet Σ, the test set consists of all strings $w \in \Sigma^*$ with $|w| \leq 6$. For $|\Sigma| = 2$ this yields $\sum_{i=0}^{6} 2^i = 127$ strings; for $|\Sigma| = 3$ it yields $\sum_{i=0}^{6} 3^i = 1{,}093$ strings. This provides sufficient coverage to distinguish all minimised DFAs of up to 18 states (the maximum in our benchmark) with high confidence.

Parse failure handling. If the model output cannot be parsed into a valid DFA — for example, because the model produces free-form text instead of a structured transition table — the output is marked

incorrect with a parse failure flag. Parse failure rates are tracked separately and reported as part of the zero-shot and few-shot baseline analysis in Section 4.1.

All fine-tuning experiments use a fixed random seed of 42 with full determinism enforced across Python, NumPy, PyTorch, and CUDA. Results are reported from single deterministic runs (temperature 0.0 for all evaluations); given the exact-match nature of the DFA verification task, variance across runs is expected to be negligible.

E.2.2 Tier-Stratified Evaluation

Results are reported separately for each of the four difficulty tiers (T1–T4) as well as overall. For each condition and tier, accuracy is computed as:

$$\text{Accuracy}(T_k) = \frac{\text{number of correct outputs on tier } k \text{ examples}}{\text{total tier } k \text{ examples in the test set}}.$$

The overall accuracy is the unweighted average across all test examples regardless of tier. All evaluation is performed on the held-out test set (5% of the full dataset, 1,000 examples) that is never seen during training. The validation set (5%, 400 examples: 200 IID, 200 OOD) is used only for checkpoint selection during training and is not reported as a test result.

E.2.3 Mechanistic Analysis Evaluation

ADS neuron scoring. For each of the 56,000 neuron–layer pairs, the ADS score is computed over the set of correct and incorrect examples from the 285-example Tier 4 mechanistic subset. The mechanistic subset is drawn from the test set and reserved exclusively for mechanistic analysis; it does not overlap with the examples used for accuracy evaluation in Section 4.1. The baseline accuracy on this 285-example subset (85.3%) differs slightly from the full test set Tier 4 accuracy (82.9%) due to the different difficulty distribution of this subset.

ADFA variant specificity. For each neuron, the V3 bleed ratio is computed as the ratio of the neuron's mean activation on V3 decoy examples to its mean activation on V1 and V2 examples. A neuron is considered construction-specific if its V3 bleed ratio is below 0.5, meaning it fires at less than half its peak on inputs that require no subset construction. As reported in Section 4.3, no neuron in our 56,000-candidate pool meets this threshold, with the lowest observed bleed ratio of 0.73.

Representation probing. At each of the 28 transformer layers, a logistic regression classifier is trained on the 3,584-dimensional mean hidden state vector to predict whether the model will produce a correct final answer. Training uses five-fold cross-validation on the 285 mechanistic examples (243 correct, 42 incorrect on the mechanistic subset). The reported predictive accuracy of 87.1% at Layer 10 is the mean cross-validated accuracy across the five folds.

Steering intervention evaluation. All steering interventions in Section 4.4 are evaluated on the same 285-example mechanistic subset with model weights frozen. For each intervention condition, accuracy is computed as the fraction of the 285 examples for which the steered model produces a correct DFA output, using the same exact match criterion as Section E.2.1. The baseline condition (no intervention) is run first and its 243 correct examples are excluded from subsequent rounds in the VGNS framework, which applies interventions only to the examples that failed the baseline (285 - 243 = 42 examples).

E.3 Semantic Interpretation Checks

These checks are applied during human evaluation of Section 2 outputs to ensure consistent annotation of the eleven failure modes.

Nullability consistency. Annotators verify that the model's Task 1 description correctly identifies whether $\varepsilon \in L(R)$, and that the accepted string list in Task 2 either includes or excludes the empty string accordingly. A mismatch between the stated nullability and the generated examples is recorded as a Nullability Neglect error (failure mode (ii)).

Scope and nesting. Annotators inspect whether the Task 1 explanation correctly reflects parenthesis grouping, alternation boundaries, Kleene-star scope, and concatenation ordering. Errors where the model assigns a repetition operator to the wrong subexpression are recorded as Scope and Nesting Confusion (failure mode (iv)).

Over- and under-acceptance. A model is marked incorrect on Task 2 if:

- any string in the accepted list is not in $L(R)$,
- any string in the rejected list is in $L(R)$,
- the accepted list fails to include examples from all structurally distinct branches of the language (e.g., both branches of a union are represented).

Trace consistency. For Task 3, annotators verify that the justification is logically consistent with the Task 1 description — not merely that the verdict is correct. A model that gives the right ACCEPT/REJECT label but with a justification that contradicts its Task 1 description is marked as exhibiting Descriptive–Operational Dissonance (failure mode (xi)).

E.4 Reproducibility

All evaluation scripts, DFA equivalence checkers, parsing tools, prompting templates, and training scripts are provided in Appendix G. The exact match DFA verifier, the ADS scoring pipeline, the ADFA variant generator, the representation probing script, and the VGNS steering evaluation script are all fully documented with usage instructions.

F Appendix F: Additional Results

This appendix presents supplementary qualitative and quantitative material that complements the analyses in Sections 2–4. While the main paper reports aggregated statistics and representative examples, the sections below provide:

- full raw failure examples for each semantic error category,
- targeted case studies illustrating characteristic breakdown patterns,
- zero-shot and few-shot baseline results omitted from the main paper for brevity (Section F).

These materials are not required to understand the core argument of the paper but serve as high-resolution evidence for readers wishing to inspect the model behaviors in detail.

F.1 Additional Failure Mode Examples

Section 2 introduced the taxonomy of semantic and procedural failures observed in LLMs during regex interpretation and DFA construction. Here we provide compact raw outputs for each category in order to show the mechanisms of failure directly.

Each example is structured as:

1. the original regex,
2. the models raw output (Tasks 1–3),
3. a brief explanation of why the output instantiates the failure mode.

We begin with the phenomenon: *Anchor Hallucination*, where the model recognizes most of the correct structure yet misidentifies the fixed suffix or prefix constraints of the regex.

F.1.1 Anchor Hallucination

Regex.

$$R = (b \mid a(a(bb \mid bbb)^*ba)^*b)^* \\ a(a(bb \mid bbb)^*ba)^* \\ a(bb \mid bbb)^*bba$$

Model Output (Cross-Consistency Tasks 1–3).

=== TASK 1 ===

> The language consists of strings over {a,b} that are built from an initial repetition of either a single b or a longer block that starts with a, ends with b, and may contain repeated internal segments. After that initial part, every string must contain an a, then another a, and then finish with a fixed tail that ends in `bba`. The ending is always of the form (bb or bbb repeated zero or more times) followed by `bba`, **so every string ends with at least three b's in a row**. The middle portion between the two required a's is a block of the form `a(bb or bbb)*ba` repeated zero or more times, meaning it is made of zero or more chunks each starting with a, then a number of bb/bbb groups, then ba. The same repeated-chunk structure can also appear inside the initial part, but there it is wrapped so that the whole alternative begins with a and ends with b. Because of the mandatory "aa" in the final concatenation, no accepted string can be empty or end in a. Every accepted string contains at least two a's (those two required a's), and it contains no symbols outside {a,b}. The repetition structure is nested: an outer Kleene star over (b or a(...)*b), and inside the a(...)*b alternative another Kleene star over the chunk `a(bb|bbb)*ba`. The only way to introduce a's is through these chunks and the final required `aa`. The final tail forces the last a before `bba` to be followed by a sequence of bb/bbb groups and then exactly `bba`, so endings like "...bab" or "...b" are forbidden.

=== TASK 2 ===

> **ACCEPTED:** `aabba`, `aaabbba`, `baaabbba`, `bbaabbbba`, `aaababbba`, `aabbaabbbba`, `baababbba`, `abbbaabbbba`, `babaaabbba`, `ababbbaababbba`
> **REJECTED:** `a`, `b`, `aa`, `aabb`, `abba`, `baaba`, `abab`, `bbb`, `aababa`, `babbabaab`

=== TASK 3 ===

> `aaababbb` → **REJECTED:** The regex requires the final suffix $(bb \mid bbb)^*$`bba`, so the string must end in `bba`. This string ends in `bbb`.
>
> `babbabaab` → **REJECTED:** The string ends in `aab`, which cannot match the required `bba`.
>
> `ababbbaab` → **REJECTED:** The last three symbols are `aab`, not `bba`.

Why This Is an Anchor Hallucination Error. The regex *always* forces strings to end in the literal substring `bba`; thus the final character of every accepted string is `a`. However, the model asserts

that "every string ends with at least three b's in a row," hallucinating a bbb ending. This contradicts both the true regex structure and its own accepted examples (e.g., aabba). The model correctly identifies the internal components (bb, bbb, bba) but misorders them, conflating inner repetition with the anchored suffix. This mismatch between the true anchored ending and the hallucinated ending exemplifies the **Anchor Hallucination** failure mode.

F.1.2 Nullability Neglect

Regex.

$$(b^*aaa^*ba \mid b^*aba \mid b^*ab)(a \mid b)^*$$

Model Output (Tasks 1–3).

=== TASK 1 ===

> Strings in this language always begin with zero or more b's followed by an a, as every alternative in the prefix begins with b* a. After these, the string must follow one of three core patterns: either a second a followed by zero or more a's, then b a; or b a; or just b. The first pattern requires at least two consecutive a's after the initial prefix, followed by optional additional a's, then b a. The second pattern demands exactly b a. The third pattern requires exactly b. The string may then end with any sequence of a's and b's (including empty) because of the trailing (a|b)*. Forbidden strings are those missing a complete prefix or with insufficient symbols to complete a core pattern.

=== TASK 2 ===

> **ACCEPTED:**: ab, bab, aba, baba, aaba, baaba, aaaba, baaaba, aabab, abaab
> **REJECTED:**: a, aa, aaa, aaaa, b, bb, bbb, ba, aab, bbba

=== TASK 3 ===

> aaababbb → ACCEPTED: Matches the first branch with empty b*, core a a a* b a, and suffix bbb.
> babbabaab → ACCEPTED: Matches the third branch with b* = b, core a b, suffix babaab.
> ababbbaab → ACCEPTED: Matches the third branch with empty b*, core a b, suffix abbbaab.

Why This Exemplifies Nullability Neglect. The model systematically misstates the optionality of a* in the first branch. It claims:

> "The first pattern requires at least two consecutive a's after the initial b's and first a, followed by optional additional a's."

But in the branch $b^*\,a\,a\,a^*\,b\,a$, the factor a* is *nullable*; it may contribute zero a's. Thus the branch requires only two consecutive a's, not three. The model visually conflates the sequence a a a* into a mandatory triple-a block, treating "*" as if it were "+" or a fixed repetition. The inconsistency is exposed immediately by its own Task 2 outputs, where it accepts aaba, which contains exactly two a's in the core.

F.1.3 Atomic Unit Blindness

Regex.

$$(b \mid a(a(bb \mid bbb)^*ba)b)^*$$
$$a(a(bb \mid bbb)^*ba)^*$$
$$a(bb \mid bbb)^*bba$$

Model Output (Cross-Consistency Tasks 1–3).

=== TASK 1 ===

> The language consists of strings over {a,b} that are built from an initial repetition of either a single b or a longer block that starts with a, ends with b, and may contain repeated internal segments. After that initial repeated part, every string must contain an a, then another a, and then finish with a fixed tail that ends in bba. The ending is always of the form (bb or bbb repeated zero or more times) followed by bba, so every string ends with at least three b's in a row. The middle portion between the two required a's is a block of the form a(bb or bbb)*ba repeated zero or more times. The same chunk structure can also appear inside the initial repeated part. Because of the mandatory aa in the final concatenation, no string in the language can be empty or end in a. The repetition structure is nested and any a must appear as part of these grouped patterns rather than arbitrarily. The final tail forces the last a before the ending b's to be followed by a sequence of bb/bbb groups and then exactly bba, so endings like ...bab or ...b are forbidden.

=== TASK 2 ===

> **ACCEPTED:** aabba, aaabbba, baaabbba, bbaabbbba, aaababbba, aabbaabbbba, baababbba, abbbaabbbba, babaaabbba, ababbbaababbba
> **REJECTED:** a, b, aa, aabb, abba, baaba, abab, bbb, aababa, babbabaab

=== TASK 3 ===

> aaababbb → REJECTED: Ends with bbb, but the regex requires the literal suffix bba.
> babbabaab → REJECTED: Ends with aab, not the required bba.
> ababbbaab → REJECTED: Final three symbols are not bba, so it cannot match.

Why This Exemplifies Atomic Unit Blindness. The model accepts the string aaabbba, which ends in:

$$\underbrace{a}_{\text{prefix anchor}} \quad \underbrace{b}_{\text{from } (bb|bbb)^*} \quad \underbrace{bba}_{\text{literal suffix}}$$

This requires the Kleene-star block $(bb \mid bbb)^*$ to generate *exactly one* b. But the building blocks have fixed atomic sizes:

$$|bb| = 2, \qquad |bbb| = 3,$$

so the required equation

$$2x + 3y = 1$$

has no non-negative integer solution. Thus producing a single b is *mathematically impossible*. The model treats $(bb \mid bbb)^*$ as a loose "texture of b's" rather than a combination of indivisible atomic units of length 2 or 3. This arithmetic violation is the signature of *Atomic Unit Blindness*.

F.1.4 Scope and Nesting Confusion

Regex.

$$R = (a \mid b \mid ab)^* \Big((bb^*a \mid aaa \mid \epsilon)(ba)^*\mathbf{b} \mid aa\Big)$$

Model Output (Cross-Consistency Tasks 1–3).

=== TASK 1 ===

> The language consists of strings over {a, b} that start with any number of blocks chosen from a, b, or ab. After that prefix, the string must finish with either the suffix aa or with a more structured suffix that ends in a single final b. In the "final b" case, the part immediately before the last b is either empty, or of the form bb*a, or exactly aaa, possibly followed by repetitions of ba. Thus strings can end in aa, or end in b with the tail matching $(bb^*a \mid aaa \mid \epsilon)(ba)^*b$.

=== TASK 2 ===

> **ACCEPTED:** aa, b, ab, aab, aaab, bbab, aaabb, ababab, ababaabab, bbababab
> **REJECTED:** a, ba, aba, bbb, bab, aaba, abba, bba, aaaba, ababb

=== TASK 3 ===

> aaababbb → **REJECTED:** The string ends in b but the remaining suffix cannot match either branch of the final disjunction.
>
> babbabaab → **ACCEPTED:** It can be split as $(a|b|ab)^* =$ babbaba and then the ending aa followed by b is achieved by taking the final alternative.
>
> ababbbaab → **ACCEPTED:** A valid decomposition exists with the last b completing the structure.

Why This Is a Scope and Nesting Error. The regex defines two *mutually exclusive* endings:

1. a structured branch that must end in a final **b**: $(bb^*a \mid aaa \mid \epsilon)(ba)^*\mathbf{b}$,
2. the literal string **aa**.

The probe string babbabaab ends in aab. To accept it, the model asserts that "the ending aa followed by b is achieved," effectively treating the b from the *first* alternative as if it could be appended to the standalone aa branch. This is impossible in the true grammar: the final b is *scoped inside* the left disjunct and cannot attach to the right disjunct.

In effect, the model collapses the grouping

$$((bb^*a \mid aaa \mid \epsilon)(ba)^*\mathbf{b}) \;\mid\; aa$$

into the incorrect structure

$$((bb^*a \mid aaa \mid \epsilon)(ba)^*) \;\mid\; (aa)\mathbf{b},$$

merging two disjoint parse branches into one. This is a direct instance of **Scope and Nesting Confusion**: the model loses the boundary of the alternation and treats an internal suffix as if it were globally available outside its parenthesized scope.

F.1.5 Pseudo-Structural Hallucination

Regex.

$$(b^*a \;\mid\; b^*aa(aa \mid bb)^*a)\, ba^*$$

Model Output (Cross-Consistency Tasks 1–3).

=== TASK 1 ===

> Strings in this language always begin with zero or more b's. After the initial b's, they follow one of two paths: either a single a, or aa followed by zero or more repetitions of aa or bb, and finally another a. Every string contains a mandatory b immediately after the prefix. All strings end with this mandatory b followed by zero or more a's. No string may contain b's after this final mandatory b, except within the optional bb blocks of the second alternative. Forbidden patterns include b's after the suffix, unpaired a's in the prefix, or skips of required grouped patterns.

=== TASK 2 ===

> **ACCEPTED:** ab, bab, aba, bbabaa, aaab, baaab, aaaaab, aabbab, aaabbabaa, bbaa bb aa a bab
> **REJECTED:** a, b, aa, aab, abab, abbb, aaa, bb, baba, aaaba

=== TASK 3 ===

> aaababbb → REJECTED: Ending contains b's beyond the mandatory suffix ba*.
> babbabaab → REJECTED: Mixed b's appear after the mandatory suffix position.
> ababbbaab → REJECTED: After the prefix and mandatory b, trailing segment contains b's, violating the a* suffix.

Why This Exemplifies Pseudo-Structural Hallucination. The model accepts the string

`bbaa bb aa a bab`

(with internal spaces preserved), which visually resembles the regex because it contains plausible building blocks:

$$\{\,\mathtt{b}, \mathtt{bb}, \mathtt{aa}, \mathtt{a}\,\}$$

and mirrors the local texture induced by $(aa \mid bb)^*$. However, the string is *impossible* for the regex.

- **Suffix violation:** The regex requires the string to end in `ba*`. Thus, after the final mandatory `b`, *only* `a`'s may follow. Yet the string ends with `bab`, which contains a forbidden trailing `b`.
- **Global-structure violation:** The inner segment $(aa \mid bb)^*$ must have even length and appear only between the two boundary `a`'s of the second branch. The model's stitched sequence breaks these boundaries, incorrectly interleaving chunks across structural borders.

This is the hallmark of *Pseudo-Structural Hallucination*: the model approximates the *local texture* of the regex (assembling convincing fragments like `aa` and `bb`) but fails to enforce the *global combinatorial constraints*. It synthesizes strings that "look regex-like" without validating whether they are actually generable by any valid parse of the expression.

F.1.6 Simple-Path Bias

Regex.

$$R = \underbrace{a^*(b(ab)^*ba^*)^*}_{\text{Branch 1 (Dominant)}} \mid \underbrace{\big(a^*(b(ab)^*ba^*)^*\big)(b(ab)^*aa \mid \epsilon)b^*}_{\text{Branch 2 (Complex / Ignored)}}$$

Model Output (Cross-Consistency Tasks 1–3).

=== TASK 1 ===

> The overall regex is a union of two big alternatives. After the shared prefix in the second alternative, one may insert a "$b(ab)^*aa$" segment or nothing, followed by trailing b^*. Strings may therefore end either with a's or with "aa" immediately after a $b(ab)^*$ segment.

=== TASK 2 ===

> **ACCEPTED:** ϵ, a, aaaa, bb, bba, babba, abb, abbb, bbaa, abbabb
> **REJECTED:** `baa`, `ba`, `aba`, `aab`, `abba`, ...

Example of Simple-Path Bias. The regex provides two structurally distinct paths:

- **Branch 1:** repetition of blocks of the form $b(\cdots)b\,a^*$,
- **Branch 2:** a unique suffix of the form $b(ab)^*aa$ followed by b^*.

Although the model describes Branch 2 in Task 1, its generated examples (Task 2) and classifications exhibit a clear **Simple-Path Bias**:

1. **Generation Bias:** All "ACCEPTED" examples are realizable via Branch 1 (or the ϵ-case of Branch 2, which collapses to Branch 1). No example uniquely reflects the distinctive $b(ab)^*aa$ structure.
2. **Active Rejection of the Unique Path:** The string `baa` is the simplest witness for Branch 2 ($b \cdot aa$), yet the model lists it under **REJECTED**.

Thus the model effectively collapses the non-deterministic structure into a single, simpler path, ignoring the deeper branch and preferring the most familiar or repetitive option.

F.1.7 Complexity Aversion

Regex.

$$\begin{aligned} R = {} & a^*b \\ & \mid \big((a^*a \mid a^*b)b^*a\big) \\ & \mid \Big(\big((a^*a \mid a^*b)b^*a\big)b^*\Big)\mathbf{b}a^* \end{aligned}$$

Model Output (Cross-Consistency Tasks 1–3).

=== TASK 1 ===

> The language is the union of three main forms, all over the alphabet {a,b}. One form consists of zero or more a's followed by a single b. A second form begins with either one or more a's or some a's followed by b, then has at least one b before a final a. A third form begins similarly but requires an additional block ending in a mandatory `b` followed by optional trailing a's. In the second and third forms, the initial $(a^*a \mid a^*b)$ forces the string to begin with a's (possibly followed by a single b). In the first form, the string ends with a single b after only a's. In the second form, the string must end with a. In the third form, the string must contain a "...ab" structure followed by optional trailing a's, but no further b's.

=== TASK 2 ===

> **ACCEPTED:** `b`, `ab`, `aaab`, `aba`, `aaba`, `abbba`, `aabbbba`, `abab`, `aababb`, `aaabbbbaaa`
> **REJECTED:** `a`, `ba`, `bb`, `abaabb`, `aaaba`, `ababa`, `aabbaa`, `babb`, `abbaaaab`, `ababbbaab`

=== TASK 3 ===

aaababbb → **REJECTED:** The model claims that it ends with b and is not of the form a^*b, and that under the other branches it would need to end in a's after "...ab", which it does not.

babbabaab → **REJECTED:** All branches require the string to begin with a (except the singleton b), so this cannot match.

ababbbaab → **REJECTED:** The model states that it is neither a^*b nor a member of the other branches, as those must end in a.

Example of Complexity Aversion. The regex contains a deeply nested third branch:

$$\Big(\big((a^*a \mid a^*b)b^*a\big)b^*\Big)\mathbf{b}a^*$$

This branch allows strings that end with a block of b's (via the inner b^* and the mandatory final **b**), optionally followed by trailing a's.

The probe string aaababbb is *mathematically valid* under this branch. One correct parse is:

- Inner group: aaa (from a^*a),
- Middle: b (from b^*) followed by the mandatory a,
- Outer block: bb (from the second b^*) and then the required final b,
- Suffix: ϵ (from a^*).

Thus the string satisfies the full structure of the third alternative.

However, the model **incorrectly rejects** the string. Its explanation asserts:

> "...it ends with b and is not of the form a^*b; after the required '...ab' it would need only a's, but it ends with b's."

This reasoning omits the complex third branch entirely and substitutes a simplified prototype: it assumes that any valid "...ab" structure must be followed only by a's. The model therefore overlooks the legitimate path that allows trailing b's and defaults to rejection.

This is a clear instance of **Complexity Aversion**: when confronted with a deeply nested, multi-layered disjunction, the model avoids the cognitively expensive parse and instead falls back to an oversimplified heuristic description of the language.

F.1.8 Trace Fabrication

Regex.

$$R = (a \mid b)^*\, a\, (aa \mid bba)^*\, b\, (a \mid b)$$

Model Output (Cross-Consistency Tasks 1–3).

=== TASK 1 ===

The language consists of all strings over {a, b} that can be split into four consecutive parts: an initial block of any number of a's and b's, then a single mandatory a, then zero or more repetitions of either the block aa or the block bba, then a mandatory b, and finally one last symbol that is either a or b. Strings may begin with either letter and may even begin with the mandatory a if the initial $(a|b)^*$ part is empty. Every string must contain at least one a (the required a after the prefix) and must also contain a b near the end (the required b before the final symbol). The ending is always exactly two characters long, finishing in ba or bb. The repeated region $(aa|bba)^*$ must decompose cleanly into whole blocks and cannot contain unattested fragments.

=== TASK 2 ===

ACCEPTED: aba, abb, aaba, aabb, aaaba, ababa, abbbba, abbaba, baabbba, bbabbaaabb
REJECTED: a, b, aa, bb, ab, ba, aaa, bbbb, abab, baba

=== TASK 3 ===

aaababbb → **ACCEPTED:** Split as $(a|b)^* =$ aaab, then mandatory a, then $(aa|bba)^* =$ bba, then mandatory b, final $(a|b) =$ b.

babbabaab → **REJECTED:** The remaining substring cannot be decomposed into $(aa|bba)^*$ followed by b(a|b).

ababbbaab → **REJECTED:** The prefix consumes incorrectly because a mandatory a is required before the repeated block.

Why This Is a Trace Fabrication Error. For the probe string aaababbb, the model generates the following execution trace:

$$(aa \mid bba)^* = \texttt{bba}$$

However, the actual substring at the corresponding position (indices 5–7) is:

bbb

The model explicitly asserts the existence of a substring (bba) that is physically absent from the input. This is not merely a misinterpretation of state; it is a *fabricated match* where the model alters the input data in its internal reasoning to satisfy the regex constraints. The trace is a post-hoc rationalization that contradicts the causal reality of the string's content.

