BEYOND PATTERN MATCHING: TRACING SYMBOLIC REASONING FAILURES IN LLMS TO THEIR MECHANISTIC ORIGIN

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) have shown strong performance across reasoning benchmarks, yet whether this reflects genuine symbolic reasoning or sophisticated pattern matching remains actively debated. We investigate this question using regular languages, the simplest class of formal languages, where operator semantics are unambiguous and correctness is exactly checkable. Our work proceeds in three stages. First, by applying a cross-consistency protocol to frontier models (GPT-5.2, Grok-4.1, Gemini-2.5) and open-weight Qwen 2.5 models (1.5B, 7B, 14B), we show that models cannot reliably interpret regular expressions: they describe constraints they then violate in generation and recognition, revealing eleven systematic failure modes that recur across all architectures regardless of scale. Second, when asked to construct deterministic finite automata, these semantic failures propagate into six classes of structural construction errors that persist regardless of construction strategy. Third, we examine if training can fix these failures. Fine-tuning on a purpose-built regex-to-DFA benchmark (22,500 training, 1,000 test examples) recovers near-perfect accuracy on simpler tiers, but a persistent gap remains at the highest complexity tier that no supervision format, curriculum ordering, or model scale closes. Because the gap survives every behavioural intervention, we look inside the model. Mechanistic analysis points to a consistent picture: the outcome is largely predictable from early-layer representations, no individual neuron or localized group implements subset construction, correcting the internal representation at any layer repairs no case beyond a matched control, and inference-time steering recovers at most 2.4 percentage points of the residual gap. Thus, on this task, symbolic reasoning failures in LLMs are structural and only partially remediable.

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 & Krakauer, 2023; Yue et al., 2024; Katz et al., 2026). 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. Answering this question requires a domain where the formal rules are unambiguous and adherence to them can be checked exactly.

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

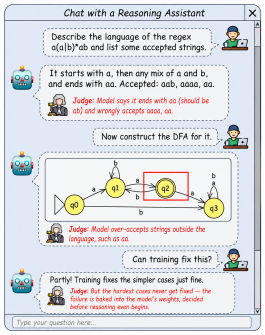


Figure 1: A reasoning assistant fails at both interpretation and construction for $a(a|b)^*ab$; a verifier checks each answer.

Regular languages (Sipser, 2013) offer a well-suited testbed for probing this distinction. Their operators have unambiguous semantics and admit definitions that eliminate interpretive ambiguity. Reasoning over regular languages requires non-trivial symbolic manipulation, including limited forms of recursion and global structural constraints. Correctness can be evaluated exactly, so we can distinguish failures of understanding (grasping what a regular expression (regex) denotes) from failures of execution (generating or recognizing strings, or transforming one representation into another). Existing benchmarks, however, do not exploit these properties. TuringQ (Zahraei & Asgari, 2024) assesses declarative knowledge, without testing whether models can apply that knowledge to concrete instances by correctly manipulating symbolic representations (Golesteanu et al., 2024). Such benchmarks leave unexamined whether models preserve invariants such as nullability, reachability, and language equivalence across transformations between regexes and deterministic finite automata.

Knowing a rule and executing it are separate abilities. A model can state the textbook definition of the Kleene star, yet fail to apply it to a specific expression. Executing the rule is itself two abilities, not one: a model must first interpret what a regex denotes, and only then construct the automaton that recognizes it. Existing evaluations of formal reasoning score only the final answer, so they cannot separate these two abilities: a model that misreads the regex looks identical to one that reads it correctly but builds the wrong machine. Figure 1 shows both failures in a single exchange: a model interprets the regex $a(a|b)^*ab$ and constructs its DFA, failing at both, and a verifier catches each error against the ground truth. We test whether models can apply regex rules, not merely state them, separating interpretation from construction failures to see which ability breaks. Our study runs in three stages, on both frontier models (GPT-5.2, Grok-4.1, Gemini-2.5) and the open-weight Qwen 2.5 family (1.5B, 7B, 14B); using both families shows whether failures are specific to one model or common across them, and Qwen's open weights let us fine-tune and probe it. *Stage 1: Can LLMs understand regular expressions?* We introduce a cross-consistency protocol that probes reasoning before any construction is attempted. Within a single prompt, the model must describe the language a regex denotes, generate accepted and rejected strings consistent with that description, and classify edge-case strings with justifications. Because all three tasks share one context, contradictions reflect the model's reasoning rather than prompt variation. This protocol surfaces eleven systematic failure modes spanning semantic interpretation, generative execution, and symbolic recognition that recur across all architectures regardless of scale (Section 3). *Stage 2: Can LLMs construct DFAs from regular expressions?* We evaluate DFA construction under two strategies, intuitive and Brzozowski-derivative, and identify six classes of structural construction errors that violate reachability and language equivalence and trace back to the interpretation failures of Stage 1. *Stage 3: Can these failures be trained away, or are they fundamental?* We test this through supervised fine-tuning, curriculum learning, and mechanistic analysis on a purpose-built regex-to-DFA benchmark (22,500 training, 1,000 test examples) spanning four tiers. Fine-tuning recovers near-perfect accuracy on simpler tiers, but a gap at the highest tier survives every supervision format, curriculum ordering, and model scale. Mechanistic analysis shows that the outcome is largely predictable from early-layer representations, no neuron or localized group implements subset construction, correcting the representation at any layer does not repair failures, and inference-time steering recovers at most 2.4 percentage points of the residual gap (Section 5).

Together these stages yield a finding new to Theory-of-Computation oriented evaluations: *on this task, symbolic reasoning failures in LLMs are structural and only partially remediable.* The weakness we isolate is algorithmic state-tracking under declarative-to-operational translation; they are encoded at the weight level, not in strategy-level decisions available at inference time, so fine-tuning repairs the simpler tiers but the hardest resists every intervention. Progress may require architectures that enforce formal invariants by construction rather than approximating them through pattern matching.

2 Related Work

Our work draws on three lines of research and connects them in a way none does alone: the classical theory of regular languages, prior evaluations of formal reasoning in LLMs, and mechanistic interpretability. First, the classical theory gives what most reasoning tasks lack: knowing exactly

when a model is right. Regex and DFA describe the same languages, standard procedures convert between them and reduce to a unique minimal form, and every step is decidable, so a model's output can be checked against verified ground truth rather than judged by plausibility; the theory also names the invariants a correct construction must preserve (nullability, reachability, language equivalence), which we test. Second, prior evaluations bracket our setting: one line asks what language classes an architecture can learn from scratch (Delétang et al., 2023), another grades what LLMs know about the theory via theory-of-computation questions (Zahraei & Asgari, 2024; Golesteanu et al., 2024); neither asks whether a pretrained model can carry out the construction a specific regex entails. Third, mechanistic interpretability locates computation inside trained models through causal tracing (Meng et al., 2022), circuit analysis (Wang et al., 2023), and activation steering (Turner et al., 2024; Wang et al., 2025), but targets factual recall, not a multi-step symbolic procedure. These two traditions — final-answer evaluation and locating where a behaviour lives — have not been joined. We do both: we force execution rather than recall on an exactly-decidable task, separate interpretation from construction so a failure traces to one or the other, and localize it inside the weights, one account from an exactly-checkable symbolic failure to its mechanistic origin. Because a model's stated reasoning often does not reflect the computation behind its answer (Turpin et al., 2023), this separation matters.

3 Can LLMs Understand Regex?

Understanding the failures requires asking: *does the breakdown occur at interpretation or at execution?* A model may misinterpret the regex (misreading nullability, compositional scope, or operator precedence) or interpret it correctly but fail to execute the construction. Conflating these modes obscures their remediation. We therefore introduce the Cross-Consistency Protocol, which probes reasoning about the regex's algebraic properties, including its informal description of the language, before DFA construction is attempted, isolating failures of understanding from failures of execution.

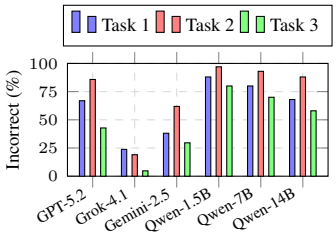


Figure 2: Whole-task error rates.

Cross-Consistency Protocol. The protocol consists of three tasks evaluated within a single prompt context (see the Cross-Consistency Protocol Example in Appendix B). **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 repetition 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 strings 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 regex. 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*". 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. This shared context does let later tasks attend to the model's own Task 1 output; we therefore verify in Appendix F that the observed dissonance is not an artifact of joint prompting. It is not: contradiction rates are as high or higher when the tasks are run in isolation, so the joint setting reports a conservative count. We apply the protocol to the 180-problem regex benchmark of Shelat et al. (2026), the same diagnostic set used for DFA construction in Section 4.

Results. If an LLM genuinely understands the language, outputs across Tasks 1–3 must be *mutually consistent*. Instead, we observe *descriptive–operational dissonance*, the recurring pattern where

a model states a constraint correctly but then violates it when generating or recognizing strings. Among frontier models, GPT-5.2 has the highest error rates (85.6% in Task 2), Grok-4.1 the lowest, and Gemini-2.5 between. Qwen 2.5 follows the same pattern at higher rates: even at 14B, errors are 68%, 88%, and 58% for Tasks 1–3. Whole-task failure peaks in Task 2, but this reflects task stringency and coverage, not worse string-level classification: Task 2 scores 20 strings under a coverage requirement, against 3 edge cases in Task 3. Normalised to per-string error, the two tasks are comparable (Appendix F). The structural signal is coverage: Task 2's dominant failure is Simple-Path Bias (Table 1), the model generating only the easy branch of the language, a failure of state-space exploration that no normalisation removes and that recurs across all architectures and scales (Figure 2). These systematic failures group into recurring modes of interpretation, generation, and recognition.