F.1.9 Greedy Parsing Failures

Regex.

$$R = a^*(b(ab)^*ba^*)^* \mid \big(a^*(b(ab)^*ba^*)^*\big)(b(ab)^*aa \mid \epsilon)b^*$$

Model Output (Cross-Consistency Tasks 1–3).

=== TASK 1 ===

> The language consists of strings over {a,b} built from an initial run of zero or more a's, followed by zero or more repetitions of a fixed block $b(ab)^*ba^*$, and possibly ending with a special suffix. The core repeated block begins with b, contains zero or more copies of ab, then another b, and ends with any number of trailing a's. The first alternative generates exactly $a^*(b(ab)^*ba^*)^*$, while the second alternative extends this prefix with either a special segment $b(ab)^*aa$ or ϵ, followed by trailing b^*. Thus strings may end with a's (via a^*), with b's (via b^*), or with "aa" immediately after a $b(ab)^*$ pattern.

=== TASK 2 ===

> **ACCEPTED:** ϵ, a, aaaa, bb, bba, babba, abb, abbb, bbaa, abbabb
> **REJECTED:** b, ba, aba, aab, baa, abba, abab, bab, aaababbb, ababbbaab

=== TASK 3 ===

> aaababbb → **REJECTED:** The model asserts that "babbb" cannot form valid blocks and that the trailing b^* cannot repair the mismatch.
>
> babbabaab → **REJECTED:** The prefix "babbaba" does not form block-structured repetitions and cannot be reinterpreted with the special ending.
>
> ababbbaab → **REJECTED:** After the initial "a", the remainder cannot be decomposed as valid blocks or as the special ending followed by trailing b's.

Example of Greedy Parsing Failure. The string aaababbb is in fact *accepted* by the second branch of the regex. A correct parse is:

- **Prefix (a^*):** aaa,
- **One block** ($b(ab)^*ba^*$)**:** babb

$$b \cdot ab \cdot b \cdot \epsilon$$

- **Suffix (b^*):** b.

Thus the entire string matches:

$$a^*(b(ab)^*ba^*)^*\,(\epsilon)\,b^*.$$

However, the model rejects the string. Its reasoning reveals a **greedy, non-backtracking commitment** to the repeated-block structure:

1. After successfully parsing the block babb, the model encounters a final b.
2. Because a block must begin with b, the model greedily attempts to start a *new* iteration of $(b(ab)^*ba^*)$.
3. The final b alone does not complete such a block (it lacks the matching closing b), causing a local failure.
4. Crucially, the model *does not backtrack* to reinterpret the final b as part of the valid b^* suffix, even though that would yield a correct full match.

This behavior exemplifies a **Greedy Parsing Failure**: the model commits prematurely to a repetitive structure, fails locally, and–lacking explicit backtracking–never explores the alternative parse permitted by the grammar. Instead of simulating the full disjunction with proper backtracking, it behaves like a greedy regular-expression engine that locks itself into the most immediate continuation and rejects valid strings.

F.1.10 Indexing and Positional Drift

Regex.

$$R = (a \mid b)^*\, a\, (aa \mid bba)^*\, b\, (a \mid b)$$

Model Output (Cross-Consistency Tasks 1–3).

=== TASK 1 ===

> The language consists of all strings over {a, b} that can be split into four consecutive parts: an initial block of any number of a's and b's, then a single mandatory a, then zero or more repetitions of either the block aa or the block bba, then a mandatory b, and finally one last symbol that is either a or b. Strings may begin with either letter and may even begin with the mandatory a if the initial $(a|b)^*$ part is empty. Every string must contain at least one a (the required a after the prefix) and must also contain a b near the end (the required b before the final symbol). The ending is always exactly two characters long in the sense that the string must finish with b followed by either a or b, so it always ends in ba or bb. The middle repeated region, if present, is constrained to be a concatenation of whole blocks, each block being exactly aa or exactly bba, and it cannot contain a lone single a or a lone single b inside that region. It is forbidden for the string to end with a lone a or a lone b, because the final $(a|b)$ requires exactly one more character after the mandatory b. Repetition occurs only in two places: the arbitrary prefix $(a|b)^*$ and the block repetition $(aa|bba)^*$, and the latter repeats in whole-block units.

=== TASK 2 ===

> **ACCEPTED:** aba, abb, aaba, aabb, aaaba, ababa, abbbba, abbaba, baabbba, bbabbaaabb
> **REJECTED:** a, b, aa, bb, ab, ba, aaa, bbbb, abab, baba

=== TASK 3 ===

> aaababbb → **ACCEPTED:** Split as $(a|b)^* =$ aaab, mandatory a, then $(aa|bba)^* =$ bba, then mandatory b, final $(a|b) =$ b.

babbabaab → **REJECTED:** After taking prefix babb, mandatory a, the remaining baab cannot be decomposed into $(aa|bba)^*$ followed by b and one final symbol.

ababbbaab → **REJECTED:** After prefix aba, mandatory b is not allowed there because the regex requires a mandatory a before the $(aa|bba)^*$ region and final b(a|b).

Why This Is an Indexing and Positional Drift Error. The model asserts that the probe string aaababbb is **ACCEPTED** and provides a step-by-step decomposition. Summing the lengths of the claimed segments reveals the error:

$$\underbrace{\texttt{aaab}}_{4} \mid \underbrace{\texttt{a}}_{1} \mid \underbrace{\texttt{bba}}_{3} \mid \underbrace{\texttt{b}}_{1} \mid \underbrace{\texttt{b}}_{1} \quad = \quad \textbf{Length 10}$$

However, the input string aaababbb has only **Length 8**.

The model maintains correct indexing up to position 4. At indices 5–7, the actual substring is bbb, but the model hallucinates the sequence bba to satisfy the inner group $(aa|bba)^*$. Having drifted off the end of the string, it then hallucinates two additional characters (indices 8 and 9) to satisfy the mandatory suffix structure. This proves the model is generating a plausible text narrative rather than executing a constrained search over the fixed input array.

F.1.11 Descriptive–Operational Dissonance

Regex.

$$\big((10^*1 \mid 0)\, 1^*(0(00^*1)^*11^*)^*0\big)^*$$
$$(10^*1 \mid 0)\, 1^*(0(00^*1)^*11^*)^*0$$
$$(00^*1)^*\, 00^*$$

Model Output (Cross-Consistency Tasks 1–3).

=== TASK 1 ===

Strings in the language begin with either a single 0 or 1 followed by zero or more 0's and then a 1. After this initial segment, zero or more 1's may appear, followed by zero or more instances of a block that starts with 0, contains repetitions of 00*1, and ends with one or more 1's, before concluding with a 0. These blocks repeat zero or more times, with one mandatory block always present. After the mandatory block, zero or more repetitions of 00*1 appear. The string always ends with one or more 0's from the final 00*. The model claims: "Strings must have *at least three* 0's minimally, as in "000"."

=== TASK 2 ===

ACCEPTED: 000, 1100, 10100, 0100, 00100, 0001100, 0010100, 110000, 00010, 00010100
REJECTED: "", 0, 00, 1, 10, 11, 100, 001, 010, 101

=== TASK 3 ===

00010111 → REJECTED: Ends with 111, violating the final 00*.
101101001 → REJECTED: Final symbol is 1; suffix must be 00*.
010111001 → REJECTED: Violates the mandatory trailing zeros.

Why This Exemplifies Descriptive–Operational Dissonance. In Task 1, the model asserts a strict global constraint:

> "Strings must have at least three 0's minimally, as in 000."

Yet in Task 2 it accepts the string

$$\texttt{1100},$$

which contains only *two* 0's. The model therefore violates the very rule it claimed was mandatory.

This mismatch reveals a fundamental disconnect between the model's *descriptive* understanding (natural-language explanation of the regex) and its *operational* behavior (actual generation of examples). The description module hallucinates a quantitative "minimum three zeros" constraint–likely influenced by repeated zero-literals in the pattern–while the generation module ignores this rule completely. Because the model correctly rejects the suffix-violating probes in Task 3, the dissonance cannot be attributed to random error: the model understands *local* constraints, but its globally stated rule and its operational behavior are logically inconsistent.

This is the hallmark of *Descriptive–Operational Dissonance*: the model's linguistic explanation and its symbolic behavior do not arise from a unified internal representation.

F.2 Zero-shot and Few-shot Baseline Results

Zero-shot performance is uniformly 0% across all model sizes and tiers — models produce natural-language descriptions but never a correctly formatted transition table. Few-shot prompting (3 demonstrations) improves parse rates but yields limited gains: the 7B model achieves 20.0% overall but only 2.4% on Tier 4, the 1.5B model fails entirely, and the 14B model achieves just 4.0%. The non-monotonic pattern reflects a known tension between capacity to follow format demonstrations and strength of pretrained generative priors.

Training	Model	Overall	T1	T2	T3	T4
Zero-shot	1.5B	0.0%	0.0%	0.0%	0.0%	0.0%
	7B	0.0%	0.0%	0.0%	0.0%	0.0%
	14B	0.0%	0.0%	0.0%	0.0%	0.0%
Few-shot (3)	1.5B	0.0%	0.0%	0.0%	0.0%	0.0%
	7B	20.0%	52.6%	19.1%	13.7%	2.4%
	14B	4.0%	2.6%	6.4%	5.5%	0.0%

Table 8: Zero-shot and few-shot baseline accuracy across tiers.

G Reproducibility Checklist

This appendix documents all resources, code, datasets, hyperparameters, compute details, and evaluation procedures used in this paper.

Artifact Repository. All code, datasets, prompting templates, training configurations, and evaluation scripts used in this paper are available in an anonymized archival repository:

https://anonymous.4open.science/r/llm-formal-reasoning-C0FA/

G.1 Code Availability

All training code, inference scripts, Tree-of-Thought drivers, correctness checkers, mechanistic analysis scripts, steering intervention scripts, and data preprocessing utilities used in this work are provided in this appendix as plain-text listings and in the artifact repository.

Available components.

- Regex-to-DFA dataset generator with tier-controlled sampling, Thompson construction, subset construction, Hopcroft minimisation, and programmatic verification (*generate_dataset.py*).
- LoRA fine-tuning scripts for Qwen 2.5-1.5B, 7B, and 14B under both CoT and No-CoT supervision formats (*train_qwen{size}_lora{_cot}.py*).
- Generic LoRA training script with early stopping, replay buffer, and flash attention support for curriculum experiments (*train_lora.py*).
- Phase-wise curriculum SLURM submission script with self-chaining job queue (*submit_curriculum_phasewise.sh*, *launch_all_curriculum.sh*).
- ADS neuron scoring pipeline with ADFA variant contrast examples (*compute_ace_scores.py*, *build_adfa_variants.py*).
- Representation-level steering vector extraction with ActAdd, probe direction, SADI, and cross-model transfer (*extract_steering_vectors.py*).
- Full steering intervention evaluation including VGNS four-round iterative framework (*run_steering_eval.py*).
- Tree-of-Thought experiment driver for frontier models with JSON enforcement, retry logic, and rate limiting.
- All prompting templates (Intuitive, Derivative, Cross-Consistency).
- DFA equivalence checker and exact match verifier.

G.2 Dataset Availability

All datasets introduced in this work — the regex-to-DFA fine-tuning benchmark (22,500 training and 1,000 test examples) and all associated curriculum phase splits — are released under the Creative Commons Attribution 4.0 International (CC BY 4.0) license. The Qwen 2.5-Instruct models used for fine-tuning are publicly available under their respective open-source licenses as specified in the official Qwen 2.5 technical report (Qwen Team, 2025). The diagnostic benchmark from Shelat et al. (2026) is used consistent with its public research release.

Diagnostic Benchmark (Sections 2–3). We use the 180 regex-to-DFA construction problems from Shelat et al. (2026), available in the artifact repository. Each problem is paired with a ground-truth minimised DFA verified for language equivalence.

Regex-to-DFA Fine-Tuning Benchmark (Section 4). We release the full benchmark introduced in this work:

- 22,500 training examples and 1,000 test examples in both CoT and No-CoT formats
- The dataset generator script with fixed seed (42) for exact reproduction
- IID and OOD validation splits (200 examples each)
- Phase-wise curriculum splits (phase1 through phase5 train and validation files)

G.3 Model Architectures

All fine-tuning and mechanistic analysis experiments use the Qwen 2.5-Instruct family (Qwen Team, 2025). Full architectural details are provided in Appendix D.1. Frontier model evaluations (Sections 2–3) use GPT-5.2, Grok-4.1, and Gemini-2.5

accessed via their respective APIs at temperature 0.0.

G.4 Training Hyperparameters

All LoRA fine-tuning hyperparameters are documented in full in Appendix D.2. For completeness, the key parameters are: $r = 32$, $\alpha = 64$, dropout 0.05, AdamW with lr $= 2 \times 10^{-4}$, cosine scheduling, warmup ratio 0.03, weight decay 0.01, 2 epochs, effective batch size 8, bfloat16 precision, maximum sequence length 8,192 tokens, random seed 42.

G.5 Compute Resources

All Qwen 2.5 fine-tuning experiments, curriculum learning experiments, and mechanistic analysis experiments are run on a SLURM-managed HPC cluster using single NVIDIA H100 80GB GPU nodes with 80GB memory allocation per job. The following resource configuration is used for all jobs:

- **GPU**: $1 \times$ NVIDIA H100 80GB
- **Memory**: 80GB
- **CPUs per task**: 8
- **Maximum wall time**: 48 hours per job
- **Precision**: bfloat16 with TF32 enabled

Frontier model experiments (GPT-5.2, Grok-4.1, Gemini-2.5) are conducted via API calls with no local GPU requirement.

Each fine-tuning job runs for a maximum of 48 hours on a single NVIDIA H100 80GB GPU. Estimated total GPU hours across all fine-tuning experiments (CoT, No-CoT, and curriculum variants across three model sizes): approximately 300–400 GPU hours. Mechanistic analysis and steering evaluation scripts require no additional gradient computation and run within 12 hours on the same hardware.

G.6 Evaluation Methodology

Full evaluation protocols are documented in Appendix E. All Section 4 evaluations use programmatic exact match verification with no human annotation. Sections 2–3 evaluations combine human annotation with automatic DFA equivalence checking using the validation pipeline from Shelat et al. (2026). The DFA exact match verifier tests all strings up to length 6 over the relevant alphabet.

G.7 Mechanistic Analysis Scripts

ADS Neuron Scoring (*compute_ace_scores.py*). Registers hooks on the SiLU output (MLP intermediate activations, 18,944-dimensional for the 7B model) of all 28 transformer layers. Samples 2,000 neurons per layer (56,000 total). Computes $\text{ADS}_{\text{good}}(l, i) = \bar{a}(l, i|\text{correct}) - \bar{a}(l, i|\text{incorrect})$ over completion tokens. Pairs with ADFA variant framework for specificity scoring. Outputs *good_neurons.json*, *bad_neurons.json*, *mixed_neurons.json*, *ace_layer_summary.json*.

ADFA Variant Generator (*build_adfa_variants.py*). Generates four contrast example types per positive Tier 4 example: V1 (same DFA, symbols renamed), V2 (different regex, isomorphic DFA), V3 (complex-looking regex producing a trivial 2-state DFA decoy), V4 (same example with step-level correctness annotations for VGNS). Outputs four JSONL files, one per variant type.

Steering Vector Extraction (*extract_steering_vectors.py*). Registers hooks on layer outputs (3,584-dimensional hidden states for 7B). Extracts mean difference vector (ActAdd), logistic regression probe direction with five-fold cross-validation, and SADI sparse vector (top-10% dimensions by absolute value). Optionally extracts cross-model transfer vector. Outputs *steering_vectors.json* and *probe_accuracy.json*.

Steering Evaluation (*run_steering_eval.py*). Evaluates all intervention conditions on the 285-example Tier 4 mechanistic subset: good-neuron amplification ($\times 1.5$), bad-neuron suppression ($\times 0.5$), functional antagonism, random neuron control, ActAdd, probe direction steering, SADI adaptive steering, and VGNS four-round iterative framework. Uses the exact match DFA verifier for correctness checking. Outputs *steering_results.json* and *vgns_results.json*.

G.8 Random Seeds

All experiments use random seed 42 with full determinism enforced:

- Python: *random.seed(42)*
- NumPy: *np.random.seed(42)*
- PyTorch: *torch.manual_seed(42)*, *torch.cuda.manual_seed_all(42)*
- HuggingFace: *set_seed(42)*
- Environment: *PYTHONHASHSEED=42*, *OMP_NUM_THREADS=1*
- cuDNN: *deterministic=True*, *benchmark=False*

The dataset generator uses *random.Random(42)* for all sampling, ensuring the exact same dataset is

reproduced on re-run.

G.9 Dataset Generator

The full dataset generator script (*generate_dataset.py*) is provided below. It implements the complete pipeline: tiered regex AST generation, Thompson NFA construction, ε-closure and subset construction, Hopcroft minimisation, 8-step CoT trace generation, Qwen ChatML formatting, programmatic verification, deduplication, and train/val/test splitting.