3.1 Taxonomy of Semantic Failures

Across all 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 regular languages. Table 1 reports per-model counts for each mode; the ordering is consistent across all models.

(Task 1) Semantic Interpretation Errors: (i) *Anchor Hallucination*: Models invent a prefix or suffix rule that the regex never states. For example, consider $(b \mid a(a(bb \mid bbb)^*ba)^*b)^*a(a(bb \mid bbb)^*ba)^*a(bb \mid bbb)^*bba$. Because `bbb` appears many times, the model concluded that "every string ends with at least three `b`'s in a row," but the regex forces every string to end with `bba`. The model mistook the most frequent substring for a rule, ignoring what the expression requires. (ii) *Nullability Neglect*: Models treat an optional part of the regex as if it were required. For example, for ba^* the model claimed that "the string must end with one or more *a*," but a^* means zero or more, so *b* alone is already valid. The model read 'zero or more' as 'one or more' and made an optional part mandatory. (iii) *Atomic Unit Blindness*: A regex builds strings out of fixed blocks, and every valid string is those blocks joined together. Models miss this and produce strings that use the right letters but not the right blocks. For example, for $(bb \mid bbb)^*$ the model generated *b*. Every valid string here is made of *bb* and *bbb* blocks, so its length is a sum of 2s and 3s, and a single *b* cannot be built from them. The model saw that *b* was the only allowed letter and produced it, treating the regex as a loose bag of characters rather than a set of fixed blocks. (iv) *Scope and Nesting Confusion (S & N)*: Models lose track of which part of the regex a repetition applies to. For example, in $x(ab^*c)^*y$ only (ab^*c) repeats, while *x* and *y* sit outside it and appear once. Models often pull *x* or *y* into the repeated part, reading the regex as $(xab^*c)^+$ or $(ab^*cy)^+$, and then wrongly claim the inner part must appear at least once.

(Task 2) Generative Execution and Coverage Failures: (v) *Pseudo-Structural Hallucination*: The model gets the individual letters and short patterns right, but breaks the rule for how the whole string must be shaped. For example, for $(ab)^*cd$ the model generated '*ababcdab*.' The string must end in *cd*, but the model added another *ab* after it. (vi) *Simple-Path Bias*: When the regex offers a choice, the model keeps picking the easy option and ignores the harder one. For example, in $(a^*b \mid b^*aa(aa \mid bb)^*a)ba^*$ almost every accepted string follows the simple a^*b branch, rarely trying the second branch, so its examples cover only part of the language. (vii) *Complexity Aversion*: The model rejects a string that is actually valid, simply because it looks long or irregular. For example, *baabba* is accepted by $b^*(a \mid b)a(bb)^*a$, yet the model rejected it. Instead of checking the string against the regex piece by piece, it turns down strings that do not look like clean, simple patterns.

(Task 3) Execution and Reasoning Errors: These failures operationally demonstrate the presence of reasoning errors in LLMs. (viii) *Trace Fabrication*: The model claims to find matches that are not actually in the string. For example, for *aaababbb* (counting positions from 0), the model claimed that positions 5–7 read *bba*, when they actually read *bbb*. It made up a matching step that never happened, rather than reading the string as it is. (ix) *Greedy Parsing Failures*: The model grabs the longest match at each step and never goes back to try another split, even when the greedy choice fails. For example, the regex $(a \mid ab)^*b$ accepts *abab* by splitting it as $ab \cdot a \cdot b$, leaving a final *b* for the required ending. The model instead splits it as $ab \cdot ab$, using up the whole string and leaving no *b* for the ending, so it wrongly rejects a valid string. (x) *Indexing and Positional Drift (I & P)*: 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.,

Table 1: Per-model **failure** counts. Errors are not mutually exclusive, so totals can exceed the failed count.

	Task 1					Task 2				Task 3				
Model	Anchor	Nullability	Atomic	S & N	Failed (Total)	Pseudo-Str.	Path Bias	Complexity	Failed (Total)	Trace	Greedy	I & P	Dissonance	Failed (Total)
GPT-5.2	24	70	15	48	120	58	102	29	154	19	8	21	37	77
Grok-4.1	7	21	4	13	43	12	22	5	34	2	0	1	4	9
Gemini-2.5	16	46	9	31	69	43	74	24	111	14	6	15	24	53
Qwen-1.5B	48	112	36	81	158	96	136	58	175	73	28	54	93	144
Qwen-7B	39	92	28	74	144	81	118	46	167	50	18	42	74	126
Qwen-14B	27	71	17	53	122	62	98	34	158	31	10	27	48	104

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*: What the model says in Task 1 does not match what it does in Tasks 2 and 3. For example, after stating that "the language forbids strings ending in *b*," the model went on to give several accepted examples that end in *b*. The model's description of the language and its actual handling of strings do not come from the same understanding, so the two contradict each other.

These failures are not isolated mistakes but systematic patterns: semantic errors in interpretation carry through into generation and recognition, and they recur across all six models regardless of scale, a structural limit rather than one of capacity. If models cannot reliably interpret a regex, what happens when they attempt to construct a DFA from one? Full model outputs of Section 3 are in Appendix G.

4 Failure Modes in DFA Construction

Section 3 established that models misinterpret regex semantics before construction is attempted. We now ask whether these misunderstandings propagate into the construction: *given a regex, can a model produce a correct DFA, and how do its semantic weaknesses manifest during synthesis?* Construction is the harder test: if the interpretation failures reflect a genuine absence of symbolic representation rather than isolated slips, they should surface here as deeper structural breakdowns, from incorrect state boundaries to the over-acceptance of strings that violate global constraints. We analyse DFAs produced under two construction strategies to determine not only *whether* models succeed but *where* their reasoning breaks down, using the 180-problem benchmark of Shelat et al. (2026) under two independent zero-shot prompts, one per construction strategy (Appendix D). In the first, *Intuitive Construction*, the model builds the DFA without any procedural template, as a student might in a traditional Theory of Computation course. The second, *Derivative-Based Construction*, follows Brzozowski derivatives (Brzozowski, 1964). Correctness uses exact language-equivalence verification (Appendix F), and prompts are in Appendix D.

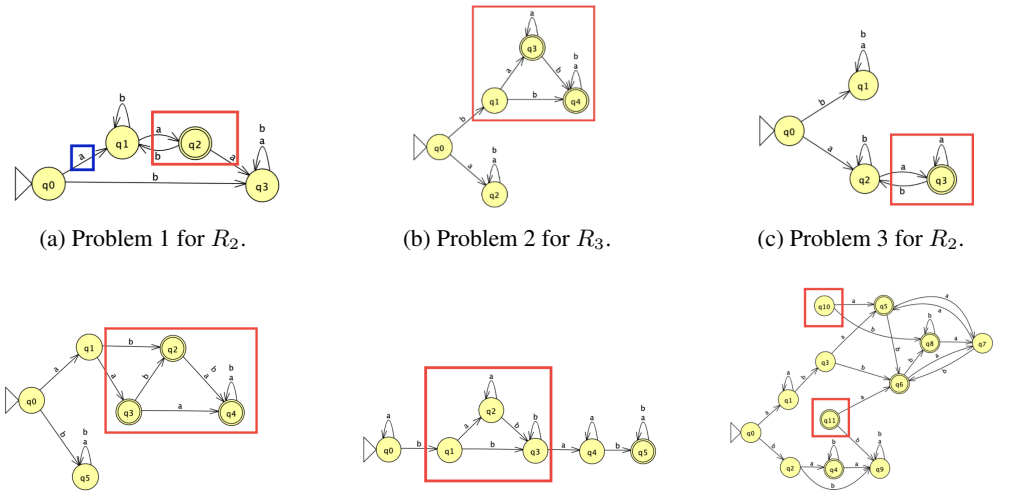
4.1 DFA Construction Errors

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.

Problem 1: Misinterpreting Kleene-Star Structure. When constructing DFA 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 3a illustrates a typical pattern. After considering the regex $R_2 : a(a \mid b)^*ab$, 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 (see Section 3).

Problem 2: Brzozowski-Derivative Errors. Under derivative-based construction, LLMs must manipulate regex symbolically. We frequently observe failures to normalise equivalent residuals or propagate nullability. This corresponds to the '*Nullability Neglect*' failure (ii) (Section 3). In Figure 3b, the model takes the derivative of $R_3 : b(a \mid b)^*a$ but commits a substitution error: it replaces the nullable ε with the universal language $(a \mid b)^*$ (marked in red). The derivative of the $(a \mid b)^*a$ with respect to a should yield $(a \mid b)^*a \cup \varepsilon$, but the model outputs $(a \mid b)^*a \cup (a \mid b)^*$, which simplifies to $(a \mid b)^*$. This overly broad residual accepts any input, erasing constraint (end with *a*).

Problem 3: Incorrect Pre-minimisation 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*



(a) Problem 1 for R_2. (b) Problem 2 for R_3. (c) Problem 3 for R_2.

(d) Problem 4 for R_4. (e) Problem 5 for R_5. (f) Problem 6 (redundant states).

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

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 3c, constitute the primary pre-minimisation errors. The structure of the initial DFA *before minimisation* directly determines the correctness of the minimised machine. When the model assigns inconsistent state semantics or incorrect transitions at this early stage, minimisation 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 3d, 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 but fail to enforce the global constraints that arise under concatenation. For example, in $R_5 : (a^*ba^+)(b^*ab)$, the final a^+ in the first block acts as a boundary: block 2 may begin only after the automaton has consumed the run of a's mandated by a^+. Prompted with the sub-expressions a^*ba^+ and b^*ab in isolation, the model constructs correct DFAs for each. But as shown in Figure 3e, during construction of R_5 it prematurely enters the b^* region of block 2 (marked in red), collapsing the required ordering. The model handles *atomic* components but fails to maintain constraints once concatenated.