Dataset Generator

```
"""
Regex -> DFA Dataset Generator for Qwen 2.5
     Instruct LoRA Fine-tuning
================================
Generates Chain-of-Thought (CoT) examples
     following:
  1. Thompson's NFA construction
  2. epsilon-closure computation
  3. Subset (Powerset) construction -> DFA
  4. DFA minimisation (Hopcroft's procedure)
  5. Full CoT trace formatted in Qwen ChatML
     template

Dataset tiers for curriculum learning:
  Tier 1 - concatenation, simple union
           (~15%)
  Tier 2 - Kleene star, plus, optional
           (~25%)
  Tier 3 - Combined: (a|b)*abb style
           (~35%)
  Tier 4 - Complex nested / multi-operator
           (~25%)

FIXES vs v1:
  - Minimised DFA transition table was wrong (
     remap logic fully rewritten)
  - Accept states in minimised DFA were
     sometimes wrong (wrong rep state chosen)
  - Verification now cross-checks NFA == DFA ==
     minDFA on all strings up to len 7
  - Verification examples now include meaningful
      accepted strings (not just len 0/1)
  - Minimisation CoT shows actual partition
     refinement steps (not just "apply Hopcroft"
     )
  - "Already minimal" case is reported cleanly
  - Dead-state handling fully decoupled from
     live-state processing
"""

import random
import json
import hashlib
import itertools
import argparse
from collections import defaultdict, deque
from copy import deepcopy
from typing import Optional

try:
    from tqdm import tqdm
except ImportError:
    class tqdm:
        def __init__(self, total=None, **kwargs)
     : self.n = 0; self.total = total
        def update(self, n=1):
            self.n += n
            print(f"\r  {self.n}/{self.total}",
     end="", flush=True)
        def __enter__(self): return self
        def __exit__(self, *a): print()
```

```
# ===================
#  1.  REGEX AST
# ===================

class RegexNode: pass

class Literal(RegexNode):
    def __init__(self, char): self.char = char
    def __repr__(self): return f"Lit({self.char!
     r})"

class Epsilon(RegexNode):
    def __repr__(self): return "eps"

class Concat(RegexNode):
    def __init__(self, left, right): self.left =
      left; self.right = right

class Union(RegexNode):
    def __init__(self, left, right): self.left =
      left; self.right = right

class Star(RegexNode):
    def __init__(self, child): self.child =
     child

class Plus(RegexNode):       # sugar: a+ = aa*
    def __init__(self, child): self.child =
     child

class Optional_(RegexNode):  # sugar: a? = a|eps
    def __init__(self, child): self.child =
     child


def desugar(node: RegexNode) -> RegexNode:
    """Rewrite Plus/Optional_ into core nodes (
     Concat/Union/Star)."""
    if isinstance(node, (Literal, Epsilon)):
        return node
    if isinstance(node, Concat):
        return Concat(desugar(node.left),
     desugar(node.right))
    if isinstance(node, Union):
        return Union(desugar(node.left), desugar
     (node.right))
    if isinstance(node, Star):
        return Star(desugar(node.child))
    if isinstance(node, Plus):
        d = desugar(node.child)
        return Concat(d, Star(deepcopy(d)))
    if isinstance(node, Optional_):
        return Union(desugar(node.child),
     Epsilon())
    raise ValueError(f"Unknown node {node}")


def regex_to_string(node: RegexNode, parent_prec
     : int = 0) -> str:
    """Convert AST back to a human-readable
     regex string."""
    if isinstance(node, Literal):   return node.
     char
    if isinstance(node, Epsilon):   return "eps"
    if isinstance(node, Star):
        inner = regex_to_string(node.child, 3)
        return f"({inner})*" if isinstance(node.
     child, (Concat, Union)) else f"{inner}*"
    if isinstance(node, Plus):
        inner = regex_to_string(node.child, 3)
        return f"({inner})+" if isinstance(node.
     child, (Concat, Union)) else f"{inner}+"
    if isinstance(node, Optional_):
        inner = regex_to_string(node.child, 3)
        return f"({inner})?" if isinstance(node.
     child, (Concat, Union)) else f"{inner}?"
    if isinstance(node, Concat):
        l = regex_to_string(node.left, 2)
        r = regex_to_string(node.right, 2)
        s = l + r
        return f"({s})" if parent_prec > 2 else
     s
    if isinstance(node, Union):
        l = regex_to_string(node.left, 1)
```

```python
        r = regex_to_string(node.right, 1)
        s = f"{l}|{r}"
        return f"({s})" if parent_prec > 1 else s
    return "?"


# ===================
#  2.  RANDOM REGEX GENERATOR  (tiered)
# ===================

ALPHABETS = {
    "binary": ["0", "1"],
    "ab":     ["a", "b"],
    "abc":    ["a", "b", "c"],
    "abcd":   ["a", "b", "c", "d"],
    "digits": ["0", "1", "2", "3"],
}


def random_regex(tier: int, alphabet: list, rng: random.Random,
                 depth: int = 0) -> RegexNode:
    """
    Generate a random regex AST.
    Tier controls allowed operators AND how aggressively we recurse.
    Higher tiers have:
      - deeper trees (more states)
      - higher star/concat probability
      - lower leaf probability at shallow depth
    """
    max_depth = {1: 4, 2: 5, 3: 6, 4: 7}[tier]

    # Leaf probability: lower at shallow depth for higher tiers.
    # This forces the tree to grow before terminating.
    base_leaf = {1: 0.40, 2: 0.32, 3: 0.25, 4: 0.20}[tier]
    leaf_prob = min(0.92, base_leaf + 0.14 * depth)

    if depth >= max_depth or rng.random() < leaf_prob:
        return Literal(rng.choice(alphabet))

    if tier == 1:
        # Pure concatenation + union only; produces linear DFAs.
        ops, weights = ["concat", "union", "concat3"], [0.50, 0.35, 0.15]
    elif tier == 2:
        # Introduce closures; tends to produce small looping DFAs.
        ops, weights = (
            ["concat", "union", "star", "plus", "optional", "concat3"],
            [0.28, 0.20, 0.22, 0.14, 0.08, 0.08]
        )
    elif tier == 3:
        # Mix: closures + longer concatenations -> mid-size DFAs.
        ops, weights = (
            ["concat", "union", "star", "plus", "concat3", "star_union"],
            [0.28, 0.20, 0.22, 0.12, 0.10, 0.08]
        )
    else:
        # Deeply nested: closures of concatenations -> larger DFAs.
        ops, weights = (
            ["concat", "union", "star", "plus", "concat3", "star_union", "star_concat"],
            [0.22, 0.18, 0.22, 0.12, 0.10, 0.08, 0.08]
        )

    op = rng.choices(ops, weights=weights)[0]

    if op == "concat":
        return Concat(
            random_regex(tier, alphabet, rng, depth + 1),
            random_regex(tier, alphabet, rng, depth + 1))
    if op == "concat3":
        # Three-way concatenation: produces longer accepted strings -> more DFA states.
        n1 = random_regex(tier, alphabet, rng, depth + 1)
        n2 = random_regex(tier, alphabet, rng, depth + 1)
        n3 = random_regex(tier, alphabet, rng, depth + 1)
        return Concat(Concat(n1, n2), n3)
    if op == "union":
        return Union(
            random_regex(tier, alphabet, rng, depth + 1),
            random_regex(tier, alphabet, rng, depth + 1))
    if op == "star_union":
        # (X|Y)* style: very common pattern, produces interesting DFAs.
        return Star(Union(
            random_regex(tier, alphabet, rng, depth + 1),
            random_regex(tier, alphabet, rng, depth + 1)))
    if op == "star_concat":
        # (XY)* style: cyclic DFAs with 2+ states per cycle.
        return Star(Concat(
            random_regex(tier, alphabet, rng, depth + 1),
            random_regex(tier, alphabet, rng, depth + 1)))
    child = random_regex(tier, alphabet, rng, depth + 1)
    if op == "star":     return Star(child)
    if op == "plus":     return Plus(child)
    return Optional_(child)


# ===================
#  3.  THOMPSON'S NFA CONSTRUCTION
# ===================

class NFAState:
    _counter = 0

    def __init__(self):
        NFAState._counter += 1
        self.id = NFAState._counter
        self.transitions: dict = defaultdict(list)
        self.epsilon_transitions: list = []

    def __repr__(self):  return f"q{self.id}"
    def __hash__(self):  return hash(self.id)
    def __eq__(self, o): return isinstance(o, NFAState) and self.id == o.id


class NFA:
    def __init__(self, start: NFAState, accept: NFAState):
        self.start  = start
        self.accept = accept   # single accept state (Thompson invariant)

    def all_states(self):
        visited, stack = set(), [self.start]
        while stack:
            s = stack.pop()
            if s in visited: continue
            visited.add(s)
            for tgts in s.transitions.values():
                stack.extend(t for t in tgts if t not in visited)
            stack.extend(t for t in s.epsilon_transitions if t not in visited)
        return visited


def thompson(node: RegexNode) -> NFA:
```

```python
    """Build NFA from desugared regex AST using
     Thompson's construction."""
    if isinstance(node, Literal):
        s, a = NFAState(), NFAState()
        s.transitions[node.char].append(a)
        return NFA(s, a)

    if isinstance(node, Epsilon):
        s, a = NFAState(), NFAState()
        s.epsilon_transitions.append(a)
        return NFA(s, a)

    if isinstance(node, Concat):
        n1 = thompson(node.left)
        n2 = thompson(node.right)
        n1.accept.epsilon_transitions.append(n2.
     start)
        return NFA(n1.start, n2.accept)

    if isinstance(node, Union):
        n1 = thompson(node.left)
        n2 = thompson(node.right)
        s, a = NFAState(), NFAState()
        s.epsilon_transitions.extend([n1.start,
     n2.start])
        n1.accept.epsilon_transitions.append(a)
        n2.accept.epsilon_transitions.append(a)
        return NFA(s, a)

    if isinstance(node, Star):
        n = thompson(node.child)
        s, a = NFAState(), NFAState()
        s.epsilon_transitions.extend([n.start, a
     ])
        n.accept.epsilon_transitions.extend([n.
     start, a])
        return NFA(s, a)

    raise ValueError(f"thompson: unexpected node
      type {type(node)}")


# ===================
#  4.  EPSILON-CLOSURE + SUBSET CONSTRUCTION
# ===================

def epsilon_closure(states: frozenset) ->
     frozenset:
    stack, closure = list(states), set(states)
    while stack:
        s = stack.pop()
        for t in s.epsilon_transitions:
            if t not in closure:
                closure.add(t)
                stack.append(t)
    return frozenset(closure)


def nfa_move(states: frozenset, char: str) ->
     frozenset:
    result = set()
    for s in states:
        result.update(s.transitions.get(char,
     []))
    return frozenset(result)


class DFA:
    def __init__(self):
        self.states        = []
        self.state_index   = {}
        self.transitions   = {}   # (state_idx,
     char) -> state_idx
        self.start         = 0
        self.accept_states = set()
        self.alphabet      = []

    def accepts(self, s: str) -> bool:
        cur = self.start
        for c in s:
            if (cur, c) not in self.transitions:
                return False
            cur = self.transitions[(cur, c)]
        return cur in self.accept_states
```

```python
def subset_construction(nfa: NFA, alphabet: list
     ) -> DFA:
    dfa = DFA()
    dfa.alphabet = sorted(alphabet)

    start_closure = epsilon_closure(frozenset([
     nfa.start]))
    dfa.states.append(start_closure)
    dfa.state_index[start_closure] = 0
    if nfa.accept in start_closure:
        dfa.accept_states.add(0)

    queue = deque([start_closure])
    while queue:
        current = queue.popleft()
        cur_idx = dfa.state_index[current]
        for char in dfa.alphabet:
            nxt = epsilon_closure(nfa_move(
     current, char))
            if not nxt:
                continue
            if nxt not in dfa.state_index:
                idx = len(dfa.states)
                dfa.states.append(nxt)
                dfa.state_index[nxt] = idx
                if nfa.accept in nxt:
                    dfa.accept_states.add(idx)
                queue.append(nxt)
            dfa.transitions[(cur_idx, char)] =
     dfa.state_index[nxt]

    return dfa


# ===================
#  5.  DFA MINIMISATION -- Hopcroft's procedure
#      Completely rewritten; previous version
     had remap bugs.
# ===================

def minimise_dfa(dfa: DFA) -> tuple:
    """
    Returns (min_dfa, partition_log).
      min_dfa       : new DFA with states
     renumbered 0..k-1, start = 0
      partition_log : list[str] of refinement
     steps for CoT
    """
    n    = len(dfa.states)
    DEAD = n   # virtual trap/dead state

    # Complete transition table.
    full = {}
    for s in range(n):
        for c in dfa.alphabet:
            full[(s, c)] = dfa.transitions.get((
     s, c), DEAD)
    for c in dfa.alphabet:
        full[(DEAD, c)] = DEAD

    all_states = set(range(n + 1))   # 0..n-1
     live, n dead

    accepting     = frozenset(dfa.accept_states)
    non_accepting = frozenset(s for s in range(n
     ) if s not in dfa.accept_states)
    dead_group    = frozenset({DEAD})

    P = set()
    if accepting:     P.add(accepting)
    if non_accepting: P.add(non_accepting)
    P.add(dead_group)

    partition_log = []
    step_num = 0

    def pstr(partition):
        parts = []
        for g in sorted(partition, key=lambda g:
      min(g)):
            labels = ", ".join(
```

```
                f"D{s}" if s != DEAD else "dead"
  for s in sorted(g))
        parts.append("{" + labels + "}")
    return "[" + ", ".join(parts) + "]"

partition_log.append(f"Initial partition P =
  {pstr(P)}")

W = set(P)
while W:
    A = W.pop()
    for c in dfa.alphabet:
        X = frozenset(s for s in all_states
 if full[(s, c)] in A)
        if not X:
            continue
        new_P = set()
        changed = False
        for Y in P:
            inter = Y & X
            diff  = Y - X
            if inter and diff:
                new_P.add(inter)
                new_P.add(diff)
                changed = True
                step_num += 1
                partition_log.append(
                    f"  Split on '{c}' (
 splitter={pstr({A})}): "
                    f"{pstr({Y})} -> {pstr({
 inter})} and {pstr({diff})}"
                )
                if Y in W:
                    W.discard(Y)
                    W.add(inter)
                    W.add(diff)
                else:
                    W.add(inter if len(inter
 ) <= len(diff) else diff)
            else:
                new_P.add(Y)
        if changed:
            P = new_P

# Drop the dead group.
live_groups = [g for g in P if g !=
 dead_group]

# Map every live old-state to its group
 index.
state_to_gi: dict = {}
for gi, group in enumerate(live_groups):
    for s in group:
        state_to_gi[s] = gi

# Choose representative = lowest-numbered
 original state in each group.
def rep(gi: int) -> int:
    return min(s for s in live_groups[gi] if
  s != DEAD)

# BFS from start group to get a clean 0-
 indexed renaming.
start_gi = state_to_gi[dfa.start]
gi_to_new: dict = {}
queue2 = deque([start_gi])
visited2: set = set()
counter = 0

while queue2:
    gi = queue2.popleft()
    if gi in visited2: continue
    visited2.add(gi)
    gi_to_new[gi] = counter
    counter += 1
    r = rep(gi)
    for c in dfa.alphabet:
        tgt = full[(r, c)]
        if tgt != DEAD:
            tgt_gi = state_to_gi.get(tgt)
            if tgt_gi is not None and tgt_gi
  not in visited2:
                queue2.append(tgt_gi)
```

```
    # Any unreachable live group (defensive).
    for gi in range(len(live_groups)):
        if gi not in gi_to_new:
            gi_to_new[gi] = counter
            counter += 1

    # Build the minimised DFA.
    min_dfa = DFA()
    min_dfa.alphabet = dfa.alphabet
    min_dfa.states   = list(range(len(
     live_groups)))
    min_dfa.start    = 0

    for gi, group in enumerate(live_groups):
        new_idx = gi_to_new[gi]
        r       = rep(gi)
        # Accept state: if any member was an
     accept state.
        if any(s in dfa.accept_states for s in
     group if s != DEAD):
            min_dfa.accept_states.add(new_idx)
        # Transitions.
        for c in dfa.alphabet:
            tgt = full[(r, c)]
            if tgt != DEAD:
                tgt_gi = state_to_gi.get(tgt)
                if tgt_gi is not None:
                    min_dfa.transitions[(new_idx
     , c)] = gi_to_new[tgt_gi]

    if step_num == 0:
        partition_log.append(
            "  No splits occurred -- the DFA is
     already minimal.")
    partition_log.append(
        f"Final: {len(live_groups)} equivalence
     classes "
        f"-> minimised DFA has {len(live_groups)
     } states."
    )

    return min_dfa, partition_log


# ===================
#  6.  VERIFICATION (NFA == DFA == minDFA on all
      strings <= len 7)
# ===================

def strings_up_to_length(alphabet: list, max_len
     : int):
    for length in range(0, max_len + 1):
        for combo in itertools.product(alphabet,
      repeat=length):
            yield "".join(combo)


def nfa_accepts_str(nfa: NFA, s: str) -> bool:
    states = epsilon_closure(frozenset([nfa.
     start]))
    for c in s:
        states = epsilon_closure(nfa_move(states
     , c))
    return nfa.accept in states


def verify_all(nfa: NFA, dfa: DFA, min_dfa: DFA,
      alphabet: list,
               max_len: int = 7) -> tuple:
    """Returns (ok: bool, error_msg: str)."""
    for s in strings_up_to_length(alphabet,
     max_len):
        a = nfa_accepts_str(nfa, s)
        b = dfa.accepts(s)
        c = min_dfa.accepts(s)
        if a != b:
            return False, f"NFA/DFA mismatch on
     '{s}': NFA={a}, DFA={b}"
        if a != c:
            return False, f"NFA/minDFA mismatch
     on '{s}': NFA={a}, minDFA={c}"
    return True, ""
```

```python
# ===================
#  7.  CoT TRACE BUILDER
# ===================

def fmt_q(s: NFAState) -> str: return f"q{s.id}"
def fmt_D(i: int) -> str:      return f"D{i}"

def fmt_nfa_set(fs: frozenset) -> str:
    ids = sorted(s.id for s in fs)
    return "{" + ", ".join(f"q{i}" for i in ids)
      + "}"


def build_cot_trace(
    regex_str: str,
    ast_raw:   RegexNode,
    nfa:       NFA,
    dfa:       DFA,
    min_dfa:   DFA,
    partition_log: list,
    alphabet:  list,
) -> str:
    lines = []
    sa = sorted(alphabet)

    # Step 1: Structure
    lines += [
        "## Step 1: Analyse the Regular
     Expression\n",
        f"Given regex: **{regex_str}**\n",
        f"Alphabet S = {{{', '.join(sa)}}}\n",
        "Parse tree structure:",
        _describe_structure(ast_raw),
    ]

    # Step 2: Thompson's Rules
    lines += ["", "## Step 2: Thompson's
     Construction Rules Applied\n"]
    rules = _collect_thompson_rules(ast_raw)
    lines.append("We apply the following
     Thompson construction rules:\n")
    for r in rules:
        lines.append(f"  - {r}")

    # Step 3: NFA State Listing
    nfa_states = sorted(nfa.all_states(), key=
     lambda s: s.id)
    lines += [
        "",
        "## Step 3: NFA States and Transitions\n
     ",
        f"NFA states: {{{', '.join(fmt_q(s) for
     s in nfa_states)}}}",
        f"Start state: {fmt_q(nfa.start)}",
        f"Accept state: {fmt_q(nfa.accept)}\n",
        "Transition function d_NFA:\n",
        "| State | Symbol | Next States |",
        "|-------|--------|-------------|",
    ]
    for s in nfa_states:
        for char, targets in sorted(s.
     transitions.items()):
            tstr = "{" + ", ".join(
                fmt_q(t) for t in sorted(targets
     , key=lambda x: x.id)) + "}"
            lines.append(f"| {fmt_q(s)} | {char}
      | {tstr} |")
        if s.epsilon_transitions:
            tstr = "{" + ", ".join(
                fmt_q(t) for t in sorted(
                    s.epsilon_transitions, key=
     lambda x: x.id)) + "}"
            lines.append(f"| {fmt_q(s)} | eps |
     {tstr} |")
    lines.append("")

    # Step 4: epsilon-closure of start
    start_ec = epsilon_closure(frozenset([nfa.
     start]))
    lines += [
        "## Step 4: Compute epsilon-closure of
     Start State\n",
```

```python
        f"eps-closure({fmt_q(nfa.start)}) = {
     fmt_nfa_set(start_ec)}",
        "This becomes DFA start state D0.\n",
    ]
    _explain_eps_closure(nfa.start, lines)

    # Step 5: Subset Construction Table
    lines += [
        "",
        "## Step 5: Subset Construction (
     Powerset Construction)\n",
        "We expand each DFA state by computing
     move() then "
        "eps-closure for each symbol.\n",
    ]
    header = ("| DFA State | NFA States | "
              + " | ".join(sa) + " | Accept? |")
    sep    = ("|-----------|------------|"
              + "|".join(["--------"] * len(sa))
      + "|---------|")
    lines += [header, sep]

    for i, nfa_set in enumerate(dfa.states):
        row = []
        for c in sa:
            t = dfa.transitions.get((i, c))
            row.append(fmt_D(t) if t is not None
      else "empty")
        acc = "Y" if i in dfa.accept_states else
      ""
        lines.append(
            f"| {fmt_D(i)} | {fmt_nfa_set(
     nfa_set)} | "
            + " | ".join(row) + f" | {acc} |")

    lines += [
        "",
        f"Total DFA states before minimisation:
     **{len(dfa.states)}**",
        f"Start state: {fmt_D(dfa.start)}",
        f"Accept states: {{{', '.join(fmt_D(i)
     for i in sorted(dfa.accept_states))}}}\n",
    ]

    # Step 5b: Detailed walkthrough
    lines.append("### Subset Construction
     Walkthrough\n")
    for i, nfa_set in enumerate(dfa.states[:min
     (8, len(dfa.states))]):
        lines.append(f"**{fmt_D(i)}** = {
     fmt_nfa_set(nfa_set)}:")
        for c in sa:
            moved = nfa_move(nfa_set, c)
            ec    = epsilon_closure(moved)
            t     = dfa.transitions.get((i, c))
            if ec:
                lines.append(
                    f"  - On '{c}': move = {
     fmt_nfa_set(moved)}, "
                    f"eps-closure = {fmt_nfa_set
     (ec)} -> {fmt_D(t)}"
                )
            else:
                lines.append(
                    f"  - On '{c}': move = empty
      "
                    f"-> dead/trap state (no
     transition recorded)")
        lines.append("")
    if len(dfa.states) > 8:
        lines.append(
            f"*(Remaining {len(dfa.states) - 8}
     states omitted for brevity.)*\n")

    # Step 6: Minimisation
    lines += [
        "## Step 6: DFA Minimisation (Hopcroft's
      Procedure)\n",
        "**Partition refinement trace:**\n",
    ]
    for entry in partition_log:
        lines.append(entry)
    lines.append("")

    if len(min_dfa.states) == len(dfa.states):
```

```
        lines.append(
            "The DFA is **already minimal** --
 all states are distinguishable. "
            "The minimised DFA is structurally
 identical to the "
            "pre-minimisation DFA.\n"
        )
    else:
        lines.append(
            f"Minimisation reduced **{len(dfa.
 states)} -> "
            f"{len(min_dfa.states)} states** "
            f"by merging indistinguishable
 equivalence classes.\n"
        )

    # Step 7: Final Minimised DFA
    lines += [
        "## Step 7: Final Minimised DFA\n",
        f"States Q = {{{', '.join(fmt_D(i) for i
  in range(len(min_dfa.states)))}}}",
        f"Start state: {fmt_D(min_dfa.start)}",
        f"Accept states F = "
        f"{{{', '.join(fmt_D(i) for i in sorted(
 min_dfa.accept_states))}}}",
        f"Alphabet S = {{{', '.join(sa)}}}\n",
        "Transition function d (-- = no
 transition / leads to dead state):\n",
    ]
    h2 = "| State | " + " | ".join(sa) + " |
 Accept? |"
    s2 = "|-------|" + "|".join(["-------"] *
 len(sa)) + "|---------|"
    lines += [h2, s2]
    for s in range(len(min_dfa.states)):
        row = []
        for c in sa:
            t = min_dfa.transitions.get((s, c))
            row.append(fmt_D(t) if t is not None
  else "--")
        acc = "Y" if s in min_dfa.accept_states
 else ""
        lines.append(f"| {fmt_D(s)} | " + " | ".
 join(row) + f" | {acc} |")
    lines.append("")

    # Step 8: Verification
    lines += [
        "## Step 8: Verification\n",
        "We test the minimised DFA against
 representative strings "
        "(both accepted Y and rejected N):\n",
        "| String | Result | State Trace |",
        "|--------|--------|-------------|",
    ]
    for word, accepted, trace in
 _verification_examples(min_dfa, alphabet):
        label = "Y Accept" if accepted else "N
 Reject"
        disp  = f'"{word}"' if word else '"eps"
 (empty string)'
        lines.append(f"| {disp} | {label} | {
 trace} |")

    # Summary
    sample = _sample_accepted(min_dfa, alphabet)
    lines += [
        "",
        "## Summary\n",
        f"The regex '{regex_str}' over S = {{{',
  '.join(sa)}}} is recognised "
        f"by a minimised DFA with:",
        f"  - **{len(min_dfa.states)} states**:
 "
        + ", ".join(fmt_D(i) for i in range(len(
 min_dfa.states))),
        f"  - **Start state**: {fmt_D(min_dfa.
 start)}",
        f"  - **Accept state(s)**: "
        + (", ".join(fmt_D(i) for i in sorted(
 min_dfa.accept_states))
           or "none"),
        f"  - **{len(min_dfa.transitions)}
 transition(s)** total",
```

```
        f"  - Example accepted strings: {sample}
 ",
    ]

    return "\n".join(lines)


# -- helpers ---------

def _sample_accepted(dfa: DFA, alphabet: list)
 -> str:
    found = []
    for length in range(0, 8):
        for combo in itertools.product(alphabet,
  repeat=length):
            word = "".join(combo)
            if dfa.accepts(word):
                found.append(f'"{word}"' if word
  else '"eps"')
        if len(found) >= 3:
            break
    return ", ".join(found) if found else "none
 (empty language)"


def _describe_structure(node: RegexNode, depth:
 int = 0) -> str:
    pad = "  " * depth
    if isinstance(node, Literal):   return f"{
 pad}Literal '{node.char}'"
    if isinstance(node, Epsilon):   return f"{
 pad}eps (epsilon)"
    if isinstance(node, Star):
        return (f"{pad}Kleene Star (*)\n"
                f"{_describe_structure(node.
 child, depth + 1)}")
    if isinstance(node, Plus):
        return (f"{pad}One-or-more (+)\n"
                f"{_describe_structure(node.
 child, depth + 1)}")
    if isinstance(node, Optional_):
        return (f"{pad}Optional (?)\n"
                f"{_describe_structure(node.
 child, depth + 1)}")
    if isinstance(node, Concat):
        return (f"{pad}Concatenation\n"
                f"{_describe_structure(node.left
 ,  depth + 1)}\n"
                f"{_describe_structure(node.
 right, depth + 1)}")
    if isinstance(node, Union):
        return (f"{pad}Union (|)\n"
                f"{_describe_structure(node.left
 ,  depth + 1)}\n"
                f"{_describe_structure(node.
 right, depth + 1)}")
    return f"{pad}?"


def _collect_thompson_rules(node: RegexNode,
 seen: set = None) -> list:
    if seen is None: seen = set()
    rules = []
    key = type(node).__name__
    if key not in seen:
        seen.add(key)
        if isinstance(node, Literal):
            rules.append(
                f"**Symbol rule** for '{node.
 char}': "
                f"two states s, a with s --'{
 node.char}'--> a"
            )
        if isinstance(node, Epsilon):
            rules.append(
                "**Epsilon rule**: two states s,
  a with s --eps--> a")
        if isinstance(node, Concat):
            rules.append(
                "**Concatenation rule**: build N
 (r1) and N(r2) independently, "
                "then add eps-transition from N(
 r1)'s accept to N(r2)'s start"
```

```
            )
        if isinstance(node, Union):
            rules.append(
                "**Union rule**: new start q0
 with eps-transitions to "
                "N(r1) and N(r2) starts; both
 accepts eps-connect to new accept qa"
            )
        if isinstance(node, Star):
            rules.append(
                "**Kleene Star rule**: new start
  q0 and accept qa; "
                "q0 --eps--> N(r).start and q0
 --eps--> qa (allow skip); "
                "N(r).accept --eps--> N(r).start
  (loop) and --eps--> qa (exit)"
            )
    children = (
        [node.left, node.right] if isinstance(
 node, (Concat, Union))
        else [node.child] if isinstance(node, (
 Star, Plus, Optional_))
        else []
    )
    for child in children:
        rules += _collect_thompson_rules(child,
 seen)
    return rules


def _explain_eps_closure(start: NFAState, lines:
  list):
    visited, queue, steps = {start}, deque([
 start]), []
    while queue:
        s = queue.popleft()
        for t in s.epsilon_transitions:
            if t not in visited:
                visited.add(t)
                queue.append(t)
                steps.append(f"  {fmt_q(s)} --
 eps--> {fmt_q(t)}")
    if steps:
        lines.append("eps-closure BFS expansion:
 ")
        lines.extend(steps)
    else:
        lines.append(
            "  (Start state has no outgoing eps-
 transitions; "
            "eps-closure = {start state})")


def _verification_examples(min_dfa: DFA,
 alphabet: list) -> list:
    """Up to 3 accepted + 3 rejected strings,
 with full state traces."""
    accepted, rejected = [], []
    for length in range(0, 9):
        for combo in itertools.product(alphabet,
  repeat=length):
            word = "".join(combo)
            if min_dfa.accepts(word) and len(
 accepted) < 3:
                accepted.append(word)
            elif not min_dfa.accepts(word) and
 len(rejected) < 3:
                rejected.append(word)
        if len(accepted) >= 3 and len(rejected)
 >= 3:
            break

    results = []
    for word in accepted + rejected:
        cur, parts, dead = min_dfa.start, [fmt_D
 (min_dfa.start)], False
        for c in word:
            nxt = min_dfa.transitions.get((cur,
 c))
            if nxt is None:
                parts.append(f"--{c}-->empty")
                dead = True
                break
```

```
                parts.append(f"--{c}-->{fmt_D(nxt)}"
 )
                cur = nxt
            ok = (not dead) and (cur in min_dfa.
 accept_states)
            results.append((word, ok, " ".join(parts
 )))
    return results


# ===================
#  8.  QWEN ChatML FORMATTER
# ===================

SYSTEM_PROMPT = (
    "You are an expert in formal language theory
     and automata. "
    "When given a regular expression, you
     convert it to a Deterministic "
    "Finite Automaton (DFA) using Thompson's
     construction to build an NFA, "
    "followed by the subset (powerset)
     construction to convert to a DFA, "
    "and finally Hopcroft's procedure to
     minimise the DFA. "
    "You always show your full working step by
     step."
)

USER_TEMPLATES = [
    "Convert the regular expression '{regex}'
     over the alphabet S = {{{alpha}}} "
    "to a minimised DFA. Show all steps.",
    "Given the regex '{regex}' with alphabet {{{
     alpha}}}, construct a minimised "
    "DFA using Thompson's construction followed
     by subset construction. "
    "Show your full work.",
    "Use Thompson's NFA construction and the
     powerset construction to build a "
    "DFA for the regex '{regex}' (S = {{{alpha
     }}}), then minimise it using "
    "Hopcroft's procedure.",
    "Construct a minimised DFA that accepts
     exactly the language described by "
    "the regular expression '{regex}' over S =
     {{{alpha}}}. Show the NFA, DFA, "
    "and minimised DFA.",
    (
        "For the regular expression '{regex}'
 over alphabet {{{alpha}}}, perform:\n"
        "1. Thompson's NFA construction\n"
        "2. epsilon-closure computation\n"
        "3. Subset construction to DFA\n"
        "4. DFA minimisation\n"
        "Show all intermediate steps."
    ),
]


def format_qwen_chatml(regex_str: str, alphabet:
  list, cot: str,
                       rng: random.Random) ->
 dict:
    alpha_str = ", ".join(sorted(alphabet))
    user_msg  = rng.choice(USER_TEMPLATES).
 format(
        regex=regex_str, alpha=alpha_str)
    return {
        "messages": [
            {"role": "system",    "content":
 SYSTEM_PROMPT},
            {"role": "user",      "content":
 user_msg},
            {"role": "assistant", "content": cot
 },
        ]
    }


# ===================
#  9.  GENERATION PIPELINE
# ===================

TIER_DIST = {1: 0.15, 2: 0.25, 3: 0.35, 4: 0.25}
```

```python
MAX_DFA_STATES     = 24
MAX_MIN_DFA_STATES = 18

# Minimum min-DFA states by tier: enforces
     complexity per tier.
MIN_DFA_BY_TIER = {1: 2, 2: 2, 3: 3, 4: 4}

HANDCRAFTED = [
    # Classic textbook examples
    ("(a|b)*abb",               ["a", "b"]),
    ("(a|b)*a(a|b)",            ["a", "b"]),
    ("(0|1)*00",                ["0", "1"]),
    ("(0|1)*0(0|1)",            ["0", "1"]),
    ("(0|1)*101",               ["0", "1"]),
    ("(a|b)*ab",                ["a", "b"]),
    ("(a|b)*aabb(a|b)*",        ["a", "b"]),
    ("(a|b)*bab",               ["a", "b"]),
    ("(0|1)*010(0|1)*",         ["0", "1"]),
    ("(0|1)*11(0|1)*",          ["0", "1"]),
    # Kleene / closure patterns
    ("a*b*",                    ["a", "b"]),
    ("a*ba*",                   ["a", "b"]),
    ("(aa|b)*",                 ["a", "b"]),
    ("(0|1)*(01)(0|1)*",        ["0", "1"]),
    ("(ab|ba)*",                ["a", "b"]),
    ("(ab|cd)*",                ["a", "b", "c", "
     d"]),
    ("(abc|bca|cab)*",          ["a", "b", "c"]),
    ("(aab|b)*",                ["a", "b"]),
    ("(a|bb)*",                 ["a", "b"]),
    # Plus and optional
    ("a+b+",                    ["a", "b"]),
    ("(ab)+",                   ["a", "b"]),
    ("a*(b|c)+a*",              ["a", "b", "c"]),
    ("a?b?c?",                  ["a", "b", "c"]),
    ("(a|b)?c",                 ["a", "b", "c"]),
    # Pure concatenation
    ("abbba",                   ["a", "b"]),
    ("1011",                    ["0", "1"]),
    ("abcd",                    ["a", "b", "c", "
     d"]),
    ("0110",                    ["0", "1"]),
    ("abab",                    ["a", "b"]),
    # Multi-symbol alphabet complexity
    ("(a|b|c)*abc",             ["a", "b", "c"]),
    ("a(b|c)*",                 ["a", "b", "c"]),
    # Harder patterns
    ("(a|b)*aba(a|b)*",         ["a", "b"]),
    ("(0|1)*001(0|1)*",         ["0", "1"]),
    ("(a|b)*abba(a|b)*",        ["a", "b"]),
    ("(0|1)*0101",              ["0", "1"]),
    ("(a|b)(a|b)(a|b)",         ["a", "b"]),
    ("(0|1)*(00|11)(0|1)*",     ["0", "1"]),
    ("a(ba)*",                  ["a", "b"]),
    ("(ab)*(ba)*",              ["a", "b"]),
    # Strings of even / divisible length
    ("(aa)*",                   ["a", "b"]),
    ("(aaa)*",                  ["a", "b"]),
    ("(ab|ba|aa|bb)*",          ["a", "b"]),
    # Contains specific substring
    ("(a|b)*aa(a|b)*",          ["a", "b"]),
    ("(0|1)*00(0|1)*",          ["0", "1"]),
    ("(a|b)*bb(a|b)*",          ["a", "b"]),
]


class _Parser:
    def __init__(self, s): self.s = s; self.pos
     = 0

    def peek(self):
        return self.s[self.pos] if self.pos <
     len(self.s) else None

    def consume(self, c=None):
        ch = self.s[self.pos]
        if c and ch != c:
            raise ValueError(f"Expected {c!r}
     got {ch!r}")
        self.pos += 1
        return ch

    def parse(self):
        node = self._union()
        if self.pos != len(self.s):
            raise ValueError(f"Leftover at {self.
     pos}")
        return node

    def _union(self):
        left = self._concat()
        while self.peek() == '|':
            self.consume('|')
            left = Union(left, self._concat())
        return left

    def _concat(self):
        nodes = []
        while self.peek() not in (None, ')', '|'
     ):
            nodes.append(self._quantified())
        if not nodes:
            raise ValueError("Empty concat")
        result = nodes[0]
        for n in nodes[1:]:
            result = Concat(result, n)
        return result

    def _quantified(self):
        base = self._atom()
        q = self.peek()
        if q == '*': self.consume(); return Star
     (base)
        if q == '+': self.consume(); return Plus
     (base)
        if q == '?': self.consume(); return
     Optional_(base)
        return base

    def _atom(self):
        c = self.peek()
        if c == '(':
            self.consume('(')
            node = self._union()
            self.consume(')')
            return node
        if c and c not in (')', '|', '*', '+', '
     ?'):
            self.consume()
            return Literal(c)
        raise ValueError(f"Unexpected char {c!r}
     ")


def parse_regex(s: str) -> RegexNode:
    return _Parser(s).parse()


def _build_entry(regex_str: str, ast_raw:
     RegexNode, alphabet: list,
                 rng: random.Random, tier) ->
     Optional[dict]:
    NFAState._counter = 0
    try:
        ast     = desugar(ast_raw)
        nfa     = thompson(ast)
        dfa     = subset_construction(nfa,
     alphabet)
        if len(dfa.states) > MAX_DFA_STATES:
            return None
        min_dfa, plog = minimise_dfa(dfa)
        min_states_floor = (MIN_DFA_BY_TIER.get(
     tier, 2)
                            if isinstance(tier,
     int) else 2)
        if (len(min_dfa.states) >
     MAX_MIN_DFA_STATES
                or len(min_dfa.states) <
     min_states_floor):
            return None
    except Exception:
        return None

    ok, err = verify_all(nfa, dfa, min_dfa,
     alphabet, max_len=7)
    if not ok:
        # Silently skip: verification failure
     means a bug slipped through.
        return None
```

```python
    cot   = build_cot_trace(
        regex_str, ast_raw, nfa, dfa, min_dfa,
     plog, alphabet)
    entry = format_qwen_chatml(regex_str,
     alphabet, cot, rng)
    entry["metadata"] = {
        "regex":          regex_str,
        "alphabet":       sorted(alphabet),
        "tier":           tier,
        "nfa_states":     len(nfa.all_states()),
        "dfa_states":     len(dfa.states),
        "min_dfa_states": len(min_dfa.states),
        "accept_states":  sorted(min_dfa.
     accept_states),
        "transitions":    len(min_dfa.
     transitions),
        "hash":           hashlib.md5(
            (regex_str + str(sorted(alphabet))).
     encode()
        ).hexdigest()[:8],
    }
    return entry


def generate_handcrafted(rng: random.Random) ->
     list:
    entries = []
    for regex_str, alphabet in HANDCRAFTED:
        try:
            ast_raw = parse_regex(regex_str)
        except Exception as e:
            print(f"  [WARN] Parse failed '{
     regex_str}': {e}")
            continue
        entry = _build_entry(regex_str, ast_raw,
      alphabet, rng, "handcrafted")
        if entry is None:
            print(f"  [WARN] Build/verify failed
      for '{regex_str}'")
        else:
            entries.append(entry)
    return entries


def generate_random_entry(rng: random.Random,
     tier: int) -> Optional[dict]:
    alpha_name = rng.choice(list(ALPHABETS.keys
     ()))
    alphabet   = ALPHABETS[alpha_name]
    ast_raw    = random_regex(tier, alphabet,
     rng)
    regex_str  = regex_to_string(ast_raw)
    if len(regex_str) < 2:
        return None
    return _build_entry(regex_str, ast_raw,
     alphabet, rng, tier)


# ===================
#  10.  STATISTICS + ENTRY POINT
# ===================

def print_stats(dataset: list):
    print("\n" + "=" * 62)
    print("  DATASET STATISTICS")
    print("=" * 62)
    tiers, sc = {}, []
    for ex in dataset:
        m = ex["metadata"]
        tiers[m["tier"]] = tiers.get(m["tier"],
     0) + 1
        sc.append(m["min_dfa_states"])
    print(f"  Total examples   : {len(dataset)
     :,}")
    for t in sorted(tiers.keys(), key=str):
        print(f"  Tier {str(t):<13} : {tiers[t
     ]:,}")
    print(f"  Min DFA states   : avg={sum(sc)/
     len(sc):.2f}, "
          f"min={min(sc)}, max={max(sc)}")
    ac = {}
    for ex in dataset:
        k = tuple(ex["metadata"]["alphabet"])
        ac[k] = ac.get(k, 0) + 1
    print("  Alphabet dist    :")
    for k, v in sorted(ac.items(), key=lambda x:
      -x[1]):
        print(f"    {list(k)}: {v:,}")
    print("=" * 62 + "\n")


def main():
    ap = argparse.ArgumentParser(
        description="Generate Regex->DFA CoT
     dataset for Qwen LoRA fine-tuning"
    )
    ap.add_argument("--n",          type=int,
     default=25000)
    ap.add_argument("--seed",       type=int,
     default=42)
    ap.add_argument("--out",        type=str,
     default="regex_dfa_dataset.jsonl")
    ap.add_argument("--val_split",  type=float,
     default=0.05)
    ap.add_argument("--test_split", type=float,
     default=0.05)
    args = ap.parse_args()

    rng     = random.Random(args.seed)
    dataset = []
    seen    = set()

    print("Generating handcrafted examples...")
    for ex in generate_handcrafted(rng):
        h = ex["metadata"]["hash"]
        if h not in seen:
            seen.add(h)
            dataset.append(ex)
    print(f"  Added {len(dataset)} handcrafted
     examples.")

    print(f"Generating up to {args.n:,} random
     examples...")
    max_attempts = args.n * 25

    with tqdm(total=args.n) as pbar:
        pbar.update(len(dataset))
        attempts = 0
        while len(dataset) < args.n and attempts
      < max_attempts:
            attempts += 1
            tier  = rng.choices(
                list(TIER_DIST), weights=list(
     TIER_DIST.values()))[0]
            entry = generate_random_entry(rng,
     tier)
            if entry is None: continue
            h = entry["metadata"]["hash"]
            if h in seen: continue
            seen.add(h)
            dataset.append(entry)
            pbar.update(1)

    print(f"\nGenerated {len(dataset):,} unique
     examples "
          f"in {attempts:,} attempts.")
    print_stats(dataset)

    rng.shuffle(dataset)
    n       = len(dataset)
    n_val   = int(n * args.val_split)
    n_test  = int(n * args.test_split)
    n_train = n - n_val - n_test

    splits = {
        "train": dataset[:n_train],
        "val":   dataset[n_train:n_train + n_val
     ],
        "test":  dataset[n_train + n_val:],
    }
    base = args.out.replace(".jsonl", "")
    for name, data in splits.items():
        path = f"{base}_{name}.jsonl"
        with open(path, "w", encoding="utf-8")
     as f:
            for ex in data:
```

```
                f.write(json.dumps(ex,
     ensure_ascii=False) + "\n")
        print(f"  Wrote {len(data):,} examples
     -> {path}")

    with open(args.out, "w", encoding="utf-8")
     as f:
        for ex in dataset:
            f.write(json.dumps(ex, ensure_ascii=
     False) + "\n")
    print(f"\n  Full dataset -> {args.out}")
    print("\nDone")


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

G.10 Fine-Tuning Scripts

The CoT and No-CoT training scripts are structurally identical across all three model sizes, differing only in model path, output directory, and system prompt. We provide the 7B CoT script as the representative example, followed by a note on the differences for other sizes and formats.

Representative Script: Qwen 2.5-7B CoT Fine-Tuning.

train_qwen7b_lora_cot.py