Problem 6: Creation of Redundant States. Many LLM-generated DFAs contain states that contribute nothing to language recognition. Figure 3f 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 3).

Errors Across Construction Methods. Figure 4 reports failure mode frequencies; Grok-4.1 is excluded because it produced complete DFAs for only 6 of 360 construction attempts, failing to converge within the token and retry budget on the rest rather than failing on output format (Appendix D). Under intuitive construction, over-acceptance (P4) dominates, at 32% for GPT-5.2 and 47% 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 even the 14B model still failing 45% of the time, showing that scaling alone does not close the gap. Pre-minimisation errors (P3) remain stable across all scales and methods, pointing to a deep representational limitation that neither scale nor prompting strategy resolves. Each method trades one failure mode for another but never eliminates

failure: the same semantic misunderstandings identified in Section 3 propagate into construction, corrupting the DFAs models produce. Emitted chain-of-thought does not repair this either: Shelat et al. (2026) report CoT at or below zero-shot on the same 180 problems for all three frontier models (Appendix E). Since construction strategy alone cannot repair symbolic reasoning, we next ask whether training can overcome what prompting cannot (Section 5).

5 Can Training Fix These Failures?

Sections 3 and 4 showed that LLMs fail the regex-to-DFA task at both interpretation and construction, and that neither scale nor construction techniques close the gap. This leaves Stage 3's question open: *can these failures be trained away, or are they fundamental?* Answering it requires more than prompting, so we move from probing to training. The diagnostic benchmark of Sections 3-4 cannot supply the data, having too few examples for fine-tuning and no structured difficulty levels. We therefore build a new regex-to-DFA benchmark of 22,500 training and 1,000 test examples, each pairing a regex with a ground-truth DFA produced and checked by the full construction procedure, so every example comes with correct intermediate steps (Appendix C covers the dataset, tiers, and formats). The benchmark is graded into four tiers of increasing difficulty: Tier 1 (symbol mapping, 2-3 state DFAs), Tier 2 (closure operators, 2-4 states), Tier 3 (mixed operators, 4-6 states), and Tier 4 (nested closures, 6-18 states). Tier 4 is the only tier where subset construction is non-trivial, so one early error makes the rest wrong, and it shows whether a model can carry out the procedure rather than pattern-match parts of it. We evaluate across the Qwen 2.5 family (1.5B, 7B, 14B), spanning an order of magnitude in scale, with three approaches that each do more than the last: prompting (zero-shot and few-shot), supervised fine-tuning, and mechanistic intervention. To separate learning from memorization, each model is evaluated every 200 steps on 400 held-out examples, split evenly between in- and out-of-distribution. Full training details are in Appendix D.

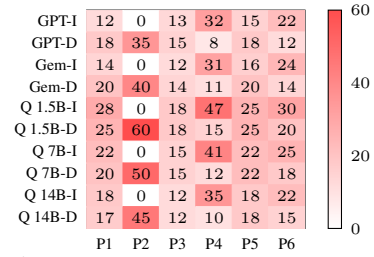


Figure 4: Failure rate (%), Intuitive (I) vs. Derivative (D) (GPT: GPT-5.2, Gem: Gemini-2.5, Q: Qwen 2.5).

5.1 Prompting and Fine-Tuning

We first try three behavioural approaches, none of which closes the Tier 4 gap. (i) *Zero-shot and Few-shot Baselines.* Zero-shot and few-shot prompting yield near-zero accuracy across all model sizes and tiers (Appendix G), so fine-tuning is necessary. (ii) *Supervised Fine-Tuning.* We fine-tune all three model sizes under two formats differing only in output: CoT, where each example includes the full construction steps before the final DFA, and No-CoT, where the model produces the DFA directly. We use LoRA (Hu et al., 2022) with base weights frozen (hyperparameters in Appendix D). Both formats reach near-perfect accuracy on Tiers 1–3 and diverge only at Tier 4 (Table 2a). 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, a ceiling that added capacity does not raise. No-CoT improves steadily to 82.0%, 85.4%, and 87.7%. This inversion of the usual CoT advantage is consistent with error accumulation over long generations: a Tier 4 CoT target spans thousands of tokens (Thompson fragments, the full NFA table, subset mappings, and Hopcroft splits; Appendix C), and under exact-match scoring a single slip anywhere in that trace invalidates the final DFA, whereas No-CoT emits only the final transition table in far fewer tokens and so exposes less surface for cascading errors (Dziri et al., 2023). We analyse this length dependence in Appendix C. This also bears on whether the Tier 4 gap is a representational limit or merely the absence of a scratchpad: were it the latter, CoT should exceed No-CoT, yet No-CoT matches or beats it (Merrill & Sabharwal, 2024), so the limitation lies in the weights, not in inference-time reasoning (Appendix E). (iii) *Curriculum Learning.* We next test whether restructuring training order helps (Bengio et al., 2009), dividing training into five phases from Tier 1 to Tier 4, with a balanced replay phase intended to prevent catastrophic forgetting, on the 7B model, our scaling midpoint. No ordering approaches joint training (Table 2b): the best (natural easy-to-hard) reaches 43.5% overall and 39.0% on Tier 4, far below joint CoT SFT on the same 7B model (96.5%, 82.9%). This gap is not a compute or data artifact but reflects catastrophic forgetting on earlier tiers and plasticity loss on Tier 4 (Appendix C). The worst (reverse order) collapses to 26.0%, and hard-first reaches only 39.0%, confirming that early exposure to hard examples hurts. The Tier 4 gap persists regardless of training order. Full curriculum details are in Appendix C.

Table 2: Mitigation results on the regex-to-DFA benchmark. (a) Fine-tuning accuracy (%) by tier; (b) curriculum ordering on the 7B model; (c) inference-time steering on the 285-example Tier 4 mechanistic subset (baseline 85.3%, separate from the 1,000-example test set in (a); Appendix F).

(a) Fine-tuning accuracy				(b) Curriculum (7B)			(c) Steering (Tier 4)			
Train	**Size**	**Overall**	**T4**	**Order**	**Overall**	**T4**	**Condition**	**Acc.**	**Δ**	**Level**
CoT	1.5B	96.0	80.5	Natural ($1 \to 4$)	43.5	39.0	Baseline (CoT 7B)	85.3	—	—
CoT	7B	96.5	82.9	Reverse ($4 \to 1$)	26.0	29.3	Good neurons ($\times 1.5$)	86.0	$+0.7$	Neuron
CoT	14B	96.3	82.1	Hard-first ($4, 1 \to 3$)	39.0	34.1	Bad neurons ($\times 0.5$)	85.6	$+0.3$	Neuron
No-CoT	1.5B	96.3	82.0	Mid-out ($3 \to 1, 4$)	43.5	39.0	Func. antagonism	86.0	$+0.7$	Neuron
No-CoT	7B	97.0	85.4	Random	37.0	31.7	Random (control)	85.6	$+0.3$	Control
No-CoT	14B	96.6	87.7	—	—	—	ActAdd	85.6	$+0.3$	Representation
—	—	—	—	—	—	—	Probe direction	84.9	-0.4	Representation
—	—	—	—	—	—	—	SADI	86.3	$+1.0$	Representation
—	—	—	—	—	—	—	VGNS (4-round) [ours]	87.7	$+2.4$	Combined

5.2 Mechanistic Analysis

Tier 4 gaps persist regardless of supervision format or training order, so we look inside the CoT-trained 7B model, whose Tier 4 accuracy stays flat across a $10\times$ scale increase. This raises a question: *why does the model still fail on Tier 4, even after seeing thousands of correct examples during training?* To answer it, we ask whether any part of the model is responsible for subset construction. Prior work has localized specific behaviours to identifiable components (Meng et al., 2022; Li et al., 2026), so we test whether subset construction is similarly localized. Take the regex $(ab)^*c$: at every step the model must track which automaton states are simultaneously active, which is exactly what subset construction does when converting a nondeterministic finite automaton into a DFA. If the procedure were localized, specific neurons or layers would fire whenever it runs, regardless of the regex's surface details; if not, the computation is distributed rather than carried by any identifiable part.

Activation Difference Scoring. We examine the activations of individual neurons across all 28 transformer layers of the CoT-trained 7B model on a held-out set of 285 Tier 4 examples reserved for mechanistic analysis (baseline accuracy 85.3%; 243 correct, 42 incorrect). For each of the 56,000 neuron–layer pairs sampled (2,000 neurons per layer from the 18,944-dimensional MLP activations), we measure whether the neuron fires more strongly when the model produces a correct DFA than when it fails (Li et al., 2026). We call this Activation Difference Scoring (ADS). Correlation with correctness, however, is not enough: a neuron might fire strongly on correct outputs simply because they tend to be longer, or use a particular alphabet. To isolate neurons that encode subset construction rather than surface properties, we test each neuron against three contrast inputs: (i) *V1*: The same DFA with the alphabet renamed (e.g., a→0, b→1). A construction neuron should still fire strongly, since the task is identical. (ii) *V2*: A different regex producing a DFA of similar structure and sizerR. Again, a construction-specific neuron should fire strongly. (iii) *V3*: A regex that *looks* complex but produces a trivial 2-state DFA, a decoy. A construction neuron should fire weakly here. A neuron truly tracking subset construction thus scores high on V1 and V2 but low on V3. We capture this with a specificity score for neuron i in layer l: $\text{Specificity}(l, i) = \frac{\text{V1}+\text{V2}}{2} - \text{V3}$, where V1, V2, and V3 are that neuron's mean activations, averaged over completion tokens, on the three contrast sets. No neuron behaves this way, firing strongly on V1/V2 but weakly on V3. Even the most selective candidate, in Layer 27, still fires at 91% of its own peak activation on the trivial-DFA decoy V3, so it does not track subset construction; this pattern holds across all 56,000 candidates. Thus no single neuron or small group carries out the procedure, and the computation is distributed across many neurons.

Where in the model is the outcome decided? We next ask, across the model's 28 layers, when the outcome becomes predictable. At each layer we train a linear classifier to predict, from the internal state at that layer alone, whether the final DFA will be correct. By Layer 10, before the model has completed half its computation, a linear probe separates eventual successes from failures with AUROC 0.85 and balanced accuracy 0.845 (raw accuracy 87.0% against an 85.3% majority-class baseline). The later layers, where DFA construction runs (layers 20–27 hold the densest concentrations of correctness-correlated neurons, peaking at Layer 27), add little: the outcome is already readable from the early representation. But readable is not caused: the probe shows the outcome can be read off the early layers, not that they determine it. To tell the difference, we intervene directly.

Activation patching: can correcting the model's internal state repair the failure? To test whether the early representation merely predicts the outcome or causes it, we use activation patching (Meng et al., 2022). For a failing example, we replace the model's internal state at a chosen layer with the state averaged over all examples it solves correctly, and then let it finish generating. If the failure were simply a bad early representation, injecting a good one should repair it. We try this at three levels of scope. Patching each layer on its own recovers no failures across layers 0 to 25, and only 4 to 5 examples at the final two layers. Patching whole ranges of layers together (for example, layers 0 to 4, or 10 to 14) shows the same pattern: zero recovery except the last range (layers 24 to 27), which recovers 5. Even patching all 28 layers at once, replacing the entire internal trajectory with the correct one, recovers none. As a control, we run the same patch using the state from incorrect examples, and it recovers as many or more (7 versus 5 at the last layer). Since injecting a wrong representation helps as much as a correct one, these late flips reflect perturbation of the near-final output, not genuine repair. Correcting the model's internal representation toward a generic, problem-agnostic mean of correct states, anywhere in the network and at any scope, does not repair the failure. Because this mean is taken across structurally heterogeneous examples, it removes instance-specific conditioning, so this result speaks to aggregate correctness directions and does not establish that an instance-accurate correct state would fail. To probe whether the failure depends on the regex's surface form, we evaluated alphabet-swapped variants (V1) of identical DFA topology for the 42 failing Tier-4 examples. Surface form has a partial effect: relabeling the alphabet converts 11 of 42 failures into successes, while the majority persist. The failure is therefore neither purely topological nor purely a matter of surface tokens; both the underlying automaton and its symbolic encoding influence Tier-4 success. A clean instance-matched patch (injecting a correct twin's representation into its failing base) requires a correct twin that is also token-aligned with the base; in our data the correct-twin and token-aligned sets do not overlap, so we do not make an instance-matched causal claim. One milder possibility remains: the representation may be only slightly off, so a gentle nudge rather than a full replacement could tip a failing run toward success. We test this next.

Can we fix failures by intervening at inference time? We now ask whether we can repair a failing run by nudging the model's internal activations, without retraining. The neuron analysis so far only asked whether a neuron *fires* like a subset-construction detector; here we test whether *acting* on the internals changes the outcome. We try interventions at two levels, individual neurons and then whole-layer representations (all results in Table 2c). *(i) Neuron-level interventions.* Amplifying the neurons most associated with correct outputs ($1.5\times$) gains $+0.7$ pp, and damping those associated with failures ($0.5\times$) gains $+0.3$ pp; doing both gives only $+0.7$ pp. Crucially, applying the same pushes to *randomly* chosen neurons gives the same $+0.3$ pp, so targeting the "important" neurons beats random by just 0.4 pp. No neuron or small group can be pushed to produce subset construction. *(ii) Representation-level steering.* Since no neuron gives leverage, we move up to the whole hidden state at Layer 10 and try three ways to steer it. ActAdd (Turner et al., 2024) adds a single fixed direction pointing from "incorrect" toward "correct" outputs; it yields only $+0.3$ pp, because the failing examples do not form one cluster. Steering instead along the direction a trained probe uses to *tell apart* correct from incorrect states *hurts* (-0.4 pp): the direction that best *predicts* whether an answer is correct is not the direction that *makes* it correct. Finally, SADI (Wang et al., 2025), which adapts the intervention per example, does best of the three at $+1.0$ pp (86.3%), still under 10% of the total gap. *(iii) Verification-Guided Neuron Steering (VGNS), our method.* We propose an iterative scheme that escalates the intervention only on examples that remain incorrect, using the verifier to select them after each round. As a baseline, resampling five times per Tier 4 example (temperature 0.8) recovers zero additional examples: the model gives a different incorrect DFA each time but never a correct one, so the solution is not recoverable within this sampling budget (Appendix G). We therefore apply progressively stronger interventions to the still-unsolved examples after each round: no intervention, then functional antagonism (amplifying good and damping bad neurons together), then SADI, then both at $2\times$ strength. This reaches **87.7%** ($+2.4$ pp), the best across all conditions. Yet 35 examples resist all four rounds, indicating that for the residual cases the correct computation is not recoverable by the inference-time interventions we test (Appendix F).

Concluding Remarks. Across three stages, LLMs misinterpret regexes in eleven modes that propagate into six classes of DFA construction failure, and fine-tuning leaves a Tier 4 gap no supervision format, curriculum, or scale closes. Mechanistically, the outcome is predictable from early layers, no neuron or group implements subset construction, patching repairs nothing, and steering recovers at most 2.4 pp: the model approximates subset construction rather than executing it. On this task, symbolic reasoning failures are structural and only partially remediable (limitations in Appendix H).

AI USE STATEMENT

In this work, we used generative AI tools to implement methods (writing, refactoring, and debugging experiment and evaluation code) and to draft and edit parts of the manuscript for readability. We have not used generative AI tools to generate synthetic datasets, develop the theoretical or conceptual framework, propose or refine hypotheses, design the research methodology or experiments, or interpret results; these originate entirely from the authors. Additionally, we used generative AI tools to format references and to suggest improvements to the structure and phrasing of the paper. We have reviewed all AI-assisted work: all AI-assisted code was verified and tested for correctness by the authors, and all AI-edited text was checked against the underlying results and claims. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies the formal reasoning of large language models using only synthetic and researcher-authored formal structures, regex and DFA, and involves no human data, personal information, or sensitive content. All fine-tuning data is machine-generated, so models are never exposed to harmful or biased material. Human involvement was limited to blinded annotation of model outputs against formal ground truth, and all mitigation experiments are fully automatic. We use parameter-efficient training to keep compute modest. A full discussion, including data provenance, evaluation fairness, and environmental impact, appears in Appendix I.

REPRODUCIBILITY STATEMENT

We release all code for dataset generation, model training, inference, construction evaluation, mechanistic analysis, and DFA correctness verification, together with prompting templates, hyperparameters, and data-processing pipelines, in an anonymized repository: `https://anonymous.4open.science/r/llm-formal-reasoning-C0FA/`. The benchmark, its four tiers, and data formats are described in Appendix C; full training and prompting details in Appendix D; and the semantic-equivalence checker used for correctness in Appendix F. All results were collected on 24 March 2026; frontier-model results (GPT-5.2, Grok-4.1, Gemini-2.5), obtained via their public APIs, may differ if those models are updated. Every dataset used or produced is synthetic or publicly available, so all artifacts can be released without restriction.

REFERENCES

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. ProofNet: Autoformalizing and formally proving undergraduate-level mathematics, 2023. arXiv preprint arXiv:2302.12433.

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

Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the Association for Computing Machinery*, 11(4):481–494, October 1964. doi: 10.1145/321239.321249. URL `https://doi.org/10.1145/321239.321249`.

François Chollet. On the measure of intelligence, 2019. arXiv preprint arXiv:1911.01547.

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. Neural networks and the chomsky hierarchy. In *Proceedings of the 11th International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=WbxHAzkeQcn`.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Sean Welleck, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D. Hwang, Soumya Sanyal, Xiang

Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. Faith and fate: Limits of transformers on compositionality. In *Advances in Neural Information Processing Systems 36*, pp. 70293–70332, 2023.

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, Natalie Stewart, Bogdan Grechuk, Tetiana Grechuk, Shreepranav Varma Enugandla, and Mark Wildon. FrontierMath: A benchmark for evaluating advanced mathematical reasoning in AI, 2024. arXiv preprint arXiv:2411.04872v2.

Matei A. Golesteanu, Garrett B. Vowinkel, and Ryan E. Dougherty. Can ChatGPT pass a theory of computing course? In *Proceedings of the 2024 on ACM Virtual Global Computing Education Conference V. 1*, SIGCSE Virtual 2024, pp. 33–38, Virtual Event, NC, USA, December 2024. Association for Computing Machinery. doi: 10.1145/3649165.3690116. URL https://dl.acm.org/doi/10.1145/3649165.3690116.

John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz (eds.), *Theory of Machines and Computations*, pp. 189–196. Academic Press, New York, 1971. doi: 10.1016/B978-0-12-417750-5.50022-1. Proceedings of an International Symposium held at Technion, Haifa, Israel, August 16–19, 1971.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

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

Michael Katz, Harsha Kokel, and Sarath Sreedharan. Seemingly simple planning problems are computationally challenging: The Countdown game. In *The 2nd Workshop on World Models: Understanding, Modelling and Scaling at ICLR 2026*, 2026. URL https://openreview.net/forum?id=c7O0NrLJrQ.

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. BIG-Bench Extra Hard. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 26473–26501, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.1285. URL https://aclanthology.org/2025.acl-long.1285/.

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. Solving quantitative reasoning problems with language models. In *Advances in Neural Information Processing Systems 35*, pp. 3843–3857. Curran Associates, Inc., 2022.