```
import os
import time
import random
import numpy as np
import torch
import csv
from datasets import load_dataset,
     concatenate_datasets
from transformers import (
    AutoTokenizer,
    AutoModelForCausalLM,
    TrainingArguments,
    TrainerCallback,
    set_seed,
)
from peft import LoraConfig, get_peft_model
from trl import SFTTrainer
from tqdm import tqdm

# ===================
# CONFIGURATION & REPRODUCIBILITY
# ===================
SEED = 42
set_seed(SEED)

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed_all(SEED)

torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

os.environ["PYTHONHASHSEED"] = str(SEED)
os.environ["OMP_NUM_THREADS"] = "1"

MODEL_PATH = "Qwen/Qwen 2.5-7B-Instruct"

TRAIN_FILE =
     "dataset_reasoning/train_ps_10.jsonl"
IID_FILE   =
     "dataset_reasoning/iid_ps_final_10.jsonl"
OOD_FILE   =
     "dataset_reasoning/ood_ps_010.jsonl"

OUTPUT_DIR = "lora_output_qwen7b_cot"
```

```
MAX_SEQ_LEN = 8192

EVAL_STEPS = 200
SAVE_STEPS = 200
LOG_STEPS  = 10

IID_EVAL_SAMPLES = 200
OOD_EVAL_SAMPLES = 200

os.makedirs(OUTPUT_DIR, exist_ok=True)

print("=" * 80)
print(f"Random seed: {SEED} | GPU:
     {torch.cuda.get_device_name(0)}")
print("=" * 80)

# ===================
# SYSTEM PROMPT
# ===================
SYSTEM_PROMPT = (
    "You are a DFA construction assistant. "
    "You must output the TRACE and FINAL DFA
     exactly in the same format "
    "as training data. "
    "Do not output any explanation outside the
     TRACE markers."
)

# ===================
# CHAT TEMPLATE FORMATTER
# ===================
def build_chat_text(tokenizer, prompt, output):
    messages = [
        {"role": "system",    "content":
     SYSTEM_PROMPT},
        {"role": "user",      "content":
     prompt},
        {"role": "assistant", "content":
     output},
    ]
    return
     tokenizer.apply_chat_template(messages,
     tokenize=False)

def format_sample(example, tokenizer):
    prompt = example["prompt"]
    output = example["output"]
    text   = build_chat_text(tokenizer, prompt,
     output)
    return {"text": text}

# ===================
# CHECKPOINT FINDER
# ===================
def find_last_checkpoint(output_dir):
    if not os.path.isdir(output_dir):
        return None
    checkpoints = [
        d for d in os.listdir(output_dir)
        if d.startswith("checkpoint-")
    ]
    if not checkpoints:
        return None
    checkpoints = sorted(
        checkpoints, key=lambda x:
     int(x.split("-")[-1]))
    return os.path.join(output_dir,
     checkpoints[-1])

# ===================
# CALLBACKS (TQDM + CSV LOGGER)
# ===================
class MetricsLogger(TrainerCallback):
    def __init__(self, output_dir):
        self.metrics_file = os.path.join(
            output_dir, "training_metrics.csv")
        with open(self.metrics_file, "w",
     newline="") as f:
            csv.writer(f).writerow(
                ["step", "epoch", "loss",
     "eval_loss"])

    def on_log(self, args, state, control,
     logs=None, **kwargs):
        if logs:
```

```
            with open(self.metrics_file, "a",
     newline="") as f:
                csv.writer(f).writerow([
                    state.global_step,
                    round(state.epoch, 4) if
     state.epoch else "",
                    logs.get("loss", ""),
                    logs.get("eval_loss", "")
                ])

class TqdmCallback(TrainerCallback):
    def __init__(self):
        self.pbar = None

    def on_train_begin(self, args, state,
     control, **kwargs):
        self.pbar = tqdm(
            total=state.max_steps,
     desc="Training", unit="step")

    def on_step_end(self, args, state, control,
     **kwargs):
        if self.pbar:
            self.pbar.update(1)

    def on_train_end(self, args, state,
     control, **kwargs):
        if self.pbar:
            self.pbar.close()

# ===================
# MAIN
# ===================
def main():
    # ----------------------------
    # 1. LOAD TOKENIZER
    # ----------------------------
    print("Loading tokenizer...")
    tokenizer = AutoTokenizer.from_pretrained(
        MODEL_PATH,
        trust_remote_code=True,
        use_fast=True
    )
    if tokenizer.pad_token is None:
        tokenizer.pad_token =
     tokenizer.eos_token
    tokenizer.padding_side = "right"
    tokenizer.model_max_length = MAX_SEQ_LEN

    # ----------------------------
    # 2. LOAD DATASETS
    # ----------------------------
    print("Loading datasets...")
    train_data = load_dataset(
        "json", data_files=TRAIN_FILE,
     split="train")
    iid_data   = load_dataset(
        "json", data_files=IID_FILE,
     split="train")
    ood_data   = load_dataset(
        "json", data_files=OOD_FILE,
     split="train")
    print(f"Train: {len(train_data):,} | "
          f"IID: {len(iid_data):,} | "
          f"OOD: {len(ood_data):,}")

    # ----------------------------
    # 3. FORMAT INTO CHAT TEMPLATE
    # ----------------------------
    print("Formatting train dataset...")
    train_data = train_data.map(
        lambda x: format_sample(x, tokenizer),
        remove_columns=train_data.column_names,
        num_proc=4
    )
    print("Formatting IID dataset...")
    iid_data = iid_data.map(
        lambda x: format_sample(x, tokenizer),
        remove_columns=iid_data.column_names,
        num_proc=4
    )
    print("Formatting OOD dataset...")
    ood_data = ood_data.map(
        lambda x: format_sample(x, tokenizer),
        remove_columns=ood_data.column_names,
```

```
        num_proc=4
    )
    train_data = train_data.shuffle(seed=SEED)

    # ----------------------------
    # 4. BUILD EVAL SET (MERGED IID + OOD)
    # ----------------------------
    if len(iid_data) > IID_EVAL_SAMPLES:
        iid_eval =
     iid_data.shuffle(seed=SEED).select(
            range(IID_EVAL_SAMPLES))
    else:
        iid_eval = iid_data

    if len(ood_data) > OOD_EVAL_SAMPLES:
        ood_eval =
     ood_data.shuffle(seed=SEED).select(
            range(OOD_EVAL_SAMPLES))
    else:
        ood_eval = ood_data

    eval_dataset =
     concatenate_datasets([iid_eval, ood_eval])
    print(f"Eval dataset size:
     {len(eval_dataset):,}")

    # ----------------------------
    # 5. LOAD MODEL
    # ----------------------------
    print("Loading full bf16 Qwen model (LoRA
     training)...")
    model =
     AutoModelForCausalLM.from_pretrained(
        MODEL_PATH,
        torch_dtype=torch.bfloat16,
        device_map="auto",
        trust_remote_code=True,
        use_cache=False
    )
    model.gradient_checkpointing_enable()
    model.config.use_cache = False

    # ----------------------------
    # 6. APPLY LoRA
    # ----------------------------
    print("Applying LoRA...")
    lora_config = LoraConfig(
        r=32,
        lora_alpha=64,
        lora_dropout=0.05,
        bias="none",
        task_type="CAUSAL_LM",
        target_modules=[
            "q_proj", "k_proj", "v_proj",
     "o_proj",
            "gate_proj", "up_proj", "down_proj"
        ],
        use_dora=False,
    )
    model = get_peft_model(model, lora_config)
    model.print_trainable_parameters()

    # ----------------------------
    # 7. TRAINING ARGUMENTS
    # ----------------------------
    training_args = TrainingArguments(
        output_dir=OUTPUT_DIR,
        overwrite_output_dir=False,

        per_device_train_batch_size=1,
        per_device_eval_batch_size=1,
        gradient_accumulation_steps=8,

        num_train_epochs=2,
        learning_rate=2e-4,
        lr_scheduler_type="cosine",
        warmup_ratio=0.03,
        weight_decay=0.01,

        bf16=True,
        tf32=True,

        logging_steps=LOG_STEPS,

        eval_strategy="steps",
        eval_steps=EVAL_STEPS,
```

```
        save_strategy="steps",
        save_steps=SAVE_STEPS,
        save_total_limit=3,

        report_to="none",
        optim="adamw_torch_fused",

        gradient_checkpointing=True,
        remove_unused_columns=True,

        dataloader_num_workers=4,
        seed=SEED,
    )

    # ----------------------------
    # 8. TRAINER
    # ----------------------------
    trainer = SFTTrainer(
        model=model,
        tokenizer=tokenizer,
        train_dataset=train_data,
        eval_dataset=eval_dataset,
        args=training_args,
        dataset_text_field="text",
        max_seq_length=MAX_SEQ_LEN,
        packing=False,
        callbacks=[TqdmCallback(),
     MetricsLogger(OUTPUT_DIR)],
    )

    # ----------------------------
    # 9. TRAIN (WITH RESUME SUPPORT)
    # ----------------------------
    last_ckpt = find_last_checkpoint(OUTPUT_DIR)
    if last_ckpt:
        print(f"Resuming from checkpoint:
     {last_ckpt}")

    print("\nStarting Training...")
    t0 = time.time()

     trainer.train(resume_from_checkpoint=last_ckpt)
    print(f"\nTraining finished in "
          f"{(time.time() - t0) / 3600:.2f}
     hours")

    # ----------------------------
    # 10. SAVE
    # ----------------------------
    print("Saving LoRA adapter and
     tokenizer...")
    trainer.model.save_pretrained(OUTPUT_DIR)
    tokenizer.save_pretrained(OUTPUT_DIR)
    print(f"Done. Model saved to {OUTPUT_DIR}")


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

Differences for other model sizes and formats.

- **1.5B CoT**: *MODEL_PATH* set to *Qwen 2.5-1.5B-Instruct*, *OUTPUT_DIR* set to *lora_output_qwen15b_cot*.
- **14B CoT**: *MODEL_PATH* set to *Qwen 2.5-14B-Instruct*, *OUTPUT_DIR* set to *lora_output_qwen14b_cot*.
- **No-CoT variants**: System prompt changed to *"You are a DFA construction assistant. You must output ONLY the DFA in the exact required format. Do not provide any explanation, extra text, or commentary."* Training data changed to *final_data/train.jsonl*.