Wenjie Li, Guansong Pang, Hezhe Qiao, Debin Gao, and David Lo. Identifying good and bad neurons for task-level controllable LLMs, 2026. URL https://arxiv.org/abs/2601.04548.

Thang Luong and Edward Lockhart. Advanced version of gemini with deep think officially achieves gold-medal standard at the international mathematical olympiad. Google DeepMind Blog, July 2025. URL https://deepmind.google/blog/advanced-version-of-gemini-with-deep-think-officially-achieves-gold-medal-standard-at-the-international-mathematical-olympiad/.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. Locating and editing factual associations in GPT. In *Advances in Neural Information Processing Systems 35*, 2022.

William Merrill and Ashish Sabharwal. The expressive power of transformers with chain of thought. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. URL https://openreview.net/forum?id=NjNGlPh8Wh.

Seyed Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. GSM-Symbolic: Understanding the limitations of mathematical reasoning in large language models. In *Proceedings of the 13th International Conference on Learning Representations*, Singapore, 2025. URL https://openreview.net/forum?id=AjXkRZIvjB.

Melanie Mitchell and David C. Krakauer. The debate over understanding in AI's large language models. *Proceedings of the National Academy of Sciences*, 120(13):e2215907120, 2023. doi: 10.1073/pnas.2215907120. URL https://www.pnas.org/doi/10.1073/pnas.2215907120.

Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability. In *International Conference on Learning Representations*, 2023.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115. arXiv preprint arXiv:2412.15115.

Shlok Shelat, Jay Raval, Souvik Roy, and Manas Gaur. Beyond memorization: Testing LLM reasoning on unseen theory of computation tasks. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 29904–29934, San Diego, California, United States, July 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.findings-acl.1495. URL https://aclanthology.org/2026.findings-acl.1495/.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36*, pp. 8634–8652, 2023.

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

Alexander Matt Turner, Lisa Thiergart, Gavin Leech, David Udell, Juan J. Vazquez, Ulisse Mini, and Monte MacDiarmid. Steering language models with activation engineering, 2024. URL https://arxiv.org/abs/2308.10248. arXiv preprint arXiv:2308.10248v5.

Miles Turpin, Julian Michael, Ethan Perez, and Samuel R. Bowman. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. In *Advances in Neural Information Processing Systems 36*, 2023.

Kevin Ro Wang, Alexandre Variengien, Arthur Conmy, Buck Shlegeris, and Jacob Steinhardt. Interpretability in the wild: a circuit for indirect object identification in GPT-2 small. In *The Eleventh International Conference on Learning Representations (ICLR)*, Kigali, Rwanda, 2023. URL https://openreview.net/forum?id=NpsVSN6o4ul.

Weixuan Wang, Jingyuan Yang, and Wei Peng. Semantics-adaptive activation intervention for LLMs via dynamic steering vectors. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=8WQ7VTfPTl.

Sean Welleck, Jiacheng Liu, Ronan Le Bras, Hanna Hajishirzi, Yejin Choi, and Kyunghyun Cho. 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, 2021. URL https://openreview.net/forum?id=Jvxa8adr3iY.

David P. Woodruff, Vincent Cohen-Addad, Lalit Jain, Jieming Mao, Song Zuo, MohammadHossein Bateni, Simina Brânzei, 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, Ravi Kumar, Silvio Lattanzi, Euiwoong Lee, Yi Li, Ioannis Panageas, Dimitris Paparas, Benjamin Przybocki, Bernardo Subercaseaux, Ola Svensson, Shayan Taherijam, Xuan Wu, Eylon Yogev, Morteza Zadimoghaddam, Samson Zhou, Yossi Matias, James Manyika, and Vahab Mirrokni. Accelerating scientific research with Gemini: Case studies and common techniques, 2026. URL https://arxiv.org/abs/2602.03837. arXiv preprint arXiv:2602.03837v3.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems 36*, pp. 11809–11822, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/271db9922b8d1f4dd7aaef84ed5ac703-Abstract-Conference.html.

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, Huan Sun, Yu Su, and Wenhu Chen. 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 (CVPR)*, pp. 9556–9567, Seattle, WA, USA, June 2024. IEEE/CVF. URL https://openaccess.thecvf.com/content/CVPR2024/html/Yue_MMMU_A_Massive_Multi-discipline_Multimodal_Understanding_and_Reasoning_Benchmark_for_CVPR_2024_paper.html.

Pardis Sadat Zahraei and Ehsaneddin Asgari. TuringQ: Benchmarking AI comprehension in theory of computation. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 12267–12280, Miami, Florida, USA, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.715. URL https://aclanthology.org/2024.findings-emnlp.715/.

A Formal Background

This appendix summarizes the formal machinery underlying regex, DFA, and the procedural constructions used across this work. Definitions in A.1–A.4 underpin the diagnostic evaluation (the understanding and construction sections); definitions in A.5–A.7 underpin the fine-tuning pipeline. Definitions in A.1–A.3 and A.5–A.6 follow Sipser (2013); A.4 follows Brzozowski (1964) and A.7 follows Hopcroft (1971).

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 the understanding evaluation.

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,
- $\delta : Q \times \Sigma \rightharpoonup Q$ is the transition function, completed to a total function by directing any undefined pair to a dead state q_d,
- $q_0 \in Q$ 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 the understanding evaluation, 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 the DFA construction evaluation, nullability propagation is a required step in Brzozowski derivative computation (Appendix A), 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 the DFA construction evaluation (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:

$$
\begin{aligned}
Q &= \{\, [D_w(R)]_\sim : w \in \Sigma^* \,\}, \\
q_0 &= [R]_\sim, \\
F &= \{\, [D]_\sim \in Q : \mathsf{Null}(D) \,\}, \\
\delta([D]_\sim, a) &= [D_a(D)]_\sim,
\end{aligned}
$$

where $[\cdot]_\sim$ denotes the equivalence class of a regular expression under *similarity* ($\sim$): the congruence generated by associativity, commutativity, and idempotence of union ($R \mid R \sim R$) together with the identity and annihilation laws ($\varepsilon R \sim R$, $R \mid \emptyset \sim R$, $\emptyset R \sim \emptyset$). Identifying derivatives modulo similarity is essential: without it, repeated differentiation can produce infinitely many syntactically distinct expressions denoting the same language, and Q need not be finite. Under similarity, Brzozowski (1964) shows that R has finitely many distinct derivatives, so Q is finite and the construction terminates; this finiteness corresponds, via the Myhill–Nerode theorem, to the finitely many semantic left quotients of $L(R)$. In the DFA construction evaluation, models are asked to follow this procedure under the derivative-based construction prompt (Appendix D) as a diagnostic probe of algebraic manipulation capability. The similarity-normalisation step is exactly the algebraic reduction that Problem 2 in the construction evaluation (Section 4) probes: models that fail to identify equivalent residuals produce derivative automata with spurious, unmerged states.

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 (as similarity classes, representatives shown) $\{[ab], [b], [\varepsilon], [\emptyset]\}$ corresponding to positions in parsing the string, with $[\varepsilon]$ 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 $s \xrightarrow{\varepsilon} t$.
- **Empty language** $\emptyset$: two states s and t with no transition between them, so no string is accepted.

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 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: $\emptyset \to D_{\text{dead}}$.
- $D_2 = \{s_3\}$ (accepting). On a: $\emptyset \to D_{\text{dead}}$; on b: $\emptyset \to D_{\text{dead}}$. No new state sets are generated, so construction halts.

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.

Construction	Used in	Role
Regular expressions (A.1)	Understanding, Construction, Mitigation	Task inputs throughout
DFA (A.2)	Understanding, Construction, Mitigation	Target output throughout
Nullability (A.3)	Understanding, Construction	Failure mode (Nullability Neglect, Problem 2)
Brzozowski derivatives (A.4)	Construction	Derivative-based construction diagnostic
Thompson's construction (A.5)	Mitigation	CoT trace generation pipeline
Subset construction (A.6)	Construction, Mitigation	Ground-truth DFAs for the diagnostic benchmark (Appendix C); Tier 4 benchmark and CoT trace pipeline
Hopcroft minimisation (A.7)	Construction, Mitigation	DFA equivalence check in construction eval (Appendix F); final minimised DFA and CoT trace pipeline

Table 3: Formal constructions and their roles across the paper.

0. **Reachability pruning.** Remove all states not reachable from q_0. Partition refinement merges indistinguishable states but does not eliminate unreachable ones, so a DFA containing unreachable states (as in Problem 6 of the construction evaluation) does not minimise to its canonical form unless such states are pruned first.
1. Initialise partition $P = \{F,\ Q \setminus F\} \setminus \{\emptyset\}$ (omitting the empty block when $F = \emptyset$ or $F = Q$, in which case $P = \{Q\}$ and the DFA is already minimal).
2. Maintain a worklist $W = P$.
3. While W is non-empty, pick and remove a block A from W ($W \leftarrow W \setminus \{A\}$). For each $a \in \Sigma$, compute $X = \{q \in Q \mid \delta(q, a) \in A\}$. For each block $B \in P$ with $B \cap X \neq \emptyset$ and $B \setminus X \neq \emptyset$, replace B in P by $B \cap X$ and $B \setminus X$. Update the worklist accordingly: if $B \in W$, replace it by both sub-blocks; otherwise add only the smaller of the two, $\arg\min(|B \cap X|, |B \setminus X|)$. Adding only the smaller sub-block is what yields the $O(|\Sigma|\, n \log n)$ bound; adding both unconditionally reduces the procedure to Moore's $O(n^2)$ refinement.
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 the fine-tuning experiments include the full Hopcroft partition refinement trace, recording each split step as a separate JSON object.

A.8 Relevance to This Paper

Table 3 summarises which formal constructions are exercised in each part of the paper.