Generic Curriculum Training Script.

train_lora.py

```
"""
train_lora.py  LoRA SFT Training for Qwen 2.5
     RegexDFA
=======================
Supports: Qwen 2.5-1.5B-Instruct, Qwen 2.5-7B-
     Instruct, Qwen 2.5-14B-Instruct
Hardware: Optimised for H100 80GB (AU HPC, cn5/
     cn6)
"""

import os
import json
import math
import logging
import argparse
import random
from typing import List, Dict, Optional

import torch
from transformers import (
    AutoTokenizer,
    AutoModelForCausalLM,
    TrainingArguments,
    Trainer,
    DataCollatorForSeq2Seq,
    set_seed,
    EarlyStoppingCallback,
)
from peft import LoraConfig, TaskType,
     get_peft_model
from datasets import Dataset

logger = logging.getLogger(__name__)
logging.basicConfig(
    format="%(asctime)s | %(levelname)s | %(name
     )s | %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
    level=logging.INFO,
)


#
#  1. ARGUMENT PARSING
#

def parse_args():
    p = argparse.ArgumentParser(description="
     LoRA SFT  Qwen 2.5 RegexDFA")

    p.add_argument("--model_path",  required=
     True)
    p.add_argument("--train_file",  required=
     True)
    p.add_argument("--val_file",    required=
     True)
    p.add_argument("--output_dir",  default="./
     lora_output")
    p.add_argument("--replay_file", default=None
     )

    p.add_argument("--lora_r",           type=
     int,   default=32)
    p.add_argument("--lora_alpha",       type=
     int,   default=64)
    p.add_argument("--lora_dropout",     type=
     float, default=0.05)
    p.add_argument("--lora_target_modules",
     nargs="+",
                   default=["q_proj","k_proj","
     v_proj","o_proj",
                            "gate_proj","up_proj
     ","down_proj"])

    p.add_argument("--num_train_epochs",
         type=int,   default=3)
    p.add_argument("--
     per_device_train_batch_size", type=int,
     default=2)
    p.add_argument("--per_device_eval_batch_size
     ",  type=int,   default=2)
    p.add_argument("--
     gradient_accumulation_steps", type=int,
```

```
     default=8)
    p.add_argument("--learning_rate",
         type=float, default=2e-4)
    p.add_argument("--weight_decay",
         type=float, default=0.01)
    p.add_argument("--warmup_ratio",
         type=float, default=0.05)
    p.add_argument("--lr_scheduler_type",
         type=str,   default="cosine")
    p.add_argument("--max_seq_length",
         type=int,   default=4096)
    p.add_argument("--seed",
         type=int,   default=42)
    p.add_argument("--replay_ratio",
         type=float, default=0.10)

    p.add_argument("--bf16",    action="
     store_true", default=True)
    p.add_argument("--no_bf16", dest="bf16",
     action="store_false")
    p.add_argument("--fp16",    action="
     store_true", default=False)
    p.add_argument("--gradient_checkpointing",
     action="store_true", default=True)
    p.add_argument("--no_gradient_checkpointing"
     ,
                   dest="gradient_checkpointing"
     , action="store_false")

    p.add_argument("--save_steps",
      type=int, default=200)
    p.add_argument("--eval_steps",
      type=int, default=200)
    p.add_argument("--logging_steps",
      type=int, default=10)
    p.add_argument("--save_total_limit",
      type=int, default=3)
    p.add_argument("--load_best_model_at_end",
      action="store_true", default=True)
    p.add_argument("--early_stopping_patience",
      type=int, default=5)

    p.add_argument("--deepspeed",  type=str,
     default=None)
    p.add_argument("--report_to",  type=str,
     default="tensorboard",
                   choices=["tensorboard","wandb
     ","none"])
    p.add_argument("--run_name",   type=str,
     default=None)

    # CHANGE 1 OF 2: accept checkpoint path from
      submit script
    p.add_argument("--resume_from_checkpoint",
     type=str, default=None)

    args = p.parse_args()

    if args.fp16:
        args.bf16 = False

    if args.run_name is None:
        model_shortname = os.path.basename(args.
     model_path.rstrip("/"))
        args.run_name = f"{model_shortname}
     _lora_r{args.lora_r}"

    return args


#
#  2. DATA LOADING AND TOKENISATION
#

def load_jsonl(path: str) -> List[Dict]:
    data = []
    with open(path, "r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if line:
                data.append(json.loads(line))
    return data


def tokenise_example(example: Dict, tokenizer,
     max_seq_length: int) -> Optional[Dict]:
    messages = example["messages"]

    full_text = tokenizer.apply_chat_template(
        messages, tokenize=False,
     add_generation_prompt=False
    )
    full_ids = tokenizer(full_text,
     add_special_tokens=False, truncation=False)
     ["input_ids"]
    labels = [-100] * len(full_ids)

    prefix_messages = []
    for msg in messages:
        if msg["role"] == "assistant":
            prefix_text = tokenizer.
     apply_chat_template(
                prefix_messages, tokenize=False,
      add_generation_prompt=True
            )
            prefix_ids = tokenizer(
                prefix_text, add_special_tokens=
     False, truncation=False
            )["input_ids"]

            end_text = tokenizer.
     apply_chat_template(
                prefix_messages + [msg],
     tokenize=False, add_generation_prompt=False
            )
            end_ids = tokenizer(
                end_text, add_special_tokens=
     False, truncation=False
            )["input_ids"]

            for i in range(len(prefix_ids), min(
     len(end_ids), len(labels))):
                labels[i] = full_ids[i]

        prefix_messages.append(msg)

    full_ids = full_ids[:max_seq_length]
    labels   = labels[:max_seq_length]

    if all(l == -100 for l in labels):
        return None

    return {
        "input_ids":      full_ids,
        "attention_mask": [1] * len(full_ids),
        "labels":         labels,
    }


def build_dataset(data: List[Dict], tokenizer,
     max_seq_length: int, desc: str) -> Dataset:
    tokenised, skipped = [], 0
    for ex in data:
        try:
            tok = tokenise_example(ex, tokenizer
     , max_seq_length)
            if tok is None:
                skipped += 1
            else:
                tokenised.append(tok)
        except Exception as e:
            logger.warning(f"Skipping example: {
     e}")
            skipped += 1

    if skipped:
        logger.warning(f"{desc}: skipped {
     skipped} examples")
    logger.info(f"{desc}: {len(tokenised)}
     examples ready")
    return Dataset.from_list(tokenised)


#
#  3. MODEL SETUP
#

def load_model_and_tokenizer(args):
    logger.info(f"Loading tokenizer from {args.
     model_path}")
```

```
    tokenizer = AutoTokenizer.from_pretrained(
        args.model_path,
        trust_remote_code=True,
        padding_side="right",
    )
    if tokenizer.pad_token is None:
        tokenizer.pad_token = tokenizer.
     eos_token

    torch_dtype = torch.bfloat16 if args.bf16
     else (
        torch.float16 if args.fp16 else torch.
     float32
    )
    logger.info(f"Using dtype: {torch_dtype}")

    attn_impl = "eager"
    try:
        import flash_attn  # noqa: F401
        attn_impl = "flash_attention_2"
        logger.info("Flash Attention 2 enabled")
    except ImportError:
        logger.info("Flash Attention 2 not found
       using eager attention")

    logger.info(f"Loading model from {args.
     model_path}")
    model = AutoModelForCausalLM.from_pretrained
     (
        args.model_path,
        torch_dtype=torch_dtype,
        device_map="auto",
        trust_remote_code=True,
        use_cache=False,
        attn_implementation=attn_impl,
    )

    if args.gradient_checkpointing:
        model.gradient_checkpointing_enable(
            gradient_checkpointing_kwargs={"
     use_reentrant": False}
        )
        logger.info("Gradient checkpointing
     enabled")

    lora_config = LoraConfig(
        r=args.lora_r,
        lora_alpha=args.lora_alpha,
        lora_dropout=args.lora_dropout,
        target_modules=args.lora_target_modules,
        task_type=TaskType.CAUSAL_LM,
        bias="none",
    )
    model = get_peft_model(model, lora_config)
    model.print_trainable_parameters()

    return model, tokenizer


#
#  4. CUSTOM TRAINER
#

class RegexDFATrainer(Trainer):
    def compute_loss(self, model, inputs,
     return_outputs=False, **kwargs):
        outputs = model(**inputs)
        loss = outputs.loss

        if self.state.global_step % 50 == 0 and
     self.state.global_step > 0:
            total_norm = sum(
                p.grad.data.norm(2).item() ** 2
                for p in model.parameters() if p.
     grad is not None
            ) ** 0.5
            self.log({"grad_norm": total_norm})

        return (loss, outputs) if return_outputs
      else loss

    def evaluate(self, *args, **kwargs):
        output = super().evaluate(*args, **
     kwargs)
        if "eval_loss" in output:
            try:
                output["eval_perplexity"] = math.
     exp(output["eval_loss"])
            except OverflowError:
                output["eval_perplexity"] =
     float("inf")
        return output


#
#  5. MAIN
#

def main():
    args = parse_args()
    set_seed(args.seed)

    logger.info("=" * 60)
    logger.info("  Qwen 2.5 LoRA SFT  RegexDFA")
    logger.info(f"  Model:     {args.model_path}
     ")
    logger.info(f"  Output:    {args.output_dir}
     ")
    logger.info(f"  LoRA r={args.lora_r}, alpha
     ={args.lora_alpha}")
    logger.info(f"  LR={args.learning_rate},
     epochs={args.num_train_epochs}")
    if args.resume_from_checkpoint:
        logger.info(f"  Resuming:  {args.
     resume_from_checkpoint}")
    logger.info("=" * 60)

    os.makedirs(args.output_dir, exist_ok=True)

    model, tokenizer = load_model_and_tokenizer(
     args)

    train_raw = load_jsonl(args.train_file)
    val_raw   = load_jsonl(args.val_file)
    logger.info(f"Raw data  train: {len(
     train_raw)}, val: {len(val_raw)}")

    if args.replay_file and os.path.exists(args.
     replay_file):
        replay_raw = load_jsonl(args.replay_file
     )
        n_replay   = int(len(train_raw) * args.
     replay_ratio)
        rng = random.Random(args.seed)
        train_raw = train_raw + rng.choices(
     replay_raw, k=n_replay)
        rng.shuffle(train_raw)
        logger.info(f"Replay: added {n_replay}
     examples ({args.replay_ratio*100:.0f}%)")

    train_dataset = build_dataset(train_raw,
     tokenizer, args.max_seq_length, "Train")
    val_dataset   = build_dataset(val_raw,
     tokenizer, args.max_seq_length, "Val")

    lengths = [len(ex["input_ids"]) for ex in
     train_dataset]
    logger.info(
        f"Seq lengths  min:{min(lengths)}, max:{
     max(lengths)}, "
        f"mean:{sum(lengths)/len(lengths):.0f},
     "
        f"p95:{sorted(lengths)[int(0.95*len(
     lengths))]}"
    )

    data_collator = DataCollatorForSeq2Seq(
        tokenizer=tokenizer,
        model=model,
        label_pad_token_id=-100,
        pad_to_multiple_of=8,
    )

    training_args = TrainingArguments(
        output_dir=args.output_dir,
        run_name=args.run_name,

        per_device_train_batch_size=args.
     per_device_train_batch_size,
        per_device_eval_batch_size=args.
     per_device_eval_batch_size,
```

```
        gradient_accumulation_steps=args.
 gradient_accumulation_steps,

        num_train_epochs=args.num_train_epochs,
        learning_rate=args.learning_rate,
        weight_decay=args.weight_decay,
        warmup_ratio=args.warmup_ratio,
        lr_scheduler_type=args.lr_scheduler_type
 ,
        optim="adamw_torch_fused",

        bf16=args.bf16,
        fp16=args.fp16,
        tf32=True,

        gradient_checkpointing=args.
 gradient_checkpointing,
        gradient_checkpointing_kwargs={"
 use_reentrant": False},
        dataloader_num_workers=4,
        dataloader_pin_memory=True,

        eval_strategy="steps",
        eval_steps=args.eval_steps,
        save_strategy="steps",
        save_steps=args.save_steps,
        save_total_limit=args.save_total_limit,
        load_best_model_at_end=args.
 load_best_model_at_end,
        metric_for_best_model="eval_loss",
        greater_is_better=False,

        logging_dir=os.path.join(args.output_dir
 , "logs"),
        logging_steps=args.logging_steps,
        report_to=args.report_to if args.
 report_to != "none" else [],

        deepspeed=args.deepspeed,
        seed=args.seed,
        remove_unused_columns=False,
        prediction_loss_only=True,
    )

    callbacks = []
    if args.early_stopping_patience > 0:
        callbacks.append(EarlyStoppingCallback(
            early_stopping_patience=args.
 early_stopping_patience
        ))

    trainer = RegexDFATrainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=val_dataset,
        processing_class=tokenizer,
        data_collator=data_collator,
        callbacks=callbacks,
    )

    n_gpus   = max(torch.cuda.device_count(), 1)
    eff_bs   = args.per_device_train_batch_size
     * args.gradient_accumulation_steps * n_gpus
    n_steps  = math.ceil(len(train_dataset) /
     eff_bs * args.num_train_epochs)
    logger.info(f"Effective batch size: {eff_bs}
      | Total steps: {n_steps}")

    logger.info("Starting training...")
    # CHANGE 2 OF 2: pass resume path to trainer
     .train()
    # When None (fresh run) this is silently
     ignored by HF Trainer
    result = trainer.train(
     resume_from_checkpoint=args.
     resume_from_checkpoint)

    logger.info("Saving adapter...")
    trainer.save_model(args.output_dir)
    tokenizer.save_pretrained(args.output_dir)

    metrics = result.metrics
    metrics["train_samples"] = len(train_dataset
     )
    trainer.log_metrics("train", metrics)
    trainer.save_metrics("train", metrics)
    trainer.save_state()

    eval_metrics = trainer.evaluate()
    eval_metrics["eval_samples"] = len(
     val_dataset)
    trainer.log_metrics("eval", eval_metrics)
    trainer.save_metrics("eval", eval_metrics)

    logger.info("=" * 60)
    logger.info("Training complete!")
    logger.info(f"  eval_loss:       {
     eval_metrics.get('eval_loss', 'N/A'):.4f}")
    logger.info(f"  eval_perplexity: {
     eval_metrics.get('eval_perplexity', 'N/A')
     :.2f}")
    logger.info(f"  adapter saved:   {args.
     output_dir}")
    logger.info("=" * 60)


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

Curriculum SLURM Scripts.

submit_curriculum_phasewise.sh

```
#!/bin/bash
# submit_curriculum_phasewise.sh
#
# Usage:
#   sbatch submit_curriculum_phasewise.sh \
#     <model_size> <ordering> <order_name> [
     queue_file] [queue_index]
#
#SBATCH --job-name=curr_pw
#SBATCH --partition=gpu
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --gres=gpu:H100:1
#SBATCH --mem=80G
#SBATCH --time=2-00:00:00
#SBATCH --output=logs/curriculum_phasewise_%j.
     out
#SBATCH --error=logs/curriculum_phasewise_%j.err
#SBATCH --mail-type=END,FAIL

set -eo pipefail
echo "Job ID: $SLURM_JOB_ID | Node:
     $SLURM_NODELIST | Started: $(date)"

# -- Arguments ------------------
MODEL_SIZE="${1:-1.5b}"
ORDERING="${2:-1,2,3,4,5}"
ORDER_NAME="${3:-natural}"
QUEUE_FILE="${4:-}"
QUEUE_INDEX="${5:-1}"

echo "Model:    $MODEL_SIZE"
echo "Ordering: $ORDERING"
echo "Name:     $ORDER_NAME"

cd $SLURM_SUBMIT_DIR
source venv/bin/activate
export TOKENIZERS_PARALLELISM=false
mkdir -p logs results

# -- Model path ----------------
if [ "$MODEL_SIZE" = "1.5b" ]; then
    BASE_MODEL="Qwen/Qwen 2.5-1.5B-Instruct"
    BATCH_SIZE=4
    GRAD_ACCUM=8
    MAX_MEM="48G"
elif [ "$MODEL_SIZE" = "7b" ]; then
    BASE_MODEL="Qwen/Qwen 2.5-7B-Instruct"
    BATCH_SIZE=2
    GRAD_ACCUM=16
    MAX_MEM="80G"
else
    echo "Unknown model size: $MODEL_SIZE"; exit
      1
fi
```

```
OUTPUT_BASE="checkpoints/curriculum_${MODEL_SIZE
     }_${ORDER_NAME}"
mkdir -p "$OUTPUT_BASE"

TEST_FILE="data/regex_dfa_dataset_test.jsonl"

IFS=',' read -ra PHASES <<< "$ORDERING"

echo ""
echo "Phase ordering: ${PHASES[*]}"
echo ""

# -- Training function -------------
train_phase() {
    local phase=$1
    local phase_num=$2
    local model_input=$3
    local output_dir=$4

    echo ""
    echo "------------------------"
    echo "  Training position $phase_num: Phase
     $phase data"
    echo "  Input:  $model_input"
    echo "  Output: $output_dir"
    echo "------------------------"

    mkdir -p "$output_dir"

    python train_lora.py \
        --model_path   "$model_input" \
        --train_file   "data/phase${phase}_train.
     jsonl" \
        --val_file     "data/phase${phase}_val.
     jsonl" \
        --output_dir   "$output_dir" \
        --lora_r                        32 \
        --lora_alpha                    64 \
        --lora_dropout                  0.05 \
        --num_train_epochs              3 \
        --per_device_train_batch_size
     $BATCH_SIZE \
        --gradient_accumulation_steps
     $GRAD_ACCUM \
        --learning_rate                 2e-4 \
        --weight_decay                  0.01 \
        --warmup_ratio                  0.05 \
        --lr_scheduler_type             cosine \
        --max_seq_length                4096 \
        --bf16 \
        --gradient_checkpointing \
        --save_steps                    200 \
        --eval_steps                    200 \
        --logging_steps                 10 \
        --save_total_limit              2 \
        --early_stopping_patience       5 \
        --report_to   none \
        --run_name    "curriculum_${MODEL_SIZE}
     _${ORDER_NAME}_pos${phase_num}_phase${phase}
     "

    echo "  Training position $phase_num (phase
     $phase) done: $(date)"
}

# -- Evaluation function -------------------
evaluate_after_phase() {
    local phase=$1
    local phase_num=$2
    local adapter_dir=$3
    local out_file=$4

    echo ""
    echo "  Evaluating after position $phase_num
      (phase $phase data)..."

    python evaluate.py \
        --base_model     "$BASE_MODEL" \
        --adapter_dir    "$adapter_dir" \
        --test_file      "$TEST_FILE" \
        --output_file    "$out_file" \
        --n_samples      200 \
        --max_new_tokens 3500 \
        --temperature    0.0
```

```
    python3 -c "
import json
with open('$out_file') as f:
    d = json.load(f)
a = d['aggregate']
t = a.get('by_tier', {})
print(
    f'  Position $phase_num (Phase $phase): '
    f'overall={a[\"exact_equiv_rate\"]:.3f}  '
    f't1={t.get(\"1\",{}).get(\"exact_equiv\",0)
     :.3f}  '
    f't2={t.get(\"2\",{}).get(\"exact_equiv\",0)
     :.3f}  '
    f't3={t.get(\"3\",{}).get(\"exact_equiv\",0)
     :.3f}  '
    f't4={t.get(\"4\",{}).get(\"exact_equiv\",0)
     :.3f}'
)
"
    echo "  Eval done: $(date)"
}

# ====================
#  MAIN LOOP
# ====================

CURRENT_MODEL="$BASE_MODEL"

for i in "${!PHASES[@]}"; do
    phase_num=$((i + 1))
    phase="${PHASES[$i]}"

    PHASE_OUTPUT="$OUTPUT_BASE/pos${phase_num}
     _phase${phase}"
    EVAL_OUTPUT="results/curriculum_${MODEL_SIZE
     }_${ORDER_NAME}_pos${phase_num}_phase${
     phase}.json"

    train_phase "$phase" "$phase_num" "
     $CURRENT_MODEL" "$PHASE_OUTPUT"
    evaluate_after_phase "$phase" "$phase_num" "
     $PHASE_OUTPUT" "$EVAL_OUTPUT"

    CURRENT_MODEL="$PHASE_OUTPUT"
done

# ====================
#  FINAL SUMMARY TABLE
# ====================

echo ""
echo "============================"
echo "  CURRICULUM RESULTS: ${MODEL_SIZE} |
     Order: ${ORDER_NAME} | ${ORDERING}"
echo "============================"
echo ""

python3 -c "
import json, os

model_size  = '$MODEL_SIZE'
order_name  = '$ORDER_NAME'
ordering    = '$ORDERING'.split(',')

print(f'Order: {ordering}')
print()
print(
    f'{\"Pos\":>4}  {\"Phase\":>5}  {\"Overall
     \":>8}  '
    f'{\"Tier1\":>6}  {\"Tier2\":>6}  {\"Tier3
     \":>6}  {\"Tier4\":>6}'
)
print('-' * 55)

for i, phase in enumerate(ordering):
    pos = i + 1
    fname = (
        f'results/curriculum_{model_size}_{
     order_name}'
        f'_pos{pos}_phase{phase}.json'
    )
    try:
        with open(fname) as f:
            d = json.load(f)
        a = d['aggregate']
        t = a.get('by_tier', {})
```

```
        print(
            f'{pos:>4}  {phase:>5}  '
            f'{a[\"exact_equiv_rate\"]:>8.3f}  '
            f'{t.get(\"1\",{}).get(\"exact_equiv
    \",0):>6.3f}  '
            f'{t.get(\"2\",{}).get(\"exact_equiv
    \",0):>6.3f}  '
            f'{t.get(\"3\",{}).get(\"exact_equiv
    \",0):>6.3f}  '
            f'{t.get(\"4\",{}).get(\"exact_equiv
    \",0):>6.3f}'
        )
    except FileNotFoundError:
        print(f'{pos:>4}  {phase:>5}  (not found
    )')

print('-' * 55)
"

echo ""
echo "Finished: $(date)"

# =====================
#  SELF-CHAIN: submit next job from queue
# =====================

if [ -n "$QUEUE_FILE" ] && [ -f "$QUEUE_FILE" ];
     then
    TOTAL=$(wc -l < "$QUEUE_FILE")
    NEXT_INDEX=$((QUEUE_INDEX + 1))

    if [ "$NEXT_INDEX" -le "$TOTAL" ]; then
        NEXT=$(sed -n "${NEXT_INDEX}p" "
    $QUEUE_FILE")
        NEXT_MODEL=$(echo "$NEXT" | cut -d'|' -
    f1)
        NEXT_NAME=$(echo "$NEXT"  | cut -d'|' -
    f2)
        NEXT_ORDER=$(echo "$NEXT" | cut -d'|' -
    f3)

        echo ""
        echo "Submitting next job ($NEXT_INDEX/
    $TOTAL): $NEXT_MODEL $NEXT_NAME (
    $NEXT_ORDER)"

        sbatch \
            --job-name="curr_${NEXT_MODEL}_${
    NEXT_NAME}" \
            --output="logs/curriculum_${
    NEXT_MODEL}_${NEXT_NAME}_%j.out" \
            --error="logs/curriculum_${
    NEXT_MODEL}_${NEXT_NAME}_%j.err" \
            submit_curriculum_phasewise.sh \
                "$NEXT_MODEL" "$NEXT_ORDER" "
    $NEXT_NAME" \
                "$QUEUE_FILE" "$NEXT_INDEX"

        echo "Next job submitted. $(( TOTAL -
    NEXT_INDEX )) remaining."
    else
        echo ""
        echo "All $TOTAL curriculum jobs
    completed."
    fi
fi
```

launch_all_curriculum.sh

```
#!/bin/bash
# launch_all_curriculum.sh

MODEL_TARGET="${1:-both}"

declare -A ORDERINGS
ORDERINGS["natural"]="1,2,3,4,5"
ORDERINGS["reverse"]="5,4,3,2,1"
ORDERINGS["hard_first"]="5,1,2,3,4"
ORDERINGS["easy_first"]="1,5,2,4,3"
ORDERINGS["mid_out"]="3,2,4,1,5"
ORDERINGS["random"]="2,5,1,4,3"

ORDER_NAMES=(natural reverse hard_first
    easy_first mid_out random)

echo "============================"
echo "  Curriculum Phase-wise Experiment
    Launcher"
echo "  Target models: $MODEL_TARGET"
echo "  Strategy: 2 parallel chains (1.5B chain
    + 7B chain)"
echo "  Orderings:"
for name in "${ORDER_NAMES[@]}"; do
    echo "    $name: ${ORDERINGS[$name]}"
done
echo "============================"
echo ""

mkdir -p logs results

submit_first_in_chain() {
    local queue_file=$1
    local index=1

    local FIRST=$(head -1 "$queue_file")
    local MODEL=$(echo "$FIRST" | cut -d'|' -f1)
    local NAME=$(echo "$FIRST"  | cut -d'|' -f2)
    local ORDER=$(echo "$FIRST" | cut -d'|' -f3)
    local TOTAL=$(wc -l < "$queue_file")

    local JOB_ID=$(sbatch \
        --job-name="curr_${MODEL}_${NAME}" \
        --output="logs/curriculum_${MODEL}_${
    NAME}_%j.out" \
        --error="logs/curriculum_${MODEL}_${NAME
    }_%j.err" \
        submit_curriculum_phasewise.sh \
            "$MODEL" "$ORDER" "$NAME" "
    $queue_file" "$index" \
        | awk '{print $NF}')

    echo "  Submitted job 1/$TOTAL: $MODEL $NAME
      ($ORDER) -> Job $JOB_ID"
    echo "  Queue: $queue_file"
}

if [ "$MODEL_TARGET" = "1.5b" ] || [ "
    $MODEL_TARGET" = "both" ]; then
    QUEUE_15B="logs/curriculum_queue_1.5b.txt"
    > "$QUEUE_15B"
    for name in "${ORDER_NAMES[@]}"; do
        echo "1.5b|$name|${ORDERINGS[$name]}" >>
      "$QUEUE_15B"
    done
    echo "-- 1.5B queue (6 jobs, sequential
     chain) ---------------"
    cat "$QUEUE_15B"
    echo ""
    submit_first_in_chain "$QUEUE_15B"
    echo ""
fi

if [ "$MODEL_TARGET" = "7b" ] || [ "
    $MODEL_TARGET" = "both" ]; then
    QUEUE_7B="logs/curriculum_queue_7b.txt"
    > "$QUEUE_7B"
    for name in "${ORDER_NAMES[@]}"; do
        echo "7b|$name|${ORDERINGS[$name]}" >> "
     $QUEUE_7B"
    done
    echo "-- 7B queue (6 jobs, sequential chain)
      ------------------"
    cat "$QUEUE_7B"
    echo ""
    submit_first_in_chain "$QUEUE_7B"
    echo ""
fi

echo ""
echo "Submitted 2 jobs (one per chain)."
echo "Each chain self-submits the next job
     automatically."
echo "At any time: 2 jobs running (1.5B chain +
     7B chain)"
echo ""
echo "Monitor with:  squeue -u \$USER"
echo "Results in:    results/curriculum_*.json"
```

G.11 Mechanistic Analysis Scripts

ADFA Variant Generator. !

build_adfa_variants.py

```
"""
build_adfa_variants.py  --  Phase 1, Step 1B
============================================
PURPOSE:
  For each positive (correct) Tier 4 example,
     generate 4 variant types.
  These variants are used by filter_adfa_neurons.
     py to check whether
  a neuron fires for the RIGHT reason (true
     construction) vs surface
  reasons (format, alphabet, task difficulty).

READS:
  data/adfa/contrastive_positives.jsonl

PRODUCES:
  data/adfa/variants_v1_alphabet_swap.jsonl
  data/adfa/variants_v2_structural_sibling.jsonl
  data/adfa/variants_v3_complexity_decoy.jsonl
  data/adfa/variants_v4_step_specific.jsonl

THE 4 VARIANT TYPES:

  V1 -- Alphabet Swap:
    Same regex structure, different alphabet
     symbols.
    (a|b)*abb over {a,b}  ->  (0|1)*011 over
     {0,1}
    A TRUE construction neuron fires on BOTH the
      original and V1.

  V2 -- Structural Sibling:
    A different Tier 4 regex with same
     complexity level.
    A TRUE construction neuron fires on BOTH the
      original and V2.

  V3 -- Complexity Decoy:
    A simple regex that LOOKS complex but has an
      easy DFA (2-3 states).
    A TRUE construction neuron should NOT fire
     here.

  V4 -- Step-Specific:
    The SAME example as the original, annotated
     with step-level
    correctness labels for use in VGNS (Phase 5).


ADFA FILTER RULE (enforced later in
     filter_adfa_neurons.py):
  A neuron i is a TRUE DFA-construction neuron
     if:
    ACE_good on V1 > threshold
    ACE_good on V2 > threshold
    ACE_good on V3 < threshold
  All three conditions must hold simultaneously.

USAGE:
  python build_adfa_variants.py \
      --positives_file data/adfa/
     contrastive_positives.jsonl \
      --output_dir     data/adfa
"""

import os
import re
import json
import copy
import random
import argparse
import logging
from typing import Optional, List

logging.basicConfig(
    format="%(asctime)s | %(levelname)s | %(
     message)s",
    datefmt="%H:%M:%S", level=logging.INFO,
)
logger = logging.getLogger(__name__)
random.seed(42)


# ==========================
#  VARIANT TYPE 1 -- ALPHABET SWAP
# ==========================

ALPHABET_SWAP_MAP = {
    frozenset(["a", "b"]):       ["0", "1"],
    frozenset(["0", "1"]):       ["a", "b"],
    frozenset(["a", "b", "c"]): ["x", "y", "z"],
    frozenset(["x", "y", "z"]): ["a", "b", "c"],
    frozenset(["0", "1", "2"]): ["a", "b", "c"],
}


def swap_symbols_in_string(text: str, old_syms:
     list, new_syms: list) -> str:
    result = text
    for i, sym in enumerate(old_syms):
        result = result.replace(sym, f"__SYM{i}
     __")
    for i, sym in enumerate(new_syms):
        result = result.replace(f"__SYM{i}__",
     sym)
    return result


def build_variant1_alphabet_swap(ex: dict) ->
     Optional[dict]:
    alphabet = ex["metadata"].get("alphabet", ["
     a", "b"])
    new_alphabet = ALPHABET_SWAP_MAP.get(
     frozenset(alphabet))
    if new_alphabet is None:
        return None

    old_sorted = sorted(alphabet)
    new_sorted  = sorted(new_alphabet)

    variant = copy.deepcopy(ex)

    user_content = variant["messages"][1]["
     content"]
    user_content = swap_symbols_in_string(
     user_content, old_sorted, new_sorted)
    variant["messages"][1]["content"] =
     user_content

    variant["metadata"] = copy.deepcopy(ex["
     metadata"])
    variant["metadata"]["alphabet"]          =
     new_sorted
    variant["metadata"]["original_alphabet"] =
     old_sorted

    if variant.get("gold_dfa"):
        old_dfa = variant["gold_dfa"]
        new_trans = {}
        for state, trans in old_dfa["transitions
     "].items():
            new_row = {}
            for sym, target in trans.items():
                new_sym = (new_sorted[old_sorted.
     index(sym)]
                           if sym in old_sorted
     else sym)
                new_row[new_sym] = target
            new_trans[state] = new_row
        variant["gold_dfa"]["transitions"] =
     new_trans
        variant["gold_dfa"]["alphabet"]    =
     new_sorted

    variant["variant_type"] = "v1_alphabet_swap"
    variant["source_regex"] = ex["metadata"].get
     ("regex", "")
    variant["old_alphabet"] = old_sorted
    variant["new_alphabet"] = new_sorted
    variant["source_id"]    = ex["metadata"].get
     ("id", "")

    return variant
```

```python
# ==========================
#  VARIANT TYPE 2 -- STRUCTURAL SIBLING
# ==========================

def build_variant2_structural_sibling(ex: dict,

     all_positives: List[dict],
                                          ex_index:
     int = -1) -> Optional[dict]:
    alphabet     = ex["metadata"].get("alphabet"
     , ["a", "b"])
    source_regex = ex["metadata"].get("regex", "
     ")

    def is_different(p, p_idx):
        if p_idx == ex_index:
            return False
        if source_regex and p["metadata"].get("
     regex", "") == source_regex:
            return False
        return True

    gold_dfa    = ex.get("gold_dfa") or {}
    state_count = len(gold_dfa.get("states", [])
     )

    candidates = [
        (p, i) for i, p in enumerate(
     all_positives)
        if is_different(p, i)
        and set(p["metadata"].get("alphabet",
     [])) == set(alphabet)
        and state_count > 0
        and abs(len((p.get("gold_dfa") or {}).
     get(
            "states", [])) - state_count) <= 2
    ]

    if not candidates:
        candidates = [
            (p, i) for i, p in enumerate(
     all_positives)
            if is_different(p, i)
            and set(p["metadata"].get("alphabet"
     , [])) == set(alphabet)
        ]

    if not candidates:
        candidates = [
            (p, i) for i, p in enumerate(
     all_positives)
            if is_different(p, i)
        ]

    if not candidates:
        return None

    sibling, _ = random.choice(candidates)

    variant = copy.deepcopy(sibling)
    variant["variant_type"]  = "
     v2_structural_sibling"
    variant["source_id"]     = ex["metadata"].
     get("id", "")
    variant["source_regex"]  = ex["metadata"].
     get("regex", "")
    variant["sibling_id"]    = sibling["metadata
     "].get("id", "")
    variant["sibling_regex"] = sibling["metadata
     "].get("regex", "")

    return variant


# ==========================
#  VARIANT TYPE 3 -- COMPLEXITY DECOY
# ==========================

TIER2_REGEXES = {
    frozenset(["a", "b"]): [
        ("a*b",    "recognises strings of zero
     or more a's followed by one b"),
        ("ab*",    "recognises one a followed by
     zero or more b's"),
        ("a+b+",   "recognises one or more a's
     followed by one or more b's"),
        ("(a|b)*", "recognises any string over {
     a,b}"),
        ("a*b*",   "recognises zero or more a's
     followed by zero or more b's"),
        ("(ab)+",  "recognises one or more
     occurrences of ab"),
    ],
    frozenset(["0", "1"]): [
        ("0*1",    "recognises zero or more 0s
     followed by one 1"),
        ("01*",    "recognises one 0 followed by
     zero or more 1s"),
        ("(0|1)*", "recognises any binary string
     "),
        ("0+1+",   "recognises one or more 0s
     followed by one or more 1s"),
    ],
    frozenset(["a", "b", "c"]): [
        ("a*b*c*",   "recognises zero or more a'
     s, b's, then c's"),
        ("(a|b|c)*", "recognises any string over
     {a,b,c}"),
        ("abc+",     "recognises ab followed by
     one or more c's"),
    ],
}


def build_variant3_complexity_decoy(ex: dict) ->
      Optional[dict]:
    alphabet = ex["metadata"].get("alphabet", ["
     a", "b"])
    decoys   = TIER2_REGEXES.get(frozenset(
     alphabet))
    if not decoys:
        return None

    decoy_regex, decoy_description = random.
     choice(decoys)

    orig_regex   = ex["metadata"].get("regex", "
     ")
    user_content = ex["messages"][1]["content"]

    if orig_regex and orig_regex in user_content
     :
        new_user_content = user_content.replace(
     orig_regex, decoy_regex, 1)
    else:
        new_user_content = re.sub(
            r''[^']+'',
            f''{decoy_regex}'',
            user_content,
            count=1
        )

    variant = copy.deepcopy(ex)
    variant["messages"][1]["content"] =
     new_user_content
    variant["metadata"] = copy.deepcopy(ex["
     metadata"])
    variant["metadata"]["regex"]          =
     decoy_regex
    variant["metadata"]["tier"]           = 2
    variant["metadata"]["tier_original"]  = ex["
     metadata"].get("tier", 4)

    variant["gold_dfa"]      = None
    variant["model_output"]  = None
    variant["variant_type"]  = "
     v3_complexity_decoy"
    variant["source_id"]     = ex["metadata"].
     get("id", "")
    variant["source_regex"]  = orig_regex
    variant["decoy_regex"]   = decoy_regex

    return variant


# ==========================
#  VARIANT TYPE 4 -- STEP-SPECIFIC
# ==========================
```

```python
COT_STEP_PATTERNS = {
    "step1_language":  r"## Step 1",
    "step2_nfa_start": r"## Step 2",
    "step3_nfa_table": r"## Step 3",
    "step4_power_set": r"## Step 4",
    "step5_subset":    r"## Step 5",
    "step6_hopcroft":  r"## Step 6",
    "step7_final_dfa": r"## Step 7",
    "step8_verify":    r"## Step 8",
}


def check_step_presence(model_output: str) ->
     dict:
    return {
        step_name: bool(re.search(pattern,
     model_output, re.IGNORECASE))
        for step_name, pattern in
     COT_STEP_PATTERNS.items()
    }


def check_step3_nfa(model_output: str, alphabet:
      list) -> dict:
    step3_match = re.search(
        r'## Step 3.*?\n(.*?)(?=## Step 4|\Z)',
        model_output, re.DOTALL | re.IGNORECASE
    )
    if not step3_match:
        return {"present": False, "has_table":
     False,
                "col_count_correct": False}

    step3_text = step3_match.group(1)
    has_table  = bool(re.search(r'\|.*\|.*\|',
     step3_text))
    col_correct = False
    if has_table:
        header_match = re.search(r'\|([^\n]+)\|'
     , step3_text)
        if header_match:
            cols = [c.strip() for c in
     header_match.group(1).split('|')
                    if c.strip()]
            col_correct = len(cols) >= len(
     alphabet) + 1

    return {
        "present":           True,
        "has_table":         has_table,
        "col_count_correct": col_correct,
    }


def check_step5_subset(model_output: str) ->
     dict:
    step5_match = re.search(
        r'## Step 5.*?\n(.*?)(?=## Step 6|\Z)',
        model_output, re.DOTALL | re.IGNORECASE
    )
    if not step5_match:
        return {"present": False, "has_table":
     False}
    step5_text = step5_match.group(1)
    has_table  = bool(re.search(r'\|.*D\d+.*\|',
      step5_text))
    return {"present": True, "has_table":
     has_table}


def check_step6_hopcroft(model_output: str) ->
     dict:
    step6_match = re.search(
        r'## Step 6.*?\n(.*?)(?=## Step 7|\Z)',
        model_output, re.DOTALL | re.IGNORECASE
    )
    if not step6_match:
        return {"present": False}
    step6_text = step6_match.group(1).lower()
    has_partition = any(kw in step6_text for kw
     in
                        ["partition", "equivalen
     ", "minimis",
                         "minimiz", "class"])
    return {"present": True, "
     has_partition_language": has_partition}


def build_variant4_step_specific(ex: dict) ->
     dict:
    model_output = ex.get("model_output", "")
    alphabet     = ex["metadata"].get("alphabet"
     , ["a", "b"])

    step_annotations = {
        "steps_present":   check_step_presence(
     model_output),
        "step3_nfa":       check_step3_nfa(
     model_output, alphabet),
        "step5_subset":    check_step5_subset(
     model_output),
        "step6_hopcroft":  check_step6_hopcroft(
     model_output),
        "overall_correct": ex.get("correct",
     False),
    }

    variant = copy.deepcopy(ex)
    variant["variant_type"]     = "
     v4_step_specific"
    variant["source_id"]        = ex["metadata"].
     get("id", "")
    variant["step_annotations"] =
     step_annotations

    return variant


# ==========================
#  MAIN
# ==========================

def main():
    p = argparse.ArgumentParser()
    p.add_argument("--positives_file", required=
     True)
    p.add_argument("--output_dir",     default="
     data/adfa")
    args = p.parse_args()

    os.makedirs(args.output_dir, exist_ok=True)

    positives = []
    with open(args.positives_file) as f:
        for line in f:
            line = line.strip()
            if line:
                positives.append(json.loads(line
     ))
    logger.info(f"Loaded {len(positives)}
     positive examples")

    # Variant 1: Alphabet Swap
    v1_list = []
    for ex in positives:
        v = build_variant1_alphabet_swap(ex)
        if v:
            v1_list.append(v)

    v1_path = os.path.join(args.output_dir, "
     variants_v1_alphabet_swap.jsonl")
    with open(v1_path, "w") as f:
        for v in v1_list:
            f.write(json.dumps(v) + "\n")
    logger.info(f"V1 (alphabet swap):       {len
     (v1_list):>3} variants -> {v1_path}")

    # Variant 2: Structural Sibling
    v2_list = []
    for ex_index, ex in enumerate(positives):
        v = build_variant2_structural_sibling(ex
     , positives, ex_index)
        if v:
            v2_list.append(v)

    v2_path = os.path.join(
        args.output_dir, "
     variants_v2_structural_sibling.jsonl")
```

```
    with open(v2_path, "w") as f:
        for v in v2_list:
            f.write(json.dumps(v) + "\n")
    logger.info(
        f"V2 (structural sibling):  {len(v2_list
     ):>3} variants -> {v2_path}")

    # Variant 3: Complexity Decoy
    v3_list = []
    for ex in positives:
        v = build_variant3_complexity_decoy(ex)
        if v:
            v3_list.append(v)

    v3_path = os.path.join(
        args.output_dir, "
     variants_v3_complexity_decoy.jsonl")
    with open(v3_path, "w") as f:
        for v in v3_list:
            f.write(json.dumps(v) + "\n")
    logger.info(
        f"V3 (complexity decoy):    {len(v3_list
     ):>3} variants -> {v3_path}")

    # Variant 4: Step-Specific
    v4_list = []
    for ex in positives:
        v = build_variant4_step_specific(ex)
        v4_list.append(v)

    v4_path = os.path.join(
        args.output_dir, "
     variants_v4_step_specific.jsonl")
    with open(v4_path, "w") as f:
        for v in v4_list:
            f.write(json.dumps(v) + "\n")
    logger.info(
        f"V4 (step-specific):       {len(v4_list
     ):>3} variants -> {v4_path}")

    # Step annotation summary
    step3_present = sum(
        1 for v in v4_list
        if v["step_annotations"]["step3_nfa"]["
     present"])
    step5_present = sum(
        1 for v in v4_list
        if v["step_annotations"]["step5_subset"
     ]["present"])
    step6_present = sum(
        1 for v in v4_list
        if v["step_annotations"]["step6_hopcroft
     "]["present"])
    all_steps = sum(
        1 for v in v4_list
        if all(v["step_annotations"]["
     steps_present"].values()))

    logger.info(f"\nStep presence in correct
     Tier 4 model outputs:")
    logger.info(f"  Step 3 (NFA table):      {
     step3_present}/{len(v4_list)}")
    logger.info(f"  Step 5 (subset constr.): {
     step5_present}/{len(v4_list)}")
    logger.info(f"  Step 6 (Hopcroft):       {
     step6_present}/{len(v4_list)}")
    logger.info(f"  All 8 steps present:     {
     all_steps}/{len(v4_list)}")

    summary = {
        "n_positives":   len(positives),
        "v1_count":      len(v1_list),
        "v2_count":      len(v2_list),
        "v3_count":      len(v3_list),
        "v4_count":      len(v4_list),
        "v1_coverage":   f"{len(v1_list)/len(
     positives)*100:.0f}%",
        "v2_coverage":   f"{len(v2_list)/len(
     positives)*100:.0f}%",
        "v3_coverage":   f"{len(v3_list)/len(
     positives)*100:.0f}%",
        "step3_present": step3_present,
        "step5_present": step5_present,
        "step6_present": step6_present,
    }
    summary_path = os.path.join(args.output_dir,
      "variants_summary.json")
    with open(summary_path, "w") as f:
        json.dump(summary, f, indent=2)

    logger.info("\n" + "=" * 55)
    logger.info("STEP 1B COMPLETE")
    logger.info(f"  V1 coverage: {summary['
     v1_coverage']} of positives")
    logger.info(f"  V2 coverage: {summary['
     v2_coverage']} of positives")
    logger.info(f"  V3 coverage: {summary['
     v3_coverage']} of positives")
    logger.info(f"  V4 coverage: 100% (same
     example + annotations)")
    logger.info("=" * 55)
    logger.info("Next step: run
     compute_ace_scores.py (Phase 2)")


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

ADS Neuron Scoring.

compute_ace_scores.py

```
"""
compute_ace_scores.py  --  Phase 2: Activation
     Difference Scoring (ADS)
==========================
Replaces CE-based ACE scoring with activation
     difference scoring,
which is standard in mechanistic
     interpretability literature
(cf. Meng et al. 2022 "Locating and Editing
     Factual Associations in GPT").

METHOD:
  For every MLP neuron i in every layer l,
     compute:

  ADS_good(l, i) = mean_activation(i | correct
     examples)
                   - mean_activation(i | incorrect
      examples)
  -> Positive = neuron fires MORE on correct DFA
      construction
  -> High ADS_good = facilitating neuron (good
     neuron)

  ADS_bad(l, i)  = mean_activation(i | incorrect
      examples)
                   - mean_activation(i | correct
     examples)
  -> Positive = neuron fires MORE on incorrect
     outputs
  -> High ADS_bad = inhibiting neuron (bad
     neuron)

  Activations are collected at the act_fn output
      (SiLU output) of
  Qwen2MLP: gate_proj -> SiLU -> * up_proj ->
     down_proj
  Shape: (batch, seq_len, intermediate_dim
     =18944)
  We take the mean over all completion token
     positions.

OUTPUT FILES:
  ace_scores.json          -- full ADS scores
     for all neurons
  good_neurons.json        -- Top-K facilitating
      neurons
  bad_neurons.json         -- Top-K inhibiting
     neurons
  mixed_neurons.json       -- Top-K neurons high
      on both
  ace_layer_summary.json   -- per-layer
     aggregated stats

USAGE:
    python compute_ace_scores.py \
        --base_model   Qwen/Qwen 2.5-7B-Instruct
```

```
      \
        --adapter_dir  checkpoints/qwen7b_lora \
        --adfa_file    data/adfa/
     contrastive_combined.jsonl \
        --output_dir   results/ace_cot_7b \
        --top_k        100 \
        --neurons_per_layer_sample 200
"""

import os
import json
import logging
import argparse
from collections import defaultdict

import torch
import numpy as np
from transformers import AutoTokenizer,
     AutoModelForCausalLM
from peft import PeftModel
from tqdm import tqdm

logging.basicConfig(
    format="%(asctime)s | %(levelname)s | %(
     message)s",
    datefmt="%H:%M:%S", level=logging.INFO,
)
logger = logging.getLogger(__name__)


# ==========================
#  ACTIVATION COLLECTION
# ==========================

def collect_mlp_activations(model, tokenizer,
     examples,
                                layer_indices,
     device):
    """
    Collect mean MLP intermediate activations (
     SiLU output) for a list
    of examples. Returns dict: layer_idx -> np.
     array
    (n_examples, intermediate_dim).

    Activations are averaged over completion
     token positions only,
    giving one vector per example per layer.
    """
    activations = {l: [] for l in layer_indices}
    handles = []

    def make_hook(layer_idx):
        def hook(module, input, output):
            activations[layer_idx].append(output.
     detach().cpu())
        return hook

    for layer_idx in layer_indices:
        handle = (
            model.model.model.layers[layer_idx]
            .mlp.act_fn.register_forward_hook(
     make_hook(layer_idx))
        )
        handles.append(handle)

    per_example_acts = {l: [] for l in
     layer_indices}

    try:
        for ex in tqdm(examples, desc="
     Collecting activations", leave=False):
            prompt = tokenizer.
     apply_chat_template(
                ex["messages"][:2],
                tokenize=False,
                add_generation_prompt=True
            )
            gold      = ex["messages"][2]["
     content"]
            full_text = prompt + gold

            inputs = tokenizer(full_text,
     return_tensors="pt").to(device)
            prompt_len = tokenizer(
                prompt, return_tensors="pt")["
     input_ids"].shape[1]

            for l in layer_indices:
                activations[l] = []

            with torch.no_grad():
                model(**inputs)

            for l in layer_indices:
                if activations[l]:
                    act = activations[l][0]
                    completion_act = act[0,
     prompt_len:, :]
                    if completion_act.shape[0] >
      0:
                        mean_act =
     completion_act.mean(dim=0)
                    else:
                        mean_act = act[0].mean(
     dim=0)
                    per_example_acts[l].append(
                        mean_act.float().numpy()
     )
    finally:
        for h in handles:
            h.remove()

    result = {}
    for l in layer_indices:
        if per_example_acts[l]:
            result[l] = np.stack(
     per_example_acts[l], axis=0)
        else:
            result[l] = np.zeros((0, 1))
    return result


# ==========================
#  GET MODEL STRUCTURE INFO
# ==========================

def get_model_info(model) -> dict:
    try:
        cfg = model.config
        n_layers         = cfg.num_hidden_layers
        hidden_dim       = cfg.hidden_size
        intermediate_dim = getattr(cfg, "
     intermediate_size", hidden_dim * 4)
        return {
            "n_layers":         n_layers,
            "hidden_dim":       hidden_dim,
            "intermediate_dim": intermediate_dim
     ,
        }
    except Exception as e:
        logger.warning(f"Could not read model
     config: {e}")
        return {
            "n_layers": 28, "hidden_dim": 3584,
     "intermediate_dim": 18944}


# ==========================
#  MAIN SCORING LOOP
# ==========================

def compute_ace_scores(args):
    # Load model
    logger.info("Loading tokenizer...")
    tokenizer = AutoTokenizer.from_pretrained(
        args.base_model, trust_remote_code=True)

    logger.info("Loading base model...")
    base = AutoModelForCausalLM.from_pretrained(
        args.base_model, torch_dtype=torch.
     bfloat16,
        device_map="auto", trust_remote_code=
     True,
    )
    logger.info(f"Loading adapter from {args.
     adapter_dir}...")
    model = PeftModel.from_pretrained(base, args.
     adapter_dir)
```

```
    model.eval()

    info             = get_model_info(model)
    n_layers         = info["n_layers"]
    intermediate_dim = info["intermediate_dim"]
    device           = next(model.parameters()).
     device
    logger.info(
        f"Model: {n_layers} layers, "
        f"intermediate_dim={intermediate_dim},
     device={device}")

    # Load ADFA contrastive data
    logger.info(f"Loading ADFA data from {args.
     adfa_file}...")
    positives, negatives = [], []
    with open(args.adfa_file) as f:
        for line in f:
            line = line.strip()
            if line:
                ex = json.loads(line)
                if ex["label"] == 1:
                    positives.append(ex)
                else:
                    negatives.append(ex)
    logger.info(
        f"Loaded {len(positives)} positives, {
     len(negatives)} negatives")

    if args.max_examples:
        positives = positives[:args.max_examples
     ]
        negatives = negatives[:args.max_examples
     ]
        logger.info(
            f"Capped to {len(positives)}
     positives, "
            f"{len(negatives)} negatives")

    if len(negatives) == 0:
        logger.error(
            "No negatives found. "
            "Use contrastive_combined.jsonl (not
      just positives).")
        raise ValueError("No negatives in ADFA
     file.")

    os.makedirs(args.output_dir, exist_ok=True)

    # Decide which layers to score
    if n_layers <= 32:
        layers_to_score = list(range(n_layers))
    else:
        layers_to_score = list(range(0, n_layers
     , 2))
    logger.info(f"Scoring {len(layers_to_score)}
      layers")

    # Collect activations
    logger.info("Collecting activations for
     CORRECT examples...")
    pos_acts = collect_mlp_activations(
        model, tokenizer, positives,
     layers_to_score, device)

    logger.info("Collecting activations for
     INCORRECT examples...")
    neg_acts = collect_mlp_activations(
        model, tokenizer, negatives,
     layers_to_score, device)

    # Compute ADS scores
    rng      = np.random.default_rng(42)
    n_sample = args.neurons_per_layer_sample

    ace_good = defaultdict(dict)
    ace_bad  = defaultdict(dict)

    logger.info(
        f"Computing ADS scores: "
        f"{len(layers_to_score)} layers x {
     n_sample} neurons sampled")

    for layer_idx in tqdm(layers_to_score, desc=
     "ADS scoring"):
        pos = pos_acts.get(layer_idx)
        neg = neg_acts.get(layer_idx)

        if (pos is None or neg is None
                or pos.shape[0] == 0 or neg.
     shape[0] == 0):
            logger.warning(
                f"Layer {layer_idx}: missing
     activations, skipping")
            continue

        mean_pos = pos.mean(axis=0)
        mean_neg = neg.mean(axis=0)
        diff     = mean_pos - mean_neg

        neuron_sample = rng.choice(
            intermediate_dim,
            size=min(n_sample, intermediate_dim)
     ,
            replace=False
        ).tolist()

        for neuron_idx in neuron_sample:
            d = float(diff[neuron_idx])
            ace_good[layer_idx][neuron_idx] =
     max(d,  0.0)
            ace_bad[layer_idx][neuron_idx]  =
     max(-d, 0.0)

    # Save full scores
    scores_path = os.path.join(args.output_dir,
     "ace_scores.json")
    scores_data = {
        "ace_good": {
            str(l): {str(n): v for n, v in
     neurons.items()}
            for l, neurons in ace_good.items()},
        "ace_bad": {
            str(l): {str(n): v for n, v in
     neurons.items()}
            for l, neurons in ace_bad.items()},
        "meta": {
            "method":            "
     activation_difference_scoring_ADS",
            "n_layers":          n_layers,
            "intermediate_dim": intermediate_dim
     ,
            "n_positives":      len(positives),
            "n_negatives":      len(negatives),
            "layers_scored":    layers_to_score,
            "neurons_per_layer": n_sample,
        }
    }
    with open(scores_path, "w") as f:
        json.dump(scores_data, f)
    logger.info(f"Saved ADS scores -> {
     scores_path}")

    # Identify top-K neurons
    logger.info(f"Identifying top-{args.top_k}
     good and bad neurons...")

    all_neurons = []
    for layer_idx, neurons in ace_good.items():
        for neuron_idx, good_score in neurons.
     items():
            bad_score = ace_bad.get(layer_idx,
     {}).get(neuron_idx, 0.0)
            all_neurons.append({
                "layer":    int(layer_idx),
                "neuron":   int(neuron_idx),
                "ace_good": good_score,
                "ace_bad":  bad_score,
                "net_good": good_score -
     bad_score,
                "net_bad":  bad_score -
     good_score,
            })

    good_neurons  = sorted(
        all_neurons, key=lambda x: x["net_good"
     ], reverse=True)[:args.top_k]
    bad_neurons   = sorted(
```

```
        all_neurons, key=lambda x: x["net_bad"],
       reverse=True)[:args.top_k]
    mixed_neurons = sorted(
        all_neurons,
        key=lambda x: x["ace_good"] + x["ace_bad
     "],
        reverse=True)[:args.top_k // 2]

    with open(os.path.join(args.output_dir, "
     good_neurons.json"), "w") as f:
        json.dump(good_neurons, f, indent=2)
    with open(os.path.join(args.output_dir, "
     bad_neurons.json"), "w") as f:
        json.dump(bad_neurons, f, indent=2)
    with open(os.path.join(args.output_dir, "
     mixed_neurons.json"), "w") as f:
        json.dump(mixed_neurons, f, indent=2)

    logger.info(
        f"Top good neuron: layer={good_neurons
     [0]['layer']}, "
        f"neuron={good_neurons[0]['neuron']}, "
        f"ads_good={good_neurons[0]['ace_good
     ']:.4f}")
    logger.info(
        f"Top bad  neuron: layer={bad_neurons
     [0]['layer']}, "
        f"neuron={bad_neurons[0]['neuron']}, "
        f"ads_bad={bad_neurons[0]['ace_bad']:.4f
     }")

    # Per-layer summary
    layer_summary = []
    for layer_idx in layers_to_score:
        neurons = ace_good.get(layer_idx, {})
        if not neurons:
            continue
        good_vals = list(neurons.values())
        bad_vals  = [
            ace_bad.get(layer_idx, {}).get(n, 0)
      for n in neurons]
        layer_summary.append({
            "layer":          layer_idx,
            "mean_ace_good":  float(np.mean(
     good_vals)),
            "max_ace_good":   float(np.max(
     good_vals)),
            "mean_ace_bad":   float(np.mean(
     bad_vals)),
            "max_ace_bad":    float(np.max(
     bad_vals)),
            "n_good_neurons": sum(1 for v in
     good_vals if v > 0.01),
            "n_bad_neurons":  sum(1 for v in
     bad_vals  if v > 0.01),
        })

    with open(
            os.path.join(args.output_dir, "
     ace_layer_summary.json"), "w"
    ) as f:
        json.dump(layer_summary, f, indent=2)
    logger.info("ADS scoring complete.")

    # Sanity check
    top_good = good_neurons[0]['ace_good'] if
     good_neurons else 0
    top_bad  = bad_neurons[0]['ace_bad']   if
     bad_neurons  else 0
    logger.info(
        f"Sanity: top ADS_good={top_good:.4f},
     top ADS_bad={top_bad:.4f}")
    if top_good < 0.001 and top_bad < 0.001:
        logger.warning(
            "All ADS scores near zero -- check
     that positives/negatives "
            "differ in model behavior.")
    else:
        logger.info(
            "Scores look healthy. Proceed with
     filter_adfa_neurons.py.")


def main():
```

```
    p = argparse.ArgumentParser()
    p.add_argument("--base_model",   required=
     True)
    p.add_argument("--adapter_dir",  required=
     True)
    p.add_argument("--adfa_file",    required=
     True)
    p.add_argument("--output_dir",   default="
     results/ace_cot_7b")
    p.add_argument("--top_k",        type=int,
     default=100)
    p.add_argument("--neurons_per_layer_sample",
      type=int, default=200)
    p.add_argument("--max_examples", type=int,
     default=None)
    args = p.parse_args()
    compute_ace_scores(args)


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

Steering Vector Extraction.

extract_steering_vectors.py