B Cross-Consistency Protocol Example

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

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

The completion below was produced by Qwen 2.5-7B-Instruct at temperature 0.0 under the Cross-Consistency prompt (Appendix D); it is a single unedited run, shown as a representative instance of the failure modes cataloged in Section 3.1. The ground-truth DFA for this regex is given in Table 4.

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 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 $a(bb \mid bbb)^*ba$ 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 $a(\ldots)^*b$), and inside the $a(\ldots)^*b$ alternative another Kleene star over the chunk $a(bb \mid bbb)^*ba$. 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 "$\ldots bab$" or "$\ldots b$" are forbidden.

TASK 2: EXAMPLE-BASED REASONING

Accepted strings (✓ = genuinely in $L(R)$ per the ground-truth DFA; ✗ = not in $L(R)$):
$aabba$ ✓, $aaabbba$ ✗, $baaabbba$ ✗, $bbaabbbba$ ✓, $aaababbba$ ✗, $aabbaabbbba$ ✗, $baababbba$ ✗, $abbbaabbbba$ ✓, $babaaabbba$ ✗, $ababbbaababbba$ ✗

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

TASK 3: SYMBOLIC EXECUTION ON EDGE CASES

- **aaababbb** → **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** → **Rejected.** The regex forces the string to end with bba; however, this string ends with aab, which violates the required final structure.
- **ababbbaab** → **Rejected.** Although the last symbol is b, the final three characters are aab, not bba. The required tail structure is missing.

Verifier Report. Checking the completion against the ground-truth DFA (Table 4) exposes the following, matching the taxonomy of Section 3.1:

- **Task 1 (Anchor Hallucination).** The description asserts every string "ends with at least three `b`'s in a row," but the regex's literal suffix is `bba`, so every accepted string ends in `a`. The model invents a suffix the language does not have.
- **Task 1 vs. Task 2 (Descriptive–Operational Dissonance).** Task 1 states no accepted string can end in `a`, yet every string in the model's own accepted list ends in `a`, contradicting its own description.
- **Task 2 (Atomic Unit Blindness).** Strings such as `aaabbba` require $(bb \mid bbb)^*$ to emit a single `b`, impossible since $2x+3y = 1$ has no non-negative integer solution; the ground-truth DFA sends this string to the dead state q_d.
- **Task 3.** The justifications require the suffix `bba`, contradicting the Task 1 claim of a `bbb` ending.

Of the ten strings in the model's accepted list, only 3 ($aabba$, $bbaabbbba$, $abbbaabbbba$) are in fact accepted by the ground-truth DFA; the other 7 are rejected (marked above). The model's own generated examples are thus mostly inconsistent with the language it was asked to exemplify.

C Dataset Details

C.1 Diagnostic Benchmark

The diagnostic evaluation (understanding and construction) 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 the understanding evaluation (nullability neglect, anchor hallucination, scope confusion) and the construction evaluation (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 DFA 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 4, and the corresponding state diagram is shown in Figure 5. 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.

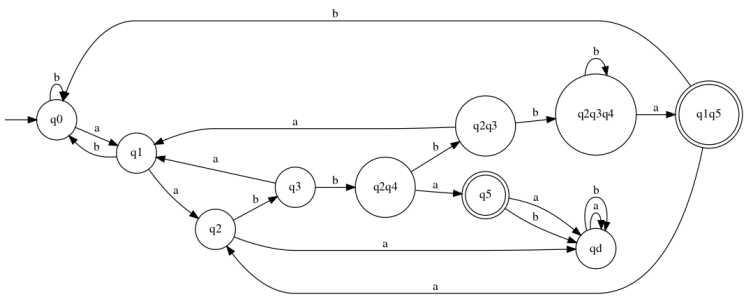


Figure 5: State diagram for the example benchmark regex R. Red transitions indicate paths to the dead state; composite state labels indicate NFA state sets from subset construction.

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 4: 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.

Usage in This Paper. This benchmark is used exclusively for controlled evaluation in the understanding and construction evaluations. In the understanding evaluation, each of the 180 regexes serves as input to the Cross-Consistency Protocol across three tasks. In the construction evaluation, the same problems are used to evaluate DFA construction under both intuitive and derivative-based prompting strategies. The benchmark is not used for training at any point. In contrast to the fine-tuning benchmark introduced for the mitigation experiments (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.

C.2 Regex-to-DFA Fine-Tuning Benchmark

The mitigation experiments use a new regex-to-DFA benchmark introduced in this work, distinct from the diagnostic benchmark in Appendix C. 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) $\rightarrow$ ε-closure and subset construction (Appendix A) $\rightarrow$ Hopcroft's minimisation (Appendix A).
3. **Verification.** The NFA, pre-minimisation DFA, and minimised DFA are verified for language equivalence symbolically: each pair is compared by minimising both automata with Hopcroft's algorithm and testing the minimal automata for isomorphism, which is exact, polynomial in the number of states, and independent of alphabet size (so it applies uniformly across all alphabets, including *digits* with $|\Sigma| = 10$). This confirms $L(\text{NFA}) = L(\text{DFA}) = L(\text{minDFA})$; 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** (≈15% of data): direct symbol mapping and simple concatenation/union; produces 2–3 state minimised DFAs.
- **Tier 2** (≈25%): introduces Kleene star, plus, and optional operators; produces small looping DFAs with 2–4 states.
- **Tier 3** (≈35%): combined operators with longer concatenations and star-of-union patterns; produces mid-size DFAs with 4–6 states.
- **Tier 4** (≈25%): deeply nested closures of concatenations; produces larger DFAs with 6–18 states requiring full end-to-end subset construction.