```
"""
extract_steering_vectors.py  --  Phase 3 + 4:
     Representation-Level Steering
========================
Extracts layer-wise steering vectors from the
     ADFA contrastive dataset.
Implements: Mean difference (ActAdd), Linear
     probe (CAA),
SADI sparse masking. Also extracts the cross-
     model transfer vector
(No-CoT direction -> CoT model).

Outputs:
  steering_vectors.json        -- all vectors
     for all layers, all methods
  probe_accuracy.json          -- linear probe
     accuracy by layer
  cross_model_vectors.json     -- No-CoT
     advantage vector (if both given)

Usage (single model):
    python extract_steering_vectors.py \
        --base_model  Qwen/Qwen 2.5-7B-Instruct
     \
        --adapter_dir checkpoints/cot_7b \
        --adfa_file   data/adfa/adfa_contrastive.
     jsonl \
        --output_dir  results/steering_cot_7b

Usage (cross-model transfer):
    python extract_steering_vectors.py \
        --base_model      Qwen/Qwen 2.5-7B-
     Instruct \
        --adapter_dir     checkpoints/cot_7b \
        --nocot_adapter   checkpoints/no_cot_7b
     \
        --adfa_file       data/adfa/
     adfa_contrastive.jsonl \
        --output_dir      results/
     steering_crossmodel_7b \
        --cross_model
"""

import os
import json
import logging
import argparse
from typing import Dict, List, Optional

import torch
import numpy as np
from sklearn.linear_model import
     LogisticRegression
from sklearn.model_selection import
     cross_val_score
from transformers import AutoTokenizer,
     AutoModelForCausalLM
```

```python
from peft import PeftModel
from tqdm import tqdm

logging.basicConfig(
    format="%(asctime)s | %(levelname)s | %(
     message)s",
    datefmt="%H:%M:%S", level=logging.INFO,
)
logger = logging.getLogger(__name__)


# ==========================
#  HIDDEN STATE EXTRACTOR
# ==========================

class HiddenStateCollector:
    """Registers hooks on all transformer layers
      to collect hidden states."""

    def __init__(self, model, layers: List[int])
     :
        self.model     = model
        self.layers    = layers
        self.collected = {}
        self.handles   = []

    def __enter__(self):
        self.collected = {}
        for layer_idx in self.layers:
            def make_hook(idx):
                def hook(module, input, output):
                    out = (output[0] if
     isinstance(output, tuple)
                           else output)
                    self.collected[idx] = (
                        out[0, -1, :].detach().
     cpu().float())
                return hook
            layer  = self.model.model.model.
     layers[layer_idx]
            handle = layer.register_forward_hook
     (make_hook(layer_idx))
            self.handles.append(handle)
        return self

    def __exit__(self, *args):
        for h in self.handles:
            h.remove()
        self.handles = []


def collect_hidden_states(model, tokenizer,
     examples: List[dict],
                          layers: List[int]) ->
     Dict[int, np.ndarray]:
    """
    Run inference on all examples and collect
     hidden states at each layer.
    Returns: {layer_idx: np.array of shape (
     n_examples, hidden_dim)}
    """
    all_states = {l: [] for l in layers}

    for ex in tqdm(examples, desc="Collecting
     hidden states"):
        messages = ex["messages"]
        prompt   = tokenizer.apply_chat_template
     (
            messages[:2], tokenize=False,
     add_generation_prompt=True
        )
        inputs = tokenizer(prompt,
     return_tensors="pt").to(model.device)

        with HiddenStateCollector(model, layers)
      as collector:
            with torch.no_grad():
                model(**inputs)
            for layer_idx in layers:
                if layer_idx in collector.
     collected:
                    all_states[layer_idx].append
     (
                        collector.collected[
     layer_idx].numpy()
                    )

    return {
        l: np.stack(states)
        for l, states in all_states.items() if
     len(states) > 0
    }


# ==========================
#  STEERING VECTOR EXTRACTION METHODS
# ==========================

def extract_mean_diff(h_pos: np.ndarray,
                      h_neg: np.ndarray) -> np.
     ndarray:
    """ActAdd / mean difference vector."""
    return h_pos.mean(axis=0) - h_neg.mean(axis
     =0)


def extract_probe_direction(h_pos: np.ndarray,
     h_neg: np.ndarray):
    """Linear probe -- logistic regression
     coefficient vector."""
    X = np.vstack([h_pos, h_neg])
    y = np.array([1] * len(h_pos) + [0] * len(
     h_neg))

    clf_cv = LogisticRegression(max_iter=2000, C
     =1.0)
    if len(X) >= 10:
        scores      = cross_val_score(
            clf_cv, X, y, cv=min(5, len(y) // 2)
     )
        cv_accuracy = float(scores.mean())
    else:
        cv_accuracy = 0.0

    clf = LogisticRegression(max_iter=2000, C
     =1.0)
    clf.fit(X, y)
    train_accuracy = float(clf.score(X, y))

    return clf.coef_[0], cv_accuracy,
     train_accuracy


def extract_sadi_vector(h_pos: np.ndarray, h_neg
     : np.ndarray,
                        top_percentile: float =
     90.0):
    """
    SADI binary masking: sparse vector keeping
     only top-p% dimensions.
    Returns: (sparse_vector, binary_mask)
    """
    mean_diff  = h_pos.mean(axis=0) - h_neg.mean
     (axis=0)
    abs_diff   = np.abs(mean_diff)
    threshold  = np.percentile(abs_diff,
     top_percentile)
    mask       = (abs_diff >= threshold).astype(
     float)
    sparse_vec = mean_diff * mask
    return sparse_vec, mask


# ==========================
#  MAIN EXTRACTION
# ==========================

def load_model(base_model_path: str,
     adapter_path: str):
    tokenizer = AutoTokenizer.from_pretrained(
        base_model_path, trust_remote_code=True)
    base = AutoModelForCausalLM.from_pretrained(
        base_model_path, torch_dtype=torch.
     bfloat16,
        device_map="auto", trust_remote_code=
     True,
    )
    model = PeftModel.from_pretrained(base,
     adapter_path)
    model.eval()
```

```python
    return model, tokenizer


def extract_all_vectors(args):
    os.makedirs(args.output_dir, exist_ok=True)

    # Load ADFA data
    logger.info("Loading ADFA data...")
    positives, negatives = [], []

    def _load_jsonl(path, default_label=None):
        items = []
        with open(path) as f:
            for line in f:
                line = line.strip()
                if not line:
                    continue
                ex = json.loads(line)
                if default_label is not None:
                    ex["label"] = default_label
                items.append(ex)
        return items

    if args.adfa_pos and args.adfa_neg:
        positives = _load_jsonl(args.adfa_pos,
     default_label=1)
        negatives = _load_jsonl(args.adfa_neg,
     default_label=0)
    else:
        for ex in _load_jsonl(args.adfa_file):
            if ex.get("label", 1) == 1:
                positives.append(ex)
            else:
                negatives.append(ex)
    logger.info(
        f"Loaded {len(positives)} positives, {
     len(negatives)} negatives")

    # Load CoT model
    logger.info("Loading CoT model...")
    cot_model, tokenizer = load_model(args.
     base_model, args.adapter_dir)

    n_layers   = cot_model.config.
     num_hidden_layers
    hidden_dim = cot_model.config.hidden_size
    layers     = list(range(n_layers))
    logger.info(f"Model: {n_layers} layers,
     hidden_dim={hidden_dim}")

    # Collect hidden states (CoT model)
    logger.info("Collecting hidden states from
     CoT model (positives)...")
    h_pos = collect_hidden_states(cot_model,
     tokenizer, positives, layers)

    logger.info("Collecting hidden states from
     CoT model (negatives)...")
    h_neg = collect_hidden_states(cot_model,
     tokenizer, negatives, layers)

    # Extract steering vectors at each layer
    logger.info("Extracting steering vectors..."
     )
    vectors_mean    = {}
    vectors_probe   = {}
    vectors_sadi    = {}
    masks_sadi      = {}
    probe_acc_cv    = {}
    probe_acc_train = {}

    for layer_idx in tqdm(layers, desc="Layers")
     :
        if layer_idx not in h_pos or layer_idx
     not in h_neg:
            continue
        hp = h_pos[layer_idx]
        hn = h_neg[layer_idx]

        v_mean = extract_mean_diff(hp, hn)
        vectors_mean[layer_idx] = v_mean

        v_probe, cv_acc, train_acc =
     extract_probe_direction(hp, hn)
```

```python
        vectors_probe[layer_idx]   = v_probe
        probe_acc_cv[layer_idx]    = cv_acc
        probe_acc_train[layer_idx] = train_acc

        v_sadi, mask = extract_sadi_vector(hp,
     hn, top_percentile=90.0)
        vectors_sadi[layer_idx] = v_sadi
        masks_sadi[layer_idx]   = mask

    # Find best layer
    best_layer = max(probe_acc_cv, key=
     probe_acc_cv.get)
    logger.info(
        f"Best layer (highest probe CV acc): {
     best_layer} "
        f"(acc={probe_acc_cv[best_layer]:.3f})")

    # Save steering vectors
    sv_data = {
        "meta": {
            "n_layers":                n_layers,
            "hidden_dim":
     hidden_dim,
            "n_positives":             len(
     positives),
            "n_negatives":             len(
     negatives),
            "best_layer":
     best_layer,
            "best_layer_probe_acc":
     probe_acc_cv[best_layer],
        },
        "vectors_mean":    {l: v.tolist() for l,
      v in vectors_mean.items()},
        "vectors_probe":   {l: v.tolist() for l,
      v in vectors_probe.items()},
        "vectors_sadi":    {l: v.tolist() for l,
      v in vectors_sadi.items()},
        "masks_sadi":      {l: v.tolist() for l,
      v in masks_sadi.items()},
        "probe_acc_cv":    {l: float(v) for l, v
      in probe_acc_cv.items()},
        "probe_acc_train": {
            l: float(v) for l, v in
     probe_acc_train.items()},
    }
    sv_path = os.path.join(args.output_dir, "
     steering_vectors.json")
    with open(sv_path, "w") as f:
        json.dump(sv_data, f)
    logger.info(f"Saved steering vectors -> {
     sv_path}")

    # Save probe accuracy summary
    probe_summary = [
        {"layer":          l,
         "cv_accuracy":    probe_acc_cv[l],
         "train_accuracy": probe_acc_train.get(l
     , 0.0)}
        for l in sorted(probe_acc_cv.keys())
    ]
    probe_path = os.path.join(args.output_dir, "
     probe_accuracy.json")
    with open(probe_path, "w") as f:
        json.dump(probe_summary, f, indent=2)
    logger.info(f"Saved probe accuracy by layer
     -> {probe_path}")

    # Cross-model transfer (optional)
    if args.cross_model and args.nocot_adapter:
        logger.info("=" * 60)
        logger.info("CROSS-MODEL TRANSFER:
     Extracting No-CoT advantage vector")
        logger.info("=" * 60)

        del cot_model
        torch.cuda.empty_cache()

        logger.info("Loading No-CoT model...")
        nocot_model, _ = load_model(args.
     base_model, args.nocot_adapter)

        logger.info(
            "Collecting hidden states from No-
     CoT model (positives)...")
```

```
        h_nocot_pos = collect_hidden_states(
            nocot_model, tokenizer, positives,
    layers)

        transfer_vectors = {}
        transfer_norms   = {}
        for layer_idx in layers:
            if layer_idx not in h_nocot_pos or
    layer_idx not in h_pos:
                continue
            h_nc   = h_nocot_pos[layer_idx]
            h_c    = h_pos[layer_idx]
            min_n  = min(len(h_nc), len(h_c))
            v_tf   = h_nc[:min_n].mean(axis=0) -
     h_c[:min_n].mean(axis=0)
            transfer_vectors[layer_idx] = v_tf
            transfer_norms[layer_idx]   = float(
    np.linalg.norm(v_tf))

        best_transfer_layer = max(transfer_norms
    , key=transfer_norms.get)
        logger.info(
            f"Best transfer layer: {
    best_transfer_layer} "
            f"(norm={transfer_norms[
    best_transfer_layer]:.4f})")

        transfer_data = {
            "meta": {
                "n_layers":             n_layers
    ,
                "hidden_dim":
    hidden_dim,
                "best_transfer_layer":
    best_transfer_layer,
            },
            "transfer_vectors": {
                l: v.tolist() for l, v in
    transfer_vectors.items()},
            "transfer_norms": {
                l: float(n) for l, n in
    transfer_norms.items()},
        }
        transfer_path = os.path.join(
            args.output_dir, "
    cross_model_vectors.json")
        with open(transfer_path, "w") as f:
            json.dump(transfer_data, f)
        logger.info(
            f"Saved cross-model transfer vectors
     -> {transfer_path}")

    logger.info("Steering vector extraction
     complete.")


def main():
    p = argparse.ArgumentParser()
    p.add_argument("--base_model",    required=
     True)
    p.add_argument("--adapter_dir",   required=
     True,
                   help="CoT model adapter path"
     )
    p.add_argument("--nocot_adapter", default=
     None,
                   help="No-CoT adapter path (
     for cross-model transfer)")
    p.add_argument("--adfa_pos",      default=
     None)
    p.add_argument("--adfa_neg",      default=
     None)
    p.add_argument("--adfa_file",     default=
     None,
                   help="Single JSONL with label
      field")
    p.add_argument("--output_dir",    default="
     results/steering_cot_7b")
    p.add_argument("--cross_model",   action="
     store_true")
    args = p.parse_args()

    if not args.adfa_file and not (args.adfa_pos
      and args.adfa_neg):
        p.error("Provide --adfa_pos + --adfa_neg
        OR  --adfa_file")
    if args.cross_model and not args.
     nocot_adapter:
        p.error("--cross_model requires --
     nocot_adapter")

    extract_all_vectors(args)


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

Steering Evaluation and VGNS.