All tier state counts refer to live states, excluding the dead state. Tier 4 is the primary focus of the mitigation analysis 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 Tier 4 regex $(ab \mid ba)^*a(bb)^*$ over $\Sigma = \{a, b\}$ (20-state NFA, 9-state subset DFA, 8-state minimised DFA including the dead state), illustrating the trace structure. The input prompt matches the No-CoT format below; the only difference is that the CoT output includes the full construction trace before the final DFA, which is what we show here.

```
=== TRACE BEGIN ===
### Original Regex
{"regex": "(ab|ba)*a(bb)*"}
### Regex After Concat Insertion
{"regex_with_concat": "(a.b|b.a)*.a.(b.b)*"}
### Thompson Step (Symbol)
{"token": "a",
 "created_fragment": {"start": "s0", "accept": ["s1"]}}
### Thompson Step (Symbol)
{"token": "b",
 "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] ...
### Thompson Step (Star)
{"frag_start": "s8", "frag_accepts": ["s9"],
 "new_start": "s10", "new_accepts": ["s11"]}
... [Thompson steps continue] ...
### Subset Construction Step
{"dfa_state": "D0",
 "nfa_subset": ["s0","s4","s8","s10","s11","s12"],
 "transitions": {"a": "D1", "b": "D2"}}
### Subset Construction Step
{"dfa_state": "D1",
```

```
 "nfa_subset": ["s1","s2","s13","s14","s18","s19"],
 "transitions": {"a": "Ddead", "b": "D3"}}
... [Subset construction steps continue] ...
### Hopcroft Refinement
{"symbol": "b",
 "split_block": ["D0","D3","D4"],
 "inter": ["D3"],
 "diff": ["D0","D4"]}
... [Hopcroft refinement steps continue] ...
### Final Renamed DFA
{"states": ["q0","q1",...,"q7"],
 "start_state": "q0",
 "accept_states": ["q1","q5","q7"],
 "transitions": {"q0": {"a":["q1"],
                        "b":["q2"]}, ...}}
=== TRACE END ===
=== FINAL DFA BEGIN ===
states: q0 q1 q2 q3 q4 q5 q6 q7
alphabet: a b
start: q0
final: q1 q5 q7
transitions:
q0 a q1
q0 b q2
...
=== 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 Sigma = {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 generated pool is split by random shuffle with seed 42 into 22,500 training and 1,000 test examples. A separate validation set of 400 examples is used for checkpoint selection: 200 in-distribution (IID) examples held out from the same generation distribution as training, and 200 out-of-distribution (OOD) examples generated with structural patterns and operator combinations not seen during training. Evaluating on both IID and OOD examples during checkpoint selection guards against selecting checkpoints that merely memorize training-set patterns.

Bounded Instances and Computational Complexity. Regex-to-DFA conversion has worst-case exponential cost, since subset construction can produce a DFA exponentially larger than the input regex. This asymptotic cost does not bear on our findings, for three reasons. First, our benchmark is bounded by construction: Tier 4 target DFAs contain at most 18 states, so no instance sits in the blow-up regime, and worst-case conversion cost is never exercised. Second, the objection does not apply to the understanding evaluation at all: the Cross-Consistency Protocol (the understanding evaluation) involves no automaton construction, so no notion of construction cost is relevant to the eleven semantic, generative, and execution failure modes documented there. Third, even on an instance that were genuinely too large to complete within the context budget, a sound reasoning system should recognize and report that it cannot finish the construction, rather than confidently emitting a structurally invalid DFA. What we observe is the latter: models produce incorrect DFA with no indication of difficulty, which is a reasoning failure rather than a resource limit. The failures we document are therefore structural errors on solvable, bounded instances, not a model running out of room on an intractable one.

C.3 Curriculum Phase Datasets

The phase-wise curriculum learning experiments use a reorganised version of the fine-tuning benchmark described in Appendix C, 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 the smallest DFAs (2–3 live states).
- **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. In this format the dead state is implicit: undefined transitions are shown as – and excluded from the reported state count, whereas the JSON-trace format and the final transition tables (Appendix A) include it explicitly. 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 DFA. 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 |
    4 DFA states before minimisation.
    ## Step 6: Minimisation (Hopcroft's Procedure)
    Reduced 4 -> 3 states.
    ## Step 7: Minimised DFA
    States: {D0,D1,D2} | Start: D0 | Accept: {D2}
    | State | 0 | 1 | Accept? |
    | 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.

C.4 Curriculum Parity with Joint Training

A natural concern is whether the large gap between phase-wise curriculum training (43.5% overall) and joint CoT SFT (96.5%) reflects reduced compute or data in the curriculum setting rather than the effect of ordering itself. It does not. The two settings are matched on every factor we control, and curriculum training in fact receives *more* total optimisation than joint SFT. We summarize the comparison in Table 5.

Data parity. Phases 1–3 and 5 train exclusively on Tiers 1, 2, 3, and 4 respectively (Appendix C). Because these four tiers together constitute the entire 22,500-example construction dataset (Ap-

Table 5: Resourcing comparison between joint CoT SFT and phase-wise curriculum training. The two settings share an identical model, adapter, optimiser, and underlying supervision target; curriculum training sees the same data pool plus replay, and runs for more epochs at a larger batch size.

Factor	**Joint CoT SFT**	**Phase-wise Curriculum**
LoRA config	rank 32, α 64	identical
optimiser	AdamW, cosine, lr 2×10^{-4}	identical
Supervision target	Thompson $\rightarrow$ subset $\rightarrow$ Hopcroft	identical
Data volume	full 22,500-example set	same construction pool, split by tier, plus tier-replay and 30% theory Q&A
Trace format	JSON trace	8-step markdown (JSON-trace control run reported in text)
Epochs	2	3 per phase
Effective batch	8	16
Early stopping	none	patience 5

pendix C), the four tier-specific phases jointly cover the same construction data that joint SFT trains on, partitioned by tier rather than shuffled together. Phase 4 then adds a balanced mixture sampled from all tiers as a replay step against forgetting, and each phase additionally interleaves theory Q&A examples (Appendix C), so curriculum training sees at least as much data as joint SFT, and strictly more once replay and the theory examples are counted. The theory Q&A is held at the same 70/30 proportion in every phase, so it is constant across all five orderings and cannot account for the ordering-dependent differences we observe (best, easy-to-hard, 43.5%; worst, reverse, 26.0%).

Compute parity. Beyond data, curriculum training runs for more epochs per phase (3 versus 2) at a larger effective batch size (16 versus 8) with identical LoRA rank, optimiser, learning-rate schedule, and random seed (Appendix D). It therefore performs at least as many total gradient updates as joint SFT. Reduced compute cannot explain the gap. Nor can the trace format: the phase files use the 8-step markdown format (Appendix C), but a control run of the natural-order curriculum on the 7B model using the JSON-trace format of joint SFT gives a similar result (43.1% overall, 38.9% on Tier 4), so the collapse is not an artefact of format. Early stopping (patience 5) applies only to the curriculum runs, but the reported figures are the best validation checkpoint, not a prematurely halted one, so it does not explain the gap either.

Interpretation. With data and compute matched or exceeded, the collapse is not explained by under-training, but it reflects two distinct sequential-training effects rather than a single cause. The degraded accuracy on the *earlier* tiers (1–3) is consistent with *catastrophic forgetting*: these tiers are trained in Phases 1–3 and, apart from the Phase 4 replay, are not reinforced during the final Tier 4 phase. The low *Tier 4* accuracy itself, however, cannot be attributed to forgetting, since under the natural ordering Tier 4 is trained in Phase 5 immediately before evaluation — the model cannot have forgotten the task it has just finished training on. This deficit instead reflects reduced adapter plasticity after sequential adaptation through the earlier phases, together with possible interference from the interleaved theory Q&A examples present in the curriculum runs but absent from joint SFT. Two observations support this combined account. First, the pattern of results is ordering-dependent in a way under-training would not predict: orderings that progress from easier to harder tiers perform best, while orderings that violate this progression (reverse, hard-first) perform worst, despite identical total resources. Second, the Tier 4 deficit is consistent with the sequential structure of the task. Merrill & Sabharwal (2024) show that transformers answering without intermediate generation cannot recognise all regular languages without intermediate steps, and that the computational benefit of chain-of-thought grows with the number of intermediate steps the model may condition on. DFA construction is precisely this class of problem: Thompson construction, subset construction, and Hopcroft minimisation together form one continuous sequential procedure. Joint SFT exposes the model to all tiers throughout training, so the state-tracking skills that Tier 4 requires are acquired alongside the simpler ones; phase-wise training introduces them only in the final phase, after the adapter has specialised to the simpler tiers, and the Phase 4 replay is not enough to restore the plasticity needed to then learn Tier 4. This offers a mechanistic account of why matching compute and data does not close the gap: the deficit lies in how the sequential structure of the procedure is represented and in the loss of plasticity

across sequential phases, not in how much the model was trained. Fully disentangling forgetting from plasticity loss would require a rehearsal schedule keeping all tiers present as an anchor mixture across every phase; we leave this to future work, noting that such a schedule converges toward the joint-training regime our results already show to be superior.

C.5 On the CoT versus No-CoT Inversion at Tier 4

At Tier 4, direct transition-table generation (No-CoT) matches or exceeds step-by-step trace generation (CoT) at every scale (82.0/85.4/87.7% vs. 80.5/82.9/82.1% at 1.5B/7B/14B), inverting the usual advantage of chain-of-thought. We note two points about this inversion.

First, it is not a matter of model capacity or trainable parameters. Both formats use an identical LoRA configuration (rank 32, α 64, the same target projection matrices, frozen base weights; Appendix D) and the same optimiser, schedule, and data. The CoT and No-CoT models therefore have the same number of trainable parameters; the only difference is the supervision target, and hence the length and content of the sequence the model must generate.

Second, the most plausible account of the inversion is error accumulation over long generations rather than any deficit specific to reasoning. A Tier 4 CoT target is a multi-thousand-token JSON trace recording every Thompson fragment, the complete NFA transition table, each power-set subset-construction step, and each Hopcroft partition split (Appendix C). Under our exact-match criterion (Appendix F), a single mislabeled state or malformed step anywhere in that trace can invalidate the final automaton, so the probability of an output-invalidating slip grows with trace length (Dziri et al., 2023). No-CoT instead emits only the final transition table, a few dozen tokens, and so exposes far less surface to cascading token-level error. This account predicts that the No-CoT advantage should be concentrated on the longest CoT targets, and our data bear this out. Binning the held-out examples by minimal-DFA size, CoT and No-CoT are effectively tied on small automata (accuracy gap -0.02 for sizes below six states, $n = 145$) and No-CoT pulls ahead only on larger ones (gap $+0.11$ for six or more states, $n = 55$, rising to about $+0.25$ at eight or more states). Consistently, the cases where No-CoT succeeds but CoT fails have much larger target automata than average (mean minimal-DFA size 9.7 versus 4.8) and longer CoT targets (mean 3,500 output tokens, our generation cap, versus 2,200 overall), whereas the rarer cases where CoT wins and No-CoT fails have small automata (mean size 3.3). This length-dependent crossover is the pattern error accumulation predicts. We report it as suggestive rather than conclusive, given the modest number of divergent cases (six No-CoT-only successes, three CoT-only), and we cannot fully separate this length penalty from the shared representational limitation that bounds both formats (the residual Tier 4 gap persists under No-CoT as well; Section 5).

D Model and Training Details

D.1 Qwen 2.5 Architecture

All fine-tuning and mechanistic analysis experiments use the Qwen 2.5-Instruct family (Qwen et al., 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 6: 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 (Section 5.2) 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 LoRA configuration is identical across all model sizes and supervision formats (CoT, No-CoT, and curriculum); the optimiser settings below apply to the joint CoT and No-CoT runs, with the curriculum differences noted at the end of this section:

- **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

optimiser and Scheduler.

- **optimiser**: 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, optimiser, learning-rate schedule, and seed are applied sequentially across phases, with three differences: 3 epochs per phase (rather than 2), gradient accumulation of 16 (effective batch size 16, rather than 8), and early stopping with patience of 5 evaluation steps (Table 5). Each phase trains on its phase-specific dataset, and the trained adapter from each phase initialises the next, implementing the sequential fine-tuning chain evaluated in Section 5.1.

D.3 Prompting Templates

We use three categories of prompting templates depending on the experiment. All frontier model experiments (the understanding and construction evaluations) use zero-shot prompting with temperature 0.0. For the baseline experiments in the mitigation experiments, 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 G. 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. The full prompt is:

```
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.
```

The output schema fixes the fields for states, alphabet, start state, accept set, and a complete transition function.

(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. The full prompt is:

```
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, ...
```

This tests nullability propagation, normalisation of equivalent residuals, and derivative rule application. It is the prompt used for derivative-based construction in the construction evaluation.

(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. The full prompt is:

```
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:
{{EDGE_CASE_1}}
{{EDGE_CASE_2}}
{{EDGE_CASE_3}}