run_steering_eval.py

```
"""
run_steering_eval.py  --  Phase 3/4/5: Steering
     Evaluation
=========================
Evaluates ALL steering methods on the full Tier
     4 test set:
  - Functional antagonism (good/bad neurons)
  - ActAdd (mean difference vector)
  - SADI adaptive (sparse masking)
  - Cross-model transfer (No-CoT direction)
  - VGNS (verification-guided adaptive loop)

Reads pre-computed:
  - good_neurons.json / bad_neurons.json  (from
     compute_ace_scores.py)
  - steering_vectors.json                 (from
     extract_steering_vectors.py)
  - cross_model_vectors.json              (from
     extract_steering_vectors.py)

Outputs:
  steering_results.json   -- accuracy table for
     all conditions
  vgns_results.json       -- per-example VGNS
     round breakdown

Usage:
    python run_steering_eval.py \
        --base_model      Qwen/Qwen 2.5-7B-
     Instruct \
        --adapter_dir     checkpoints/cot_7b \
        --test_file       data/
     regex_dfa_dataset_test.jsonl \
        --ace_dir         results/ace_cot_7b \
        --steering_dir    results/
     steering_cot_7b \
        --output_dir      results/
     steering_eval_7b \
        --conditions      all
"""

import os
import re
import json
import logging
import argparse
from typing import Optional, List, Dict, Tuple
from itertools import product

import torch
import numpy as np
from transformers import AutoTokenizer,
     AutoModelForCausalLM
from peft import PeftModel
from tqdm import tqdm

logging.basicConfig(
    format="%(asctime)s | %(levelname)s | %(
     message)s",
    datefmt="%H:%M:%S", level=logging.INFO,
)
logger = logging.getLogger(__name__)


# ==========================
#  DFA PARSER + VERIFIER
# ==========================
```

```python
def parse_dfa(text: str, alphabet: list) ->
     Optional[dict]:
    match = re.search(
        r'## Step 7.*?Transition function.*?\n
     \|.*?\n\|[-| ]+\n'
        r'((?:\|.*?\n)+)',
        text, re.DOTALL
    )
    if not match:
        match = re.search(
            r'\| State \|.*?Accept.*?\n\|[-| ]+\
     n((?:\|.*?\n)+)',
            text, re.DOTALL
        )
    if not match:
        return None

    table_text = match.group(1)
    rows = [r.strip() for r in table_text.strip
     ().split('\n')
            if r.strip().startswith('|')]
    transitions  = {}
    accept_states = set()
    states        = []
    sa            = sorted(alphabet)

    for row in rows:
        cells = [c.strip() for c in row.split('|
     ') if c.strip()]
        if len(cells) < len(sa) + 2:
            continue
        m = re.match(r'D(\d+)', cells[0])
        if not m:
            continue
        sid = int(m.group(1))
        states.append(sid)
        # Accept marker: Y or checkmark
        if cells[-1].strip() in ('Y', 'y'):
            accept_states.add(sid)
        transitions[sid] = {}
        for j, sym in enumerate(sa):
            tm = re.match(r'D(\d+)', cells[j +
     1])
            if tm:
                transitions[sid][sym] = int(tm.
     group(1))

    if not states:
        return None
    return {
        "states":      sorted(set(states)),
        "alphabet":    sa,
        "start":       0,
        "accept":      sorted(accept_states),
        "transitions": transitions,
    }


def dfa_accepts(dfa: dict, s: str) -> bool:
    state = dfa["start"]
    for ch in s:
        if ch not in dfa["transitions"].get(
     state, {}):
            return False
        state = dfa["transitions"][state][ch]
    return state in dfa["accept"]


def is_exact(pred: dict, gold: dict, alphabet:
     list,
             max_len: int = 6) -> bool:
    for length in range(max_len + 1):
        for chars in product(alphabet, repeat=
     length):
            s = "".join(chars)
            if dfa_accepts(pred, s) !=
     dfa_accepts(gold, s):
                return False
    return True
```

```python
def verify(pred_text: str, gold_text: str,
     alphabet: list) -> bool:
    pred_dfa = parse_dfa(pred_text, alphabet)
    gold_dfa = parse_dfa(gold_text, alphabet)
    if pred_dfa is None or gold_dfa is None:
        return False
    return is_exact(pred_dfa, gold_dfa, alphabet
     )


# ==========================
#  STEERING HOOKS
# ==========================

def make_antagonism_hooks(good_neurons: List[
     dict],
                          bad_neurons: List[dict
     ],
                          alpha_good: float =
     0.5,
                          alpha_bad: float =
     0.5):
    good_by_layer = {}
    for n in good_neurons:
        good_by_layer.setdefault(n["layer"], []).
     append(n["neuron"])

    bad_by_layer = {}
    for n in bad_neurons:
        bad_by_layer.setdefault(n["layer"], []).
     append(n["neuron"])

    all_layers = set(
        list(good_by_layer.keys()) + list(
     bad_by_layer.keys()))
    hooks = {}

    for layer_idx in all_layers:
        good_idx = good_by_layer.get(layer_idx,
     [])
        bad_idx  = bad_by_layer.get(layer_idx,
     [])

        def make_hook(g_idx, b_idx, a_good,
     a_bad):
            def hook(module, input, output):
                out = (output[0] if isinstance(
     output, tuple)
                       else output)
                out = out.clone()
                if g_idx:
                    out[:, :, g_idx] *= (1.0 +
     a_good)
                if b_idx:
                    out[:, :, b_idx] *= (1.0 -
     a_bad)
                if isinstance(output, tuple):
                    return (out,) + output[1:]
                return out
            hook._hook_type = 'neuron'
            return hook

        hooks[layer_idx] = make_hook(
            good_idx, bad_idx, alpha_good,
     alpha_bad)

    return hooks


def make_static_vector_hook(vector: np.ndarray,
                            layer_idx: int,
                            alpha: float = 1.0):
    """ActAdd / mean diff: add fixed vector to
     hidden state."""
    v_tensor = torch.tensor(vector, dtype=torch.
     float32)

    def hook(module, input, output):
        out = output[0] if isinstance(output,
     tuple) else output
        out = out.clone()
        v   = v_tensor.to(out.device).to(out.
     dtype)
        out += alpha * v.unsqueeze(0).unsqueeze
```

```
     (0)
        if isinstance(output, tuple):
            return (out,) + output[1:]
        return out
    return hook


def make_sadi_hook(vector: np.ndarray, mask: np.
     ndarray,
                   layer_idx: int, delta: float
     = 2.0):
    """
    SADI adaptive hook: scale intervention by
     input's own activation.
    h_new[:, :, mask] += delta * sign(q[:, mask
     ]) * v_sadi[mask]
    """
    v_tensor    = torch.tensor(vector, dtype=
     torch.float32)
    mask_tensor = torch.tensor(mask,   dtype=
     torch.bool)

    def hook(module, input, output):
        out  = output[0] if isinstance(output,
     tuple) else output
        out  = out.clone()
        v    = v_tensor.to(out.device).to(out.
     dtype)
        mask = mask_tensor.to(out.device)
        q    = out[:, -1, :]
        intervention = delta * q[:, mask].sign()
      * v[mask]
        out[:, :, mask] += intervention.
     unsqueeze(1)
        if isinstance(output, tuple):
            return (out,) + output[1:]
        return out
    return hook


# ==========================
#  GENERATION WITH HOOKS
# ==========================

def generate_with_hooks(model, tokenizer,
     messages,
                        hooks: Dict[int,
     callable],
                        max_new_tokens: int =
     3500) -> str:
    prompt = tokenizer.apply_chat_template(
        messages[:2], tokenize=False,
     add_generation_prompt=True
    )
    inputs  = tokenizer(prompt, return_tensors="
     pt").to(model.device)
    handles = []

    for layer_idx, hook_fn in hooks.items():
        hook_type = getattr(hook_fn, '_hook_type
     ', 'representation')
        if hook_type == 'neuron':
            layer = model.model.model.layers[
     layer_idx].mlp.act_fn
        else:
            layer = model.model.model.layers[
     layer_idx]
        handle = layer.register_forward_hook(
     hook_fn)
        handles.append(handle)

    try:
        with torch.no_grad():
            out_ids = model.generate(
                **inputs,
                max_new_tokens=max_new_tokens,
                temperature=0.0,
                do_sample=False,
                pad_token_id=tokenizer.
     eos_token_id,
                eos_token_id=tokenizer.
     eos_token_id,
            )
    finally:
        for h in handles:
            h.remove()

    return tokenizer.decode(
        out_ids[0][inputs["input_ids"].shape
     [1]:],
        skip_special_tokens=True,
    )


# ==========================
#  EVALUATION CONDITIONS
# ==========================

def evaluate_condition(model, tokenizer,
     tier4_data,
                       hooks, condition_name):
    results = []
    for ex in tqdm(tier4_data, desc=
     condition_name):
        messages  = ex["messages"]
        alphabet  = ex.get("metadata", {}).get("
     alphabet", ["a", "b"])
        gold_text = messages[2]["content"]
        pred_text = generate_with_hooks(model,
     tokenizer, messages, hooks)
        correct   = verify(pred_text, gold_text,
      alphabet)
        results.append({
            "correct": correct,
            "tier":    ex.get("metadata", {}).
     get("tier", 4),
        })

    accuracy = (sum(r["correct"] for r in
     results) / len(results)
                if results else 0)
    return accuracy, results


# ==========================
#  VGNS LOOP
# ==========================

def run_vgns(model, tokenizer, tier4_data,
             good_neurons, bad_neurons,
             v_sadi, mask_sadi, best_layer,
             alpha_neuron=0.5, delta_sadi=2.0):
    """
    Verification-Guided Neuron Steering.
    For each example: try escalating steering
     until correct or exhausted.
    """
    vgns_results = []

    for ex in tqdm(tier4_data, desc="VGNS"):
        messages  = ex["messages"]
        alphabet  = ex.get("metadata", {}).get("
     alphabet", ["a", "b"])
        gold_text = messages[2]["content"]

        result = {
            "metadata":    ex.get("metadata",
     {}),
            "rounds":      [],
            "solved":      False,
            "solved_round": None,
        }

        # Round 1: No steering
        pred    = generate_with_hooks(model,
     tokenizer, messages, {})
        correct = verify(pred, gold_text,
     alphabet)
        result["rounds"].append(
            {"round": 1, "method": "none", "
     correct": correct})
        if correct:
            result["solved"]       = True
            result["solved_round"] = 1
            vgns_results.append(result)
            continue

        # Round 2: Neuron antagonism
        antag_hooks = make_antagonism_hooks(
            good_neurons, bad_neurons,
```

```
alpha_neuron, alpha_neuron)
    pred    = generate_with_hooks(
        model, tokenizer, messages,
antag_hooks)
    correct = verify(pred, gold_text,
alphabet)
    result["rounds"].append(
        {"round": 2, "method": "antagonism",
 "correct": correct})
    if correct:
        result["solved"]       = True
        result["solved_round"] = 2
        vgns_results.append(result)
        continue

    # Round 3: SADI representation steering
    sadi_hooks = {
        best_layer: make_sadi_hook(
            v_sadi, mask_sadi, best_layer,
delta_sadi)
    }
    pred    = generate_with_hooks(
        model, tokenizer, messages,
sadi_hooks)
    correct = verify(pred, gold_text,
alphabet)
    result["rounds"].append(
        {"round": 3, "method": "sadi", "
correct": correct})
    if correct:
        result["solved"]       = True
        result["solved_round"] = 3
        vgns_results.append(result)
        continue

    # Round 4: Combined (stronger alpha)
    combined_hooks = make_antagonism_hooks(
        good_neurons, bad_neurons,
        alpha_neuron * 2.0, alpha_neuron *
2.0
    )
    combined_hooks[best_layer] =
make_sadi_hook(
        v_sadi, mask_sadi, best_layer,
delta_sadi * 2.0)
    pred    = generate_with_hooks(
        model, tokenizer, messages,
combined_hooks)
    correct = verify(pred, gold_text,
alphabet)
    result["rounds"].append(
        {"round": 4, "method": "combined", "
correct": correct})
    result["solved"]       = correct
    result["solved_round"] = 4 if correct
else None

    vgns_results.append(result)

solved_counts = {1: 0, 2: 0, 3: 0, 4: 0,
 None: 0}
for r in vgns_results:
    solved_counts[r["solved_round"]] = (
        solved_counts.get(r["solved_round"],
  0) + 1)

n       = len(vgns_results)
summary = {
    "total":         n,
    "solved_total":  sum(r["solved"] for r
 in vgns_results),
    "final_accuracy": (sum(r["solved"] for r
  in vgns_results) / n
                       if n > 0 else 0),
    "solved_round1": solved_counts.get(1,
  0),
    "solved_round2": solved_counts.get(2,
  0),
    "solved_round3": solved_counts.get(3,
  0),
    "solved_round4": solved_counts.get(4,
  0),
    "unsolvable":    solved_counts.get(None,
  0),
}
```

```
    return vgns_results, summary


# ==========================
#  MAIN
# ==========================

def main():
    p = argparse.ArgumentParser()
    p.add_argument("--base_model",   required=
     True)
    p.add_argument("--adapter_dir",  required=
     True)
    p.add_argument("--test_file",    required=
     True)
    p.add_argument("--ace_dir",      required=
     True)
    p.add_argument("--steering_dir", required=
     True)
    p.add_argument("--output_dir",   default="
     results/steering_eval")
    p.add_argument("--conditions",   default="
     all")
    p.add_argument("--alpha_neuron", type=float,
      default=0.5)
    p.add_argument("--delta_sadi",   type=float,
      default=2.0)
    p.add_argument("--alpha_static", type=float,
      default=1.0)
    args = p.parse_args()

    os.makedirs(args.output_dir, exist_ok=True)

    # Load model
    logger.info("Loading model...")
    tokenizer = AutoTokenizer.from_pretrained(
        args.base_model, trust_remote_code=True)
    base = AutoModelForCausalLM.from_pretrained(
        args.base_model, torch_dtype=torch.
     bfloat16,
        device_map="auto", trust_remote_code=
     True,
    )
    model = PeftModel.from_pretrained(base, args.
     adapter_dir)
    model.eval()

    # Load Tier 4 test data
    logger.info("Loading Tier 4 test data...")
    tier4_data = []
    with open(args.test_file) as f:
        for line in f:
            line = line.strip()
            if line:
                ex = json.loads(line)
                if ex.get("metadata", {}).get("
     tier") == 4:
                    tier4_data.append(ex)
    logger.info(f"Found {len(tier4_data)} Tier 4
      examples")

    # Load pre-computed neurons
    with open(os.path.join(args.ace_dir, "
     good_neurons.json")) as f:
        good_neurons = json.load(f)
    with open(os.path.join(args.ace_dir, "
     bad_neurons.json")) as f:
        bad_neurons = json.load(f)
    logger.info(
        f"Loaded {len(good_neurons)} good, {len(
     bad_neurons)} bad neurons")

    # Load steering vectors
    with open(
            os.path.join(args.steering_dir, "
     steering_vectors.json")
    ) as f:
        sv_data    = json.load(f)
    best_layer = sv_data["meta"]["best_layer"]
    logger.info(
        f"Best layer: {best_layer} "
        f"(probe acc={sv_data['meta']['
     best_layer_probe_acc']:.3f})")

    v_mean  = np.array(sv_data["vectors_mean"][
```

```
 str(best_layer)])
v_probe = np.array(sv_data["vectors_probe"][
 str(best_layer)])
v_sadi  = np.array(sv_data["vectors_sadi"][
 str(best_layer)])
mask    = np.array(
    sv_data["masks_sadi"][str(best_layer)],
 dtype=bool)

# Load transfer vector if available
transfer_path = os.path.join(
    args.steering_dir, "cross_model_vectors.
 json")
v_transfer          = None
best_transfer_layer = best_layer
if os.path.exists(transfer_path):
    with open(transfer_path) as f:
        transfer_data = json.load(f)
    best_transfer_layer = transfer_data["
 meta"]["best_transfer_layer"]
    v_transfer = np.array(
        transfer_data["transfer_vectors"][
 str(best_transfer_layer)])
    logger.info(
        f"Loaded cross-model transfer vector
  "
        f"(best layer={best_transfer_layer})
 ")

all_conditions = args.conditions == "all"
results_table  = {}

# Condition 1: Baseline
if all_conditions or "baseline" in args.
 conditions:
    logger.info("--- Condition: Baseline (no
  steering) ---")
    acc, _ = evaluate_condition(
        model, tokenizer, tier4_data, {}, "
 Baseline")
    results_table["baseline"] = acc
    logger.info(f"Baseline Tier 4: {acc:.3f}
 ")

# Conditions 2-5: Neuron-level
if all_conditions or "antagonism" in args.
 conditions:
    logger.info("--- Condition: Good neurons
  only ---")
    hooks = make_antagonism_hooks(
        good_neurons, [], args.alpha_neuron,
  0)
    acc, _ = evaluate_condition(
        model, tokenizer, tier4_data, hooks,
  "Good only")
    results_table["good_only"] = acc
    logger.info(f"Good only Tier 4: {acc:.3f
 }")

    logger.info("--- Condition: Bad neurons
 only ---")
    hooks = make_antagonism_hooks(
        [], bad_neurons, 0, args.
 alpha_neuron)
    acc, _ = evaluate_condition(
        model, tokenizer, tier4_data, hooks,
  "Bad only")
    results_table["bad_only"] = acc
    logger.info(f"Bad only Tier 4: {acc:.3f}
 ")

    logger.info("--- Condition: Functional
 antagonism ---")
    hooks = make_antagonism_hooks(
        good_neurons, bad_neurons,
        args.alpha_neuron, args.alpha_neuron
 )
    acc, _ = evaluate_condition(
        model, tokenizer, tier4_data, hooks,
  "Antagonism")
    results_table["antagonism"] = acc
    logger.info(f"Antagonism Tier 4: {acc:.3
 f}")

    logger.info("--- Condition: Random
 neurons (control) ---")
```

```
    n_layers         = model.config.
 num_hidden_layers
    intermediate_dim = model.config.
 intermediate_size
    rng = np.random.default_rng(42)
    random_good = [
        {"layer":  int(rng.integers(n_layers
 )),
         "neuron": int(rng.integers(
 intermediate_dim))}
        for _ in range(len(good_neurons))
    ]
    random_bad = [
        {"layer":  int(rng.integers(n_layers
 )),
         "neuron": int(rng.integers(
 intermediate_dim))}
        for _ in range(len(bad_neurons))
    ]
    hooks = make_antagonism_hooks(
        random_good, random_bad,
        args.alpha_neuron, args.alpha_neuron
 )
    acc, _ = evaluate_condition(
        model, tokenizer, tier4_data, hooks,
  "Random control")
    results_table["random_control"] = acc
    logger.info(f"Random control Tier 4: {
 acc:.3f}")

# Conditions 6-8: Representation-level
if all_conditions or "sadi" in args.
 conditions:
    logger.info("--- Condition: ActAdd (mean
  vector) ---")
    hooks = {best_layer:
 make_static_vector_hook(
        v_mean, best_layer, args.
 alpha_static)}
    acc, _ = evaluate_condition(
        model, tokenizer, tier4_data, hooks,
  "ActAdd")
    results_table["actadd_mean"] = acc
    logger.info(f"ActAdd mean Tier 4: {acc
 :.3f}")

    logger.info("--- Condition: Probe
 direction (static) ---")
    hooks = {best_layer:
 make_static_vector_hook(
        v_probe, best_layer, args.
 alpha_static)}
    acc, _ = evaluate_condition(
        model, tokenizer, tier4_data, hooks,
  "Probe dir")
    results_table["probe_direction"] = acc
    logger.info(f"Probe direction Tier 4: {
 acc:.3f}")

    logger.info("--- Condition: SADI
 adaptive ---")
    hooks = {best_layer: make_sadi_hook(
        v_sadi, mask, best_layer, args.
 delta_sadi)}
    acc, _ = evaluate_condition(
        model, tokenizer, tier4_data, hooks,
  "SADI")
    results_table["sadi_adaptive"] = acc
    logger.info(f"SADI adaptive Tier 4: {acc
 :.3f}")

# Condition 9: Cross-model transfer
if v_transfer is not None and (
        all_conditions or "transfer" in args.
 conditions):
    logger.info("--- Condition: Cross-model
 transfer ---")
    for alpha in [0.5, 1.0, 2.0, 5.0]:
        hooks = {best_transfer_layer:
 make_static_vector_hook(
            v_transfer, best_transfer_layer,
  alpha)}
        acc, _ = evaluate_condition(
            model, tokenizer, tier4_data,
 hooks,
```

```
                f"Transfer alpha={alpha}")
            results_table[f"transfer_alpha_{
 alpha}"] = acc
            logger.info(f"Transfer alpha={alpha}
  Tier 4: {acc:.3f}")

    # Condition 10: VGNS
    if all_conditions or "vgns" in args.
 conditions:
        logger.info(
            "--- Condition: VGNS (Verification-
 Guided Neuron Steering) ---")
        vgns_results, vgns_summary = run_vgns(
            model, tokenizer, tier4_data,
            good_neurons, bad_neurons,
            v_sadi, mask, best_layer,
            args.alpha_neuron, args.delta_sadi,
        )
        results_table["vgns"] = vgns_summary["
 final_accuracy"]
        logger.info(f"VGNS Tier 4: {vgns_summary
 ['final_accuracy']:.3f}")
        logger.info(
            f"  Solved round 1: {vgns_summary['
 solved_round1']}")
        logger.info(
            f"  Solved round 2: {vgns_summary['
 solved_round2']}")
        logger.info(
            f"  Solved round 3: {vgns_summary['
 solved_round3']}")
        logger.info(
            f"  Solved round 4: {vgns_summary['
 solved_round4']}")
        logger.info(
            f"  Unsolvable:     {vgns_summary['
 unsolvable']}")

        vgns_path = os.path.join(args.output_dir
 , "vgns_results.json")
        with open(vgns_path, "w") as f:
            json.dump({
                "summary":     vgns_summary,
                "per_example": vgns_results,
            }, f, indent=2)
        logger.info(f"Saved VGNS results -> {
 vgns_path}")

    # Save results table
    results_path = os.path.join(args.output_dir,
   "steering_results.json")
    if os.path.exists(results_path):
        with open(results_path) as f:
            existing = json.load(f)
        existing.update(results_table)
        results_table = existing
    with open(results_path, "w") as f:
        json.dump(results_table, f, indent=2)
    logger.info(f"\nSaved results table -> {
 results_path}")

    # Print summary
    logger.info("\n" + "=" * 55)
    logger.info("STEERING RESULTS -- TIER 4
 ACCURACY")
    logger.info("=" * 55)
    for condition, acc in sorted(
            results_table.items(), key=lambda x:
  -x[1]):
        logger.info(
            f"  {condition:<35} {acc:.3f}  ({acc
 *100:.1f}%)")
    logger.info("=" * 55)


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

G.11 Tree-of-Thought Experiment Setup

Templates. All ToT prompting templates (Intuitive, Derivative) used for frontier models are provided verbatim below.

Experiment Driver. The exact script used to run the Tree-of-Thought experiments is provided below, including GPT-5.x token-parameter compatibility, JSON enforcement and extraction, multi-branch evaluation, rate-limit handling, and complete logging of raw outputs and parsed tables.

ToT Driver Script

```
#!/usr/bin/env python3

import os, json, time, re
from datetime import datetime
from typing import Optional
from openai import OpenAI

MODEL            = "gpt-5.2"
TEMPERATURE      = 0.0
MAX_TOKENS       = 4000
PROMPT_FILES     = {
    "direct":     "prompts/
     analysis_cot_branch_direct.txt",
    "derivative": "prompts/
     analysis_cot_branch_derivative.txt"
}
DATASET_PATH     = "data/analysis_new_final.json
     "
OUTPUT_RAW_DIR   = "outputs/model/raw_tot"
OUTPUT_TABLE_DIR = "outputs/model/tables_tot"
OUTPUT_META_DIR  = "outputs/model/meta_tot"
RETRIES          = 3
RETRY_SLEEP      = 1.5
RATE_LIMIT_SLEEP = 0.3

client = OpenAI()
for d in [OUTPUT_RAW_DIR, OUTPUT_TABLE_DIR,
     OUTPUT_META_DIR]:
    os.makedirs(d, exist_ok=True)

def timestamp():
    return datetime.utcnow().strftime(

def safe_filename(s):
    return re.sub(r"[^0-9A-Za-z._-]", "_", s)
     [:200]

def load_json(path):
    with open(path, "r", encoding="utf-8") as f:
        return json.load(f)

def load_text(path):
    with open(path, "r", encoding="utf-8") as f:
        return f.read()

def save_json(path, obj):
    with open(path, "w", encoding="utf-8") as f:
        json.dump(obj, f, indent=2, ensure_ascii
     =False)

def make_json_safe(obj):
    if obj is None or isinstance(obj, (str,int,
     float,bool)):
        return obj
    if isinstance(obj, list):
        return [make_json_safe(x) for x in obj]
    if isinstance(obj, dict):
        return {k: make_json_safe(v) for k,v in
     obj.items()}
    if hasattr(obj, "__dict__"):
        return make_json_safe(vars(obj))
    return str(obj)

def try_parse_json(text):
    if not text: return None
    cleaned = re.sub(
        r"```(?:json)?", "", text,
        flags=re.IGNORECASE).strip("` \n")
    try: return json.loads(cleaned)
    except Exception: pass
    m = re.search(r"\{[\s\S]*\}", cleaned)
```

```
    if m:
        try: return json.loads(m.group(0))
        except Exception: pass
    return None

def completion_kwargs():
    if MODEL.startswith("gpt-5"):
        return {"max_completion_tokens":
     MAX_TOKENS}
    return {"max_tokens": MAX_TOKENS}

def build_messages(template, regex, alphabet):
    filled = (template
        .replace("{{REGEX}}", regex)
        .replace("{{ALPHABET}}", json.dumps(
     alphabet)))
    return [
        {"role": "system", "content":
            "You are an expert in formal
     languages. "
            "Output only the DFA JSON."},
        {"role": "user", "content": filled}
    ]

def call_model(messages):
    response = client.chat.completions.create(
        model=MODEL, messages=messages,
        temperature=TEMPERATURE, **
     completion_kwargs())
    text = response.choices[0].message.content
     or ""
    meta = make_json_safe({
        "finish_reason": response.choices[0].
     finish_reason,
        "usage": getattr(response, "usage", None
     )})
    return text, meta

def run_single_branch(entry, branch_name,
     template):
    regex_id = entry["id"]
    regex    = entry["regex"]
    alphabet = entry.get("alphabet", [])
    ts       = timestamp()
    messages = build_messages(template, regex,
     alphabet)
    raw_text, meta, parsed, attempts = "", None,
      None, 0

    while attempts < RETRIES:
        attempts += 1
        try:
            raw_text, meta = call_model(messages
     )
        except Exception as e:
            meta = {"error": str(e)}
            time.sleep(RETRY_SLEEP); continue
        parsed = try_parse_json(raw_text)
        if parsed: break
        messages.append({"role": "user",
            "content": "Output only the DFA JSON.
     "})
        time.sleep(RETRY_SLEEP)

    base = safe_filename(
        f"{regex_id}_{branch_name}_{ts}")
    save_json(
        os.path.join(OUTPUT_RAW_DIR, base+".json
     "),
        make_json_safe({"id": regex_id,
            "branch": branch_name, "regex":
     regex,
            "alphabet": alphabet,
            "raw_output": raw_text,
            "attempts": attempts, "meta": meta,
            "model": MODEL, "timestamp": ts}))
    save_json(
        os.path.join(OUTPUT_META_DIR,
            base+"_meta.json"),
        make_json_safe(meta))
    if parsed is None:
        parsed = {"error": "GENERATION_FAILED",
            "reason": "Invalid or non-JSON
     output",
            "regex_id": regex_id,
            "branch": branch_name,
            "model": MODEL, "attempts": attempts
     ,
            "meta": meta,
            "raw_output_snippet": raw_text
     [:1000]}
    save_json(
        os.path.join(OUTPUT_TABLE_DIR, base+".
     json"),
        parsed)

def main():
    dataset = load_json(DATASET_PATH)
    templates = {k: load_text(v)
        for k,v in PROMPT_FILES.items()}
    for entry in dataset:
        for branch, tmpl in templates.items():
            run_single_branch(entry, branch,
     tmpl)
            time.sleep(RATE_LIMIT_SLEEP)

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

G.12 Prompts for All Methods

Intuitive Construction Prompt.

Prompt Template: Intuitive DFA Construction

```
SYSTEM:
You are an expert in formal languages and
     finite automata.
Think step-by-step internally but DO NOT output
     any reasoning.
Output only a single valid JSON object matching
     the schema below.

USER:
Given the following regular expression and
     alphabet:
REGEX: {{REGEX}}
ALPHABET: {{ALPHABET}}

Task:
1. Internally derive a correct DFA recognizing
     exactly the
   language of the regex.
2. Do not output any intermediate steps.
3. OUTPUT ONLY a single JSON object.

Required JSON schema:
{
  "states": ["q0", "q1", ...],
  "alphabet": ["a", "b", ...],
  "start_state": "q0",
  "accept_states": ["q1", ...],
  "transitions": {
    "q0": { "a": "q1", "b": "q0" },
    "q1": { "a": "q1", "b": "q2" }
  }
}

Constraints:
- Output only the JSON object with no extra
     text.
- The DFA must be total (one transition per
     state per symbol).
- Use state names q0, q1, q2, ...
- No comments, explanations, or trailing commas.
```

Derivative-Based Construction Prompt.

Prompt Template: Derivative-Based DFA Construction

```
SYSTEM:
You are an expert in formal language theory.
Internally construct a DFA using Brzozowski
     derivatives.
Each DFA state must correspond to a derivative
     of the regex.
Do NOT output any intermediate reasoning.
```

```
USER:
Given the following regular expression and
     alphabet:
REGEX: {{REGEX}}
ALPHABET: {{ALPHABET}}

Task:
1. Internally compute the regex derivatives
     over the alphabet.
2. Build the corresponding DFA from these
     derivatives.
3. Ensure the DFA is total.
4. OUTPUT ONLY the final DFA as a JSON object.

Required JSON schema:
{
  "states": ["q0", "q1", ...],
  "alphabet": ["a", "b", ...],
  "start_state": "q0",
  "accept_states": ["q1", ...],
  "transitions": {
    "q0": { "a": "q1", "b": "q0" },
    "q1": { "a": "q1", "b": "q2" }
  }
}

Constraints:
- Output only valid JSON.
- DFA must be total.
- No explanatory text.
- Use compact state naming q0, q1, q2, ...
```

Cross-Consistency Prompt.

Prompt Template: Cross-Consistency Protocol

```
You are required to produce output in EXACTLY
     THREE sections.
If you violate the format, your entire output
     is INVALID.

SECTION HEADINGS MUST BE EXACTLY:
=== TASK 1 ===
=== TASK 2 ===
=== TASK 3 ===

Do NOT add extra commentary.
Do NOT reorder. Do NOT skip. Do NOT merge tasks.

INPUT
REGEX: {{REGEX}}
ALPHABET: {{ALPHABET}}

=== TASK 1 ===
Write 6-12 complete sentences describing the
     language.
You MUST describe:
- How strings begin and end
- Mandatory patterns
- Forbidden patterns
- Repetition structure
- Any blocks or grouping
No lists. Only sentences.

=== TASK 2 ===
Output EXACTLY:
ACCEPTED:
(10 accepted strings, one per line, covering
     ALL major
structural paths of the regex)

REJECTED:
(10 rejected strings, one per line)
No explanations.

=== TASK 3 ===
For each of the 3 strings:
aaababbb
babbabaab
ababbbaab

Output EXACTLY:
aaababbb -> ACCEPTED/REJECTED: <1-2 line
     justification>
babbabaab -> ACCEPTED/REJECTED: <1-2 line
     justification>
ababbbaab -> ACCEPTED/REJECTED: <1-2 line
     justification>
```