Output EXACTLY:
{{EDGE_CASE_1}} -> ACCEPTED/REJECTED: <1-2 line justification>
{{EDGE_CASE_2}} -> ACCEPTED/REJECTED: <1-2 line justification>
{{EDGE_CASE_3}} -> ACCEPTED/REJECTED: <1-2 line justification>
```

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.

(4) Few-Shot Demonstrations. The few-shot baseline (Appendix G, Table 10) prepends three fixed demonstrations to the zero-shot construction prompt. The demonstrations are selected deterministically (random seed 42) as one example each from Tiers 1, 2, and 3 of the training set, held constant across all model sizes and all test instances; each demonstration's chain-of-thought response is truncated to 1,500 tokens, which retains the full construction trace through the final Step 7 transition table. The three exemplars are the regexes $(b \mid c \mid a)b$ (Tier 1, $\Sigma = \{a, b, c\}$, 3-state minimal DFA), b^*aaa (Tier 2, $\Sigma = \{a, b\}$, 4-state), and $((a^* \mid b(b \mid a)^*bb))^*$ (Tier 3, $\Sigma = \{a, b\}$, 5-state), each presented as the full 8-step markdown trace with `D0,D1,...` state naming, matching the format the parser expects. Temperature is 0.0.

No Tier 4 demonstration is included, so the few-shot Tier 4 accuracy reflects in-context performance under easier-tier demonstrations rather than a measure of Tier 4 construction ability, which the fine-tuning results (Section 5.1) instead measure. Even though the demonstrations show the complete, correctly formatted output, the majority of few-shot outputs remain unparseable (75.0% for 7B, 93.5% for 14B; Table 10), indicating that three in-context exemplars are insufficient to induce reliable schema compliance.

D.4 Construction Prompting Configuration

The construction evaluation issues two independent zero-shot prompts per regex, one per construction strategy (intuitive and derivative-based; Appendix D), and records each result separately. We perform no search over intermediate states: there is no depth expansion, beam search, branch scoring, or backtracking.

Parameters.

- **Branches**: 2 — intuitive construction (Appendix D, prompt 1) and derivative-based construction (Appendix D, prompt 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 construction 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 construction driver implements retry logic, automatic JSON extraction, output sanitization, and rate limiting, and was applied uniformly to all three frontier models (GPT-5.2, Grok-4.1, Gemini-2.5). Grok-4.1 nonetheless frequently exhausted the per-branch retry limit and the generation token budget before emitting a parseable DFA, yielding too few complete constructions to support the per-mode failure analysis; it is therefore omitted from that analysis (Section 4) while remaining in the understanding evaluation of Section 3. Concretely, of the 180 problems per strategy, Grok-4.1 produced a complete, parseable DFA for only 4 under intuitive construction and 2 under derivative-based construction; the remaining runs exhausted the per-branch retry limit and generation-token budget without emitting a complete DFA. The non-completions are therefore failures to converge on an automaton within the budget rather than JSON-formatting rejections: the outputs were truncated or absent, not malformed. Among the six completed constructions, the two derivative outputs were correct and two of the four intuitive outputs exhibited failure modes from the P1–P6 taxonomy. Six completions are far too few to estimate a failure-mode distribution, so Grok-4.1 is omitted from Figure 4; we report these counts for transparency rather than as a basis for comparison. The full driver is provided in the artifact repository (Appendix J).

E The Scratchpad Objection and Construction Prompting

A natural objection to the construction evaluation (Section 4) is that both prompts request the DFA directly, instructing the model not to emit intermediate reasoning (Appendix D). One might therefore worry that the construction failures reflect the *absence of an external scratchpad* rather than a limit on the model's representational capacity: theoretical results establish that a fixed-depth transformer emitting no intermediate tokens is confined to a low circuit-complexity class and cannot, in a single forward pass, carry out multi-step procedures such as subset construction, whereas emitting intermediate steps strictly increases what it can compute, with the benefit growing in the number of

steps conditioned on (Merrill & Sabharwal, 2024). We address this objection directly, and conclude that it does not account for our findings.

The objection makes a testable prediction. If the Tier 4 gap were caused by the lack of a scratchpad, then supplying intermediate reasoning should close it. We have two independent tests of this prediction, one under controlled fine-tuning and one under free-form inference-time prompting, and it fails in both.

Controlled test: CoT vs. No-CoT fine-tuning. Our supervised fine-tuning (Section 5.1) compares two formats that differ *only* in whether an emitted scratchpad is present: the CoT format writes the full construction trace (Thompson $\rightarrow$ subset $\rightarrow$ Hopcroft) before the final DFA, whereas No-CoT emits the DFA directly. Both are trained on identical data with identical hyperparameters, so the comparison isolates the effect of the scratchpad itself. If the scratchpad were the missing ingredient, CoT should dominate at Tier 4. It does not: at Tier 4, No-CoT matches or exceeds CoT at every scale ($82.0/85.4/87.7\%$ for No-CoT at $1.5\text{B}/7\text{B}/14\text{B}$ versus $80.5/82.9/82.1\%$ for CoT). A serialized reasoning trace, supplied by training, does not recover the gap. Because Merrill & Sabharwal (2024) predict that emitted steps should help when the task requires serial computation, the *absence* of a CoT advantage here is evidence that the missing capability is not the scratchpad but the underlying construction competence encoded in the weights.

Inference-time test: unconstrained CoT on the 180 problems. The controlled test above uses fine-tuned Qwen models. The exact experiment the objection calls for—allowing a frontier model to emit free-form intermediate reasoning on the 180-problem diagnostic set—has been run independently by the benchmark's authors. Shelat et al. (2026) report DFA-construction success on the same 180 Unseen problems under zero-shot, chain-of-thought, and one-shot chain-of-thought prompting; across all three frontier models, emitted chain-of-thought lands at or below the zero-shot rate (e.g., zero-shot vs. CoT of 20.67% vs. 16.67%, 59.12% vs. 51.90%, and 29.33% vs. 23.33% for the three models respectively). Allowing an external scratchpad at inference time thus does not improve construction on these problems, corroborating the controlled result under a different model family and a different intervention.

On the "Tree-of-Thought" condition. Shelat et al. (2026) additionally report a condition they label "Tree-of-Thought." By its mechanics this is the two-branch parallel prompting we describe in Appendix D: two independent construction prompts generated once each, with no heuristic evaluation of intermediate thoughts and no state-space search in the sense of Yao et al. (2023). We therefore describe our own construction setup as parallel zero-shot prompting rather than Tree-of-Thought, and note that where this condition raises success (for one model it reaches 54%), it does so by attempting two strategies rather than by searching, and still leaves construction far from solved.

Conclusion. The scratchpad objection predicts that emitted intermediate reasoning should close the Tier 4 gap. Neither a trained-in scratchpad (controlled, CoT vs. No-CoT) nor a free-form inference-time scratchpad (external, on the 180 problems) does so. We conclude that the residual construction failures reflect a representational limitation formed during pretraining rather than the absence of inference-time reasoning tokens.

F Evaluation Protocols

This appendix describes the evaluation methodology for all experiments in this paper. The understanding and construction evaluations (Sections 3 and 4) involve both human annotation and automatic verification. Mitigation evaluations are fully automatic with programmatic correctness checking.

F.1 Human Evaluation

All human annotation was performed by a team of 10 members of the research team, not paid crowdworkers, and no external recruitment platform was used. Annotators were fully blinded to model identity and evaluated outputs against formal correctness criteria under the fixed rubrics described below. Each output was independently annotated by two annotators. For the cross-consistency evaluation, conflicts were resolved by a senior reviewer; for the DFA construction

evaluation, conflicts were resolved by discussion among the annotators. This blinded, rubric-based process applies to all human annotation described in this section.

F.1.1 Cross-Consistency Protocol

The Cross-Consistency Protocol evaluation requires human assessment of three tasks per model output. For each model output, the two annotators 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.

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 descriptive–operational dissonance and used to populate the semantic failure taxonomy of the understanding evaluation.

F.1.2 DFA Construction Evaluation

For the DFA construction experiments, generated DFAs are evaluated for correctness against the ground-truth DFA from the 180-problem benchmark. Evaluation proceeds in two stages, followed by a separate failure-mode annotation pass:

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 same exact criterion as the mitigation experiments (Appendix F): a bounded-length membership check on all strings up to length 6 over the relevant alphabet serves as a fast pre-filter, and every DFA that passes is then verified by minimising both automata with Hopcroft's algorithm (Appendix A) and testing the two minimal DFAs for isomorphism. Since the minimal DFA of a regular language is unique up to state relabeling, this isomorphism test certifies language equivalence independent of any length bound.

Failure mode annotation. For the failure mode analysis in the construction evaluation and Figure 4, each incorrect DFA is additionally annotated with the failure modes (P1–P6) from the taxonomy, following the same two-annotator, blinded process described in Appendix F, with construction disagreements resolved by discussion among the annotators. Reviewers are provided with the ground-truth DFA and the regex for reference. Failure mode labels are not mutually exclusive: a single DFA may exhibit multiple failure modes simultaneously, as reflected in Figure 4. Where a failure mode is specific to a construction strategy, it is scored only under that strategy. Problem 2 (Brzozowski-derivative errors) is assigned only to derivative-based outputs, by inspecting the emitted residual and nullability steps; it is not assigned to intuitive outputs, which contain no derivative steps to evaluate. Nullability errors in intuitive outputs are instead identified from the automaton's acceptance behaviour and recorded under Problem 4 (over-acceptance).

F.2 Automatic Evaluation

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

F.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 DFA, verified by an exact isomorphism test on the minimised automata (described below). A bounded-length string check is used only as a fast pre-filter, not as the equivalence criterion itself.

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.

Equivalence testing. We test language equivalence in two stages. First, as a fast pre-filter, we compare the predicted and ground-truth DFAs on all strings $w \in \Sigma^*$ with $|w| \leq 6$: for $|\Sigma| = 2$ this yields $\sum_{i=0}^{6} 2^i = 127$ strings, and for $|\Sigma| = 3$ it yields $\sum_{i=0}^{6} 3^i = 1{,}093$ strings. Any predicted DFA that disagrees with the ground truth on a short string is immediately marked incorrect. This cheaply eliminates the large majority of incorrect predictions, whose errors — incorrect suffix or boundary conditions, collapsed or merged states, and over-acceptance of non-language strings — produce a distinguishing string well within this length.

Bounded-length agreement does not by itself guarantee equivalence, since two inequivalent DFAs with many states can agree on all short strings; a distinguishing string is only guaranteed to exist below length $m + n$ for DFAs with m and n states. Because our benchmark contains minimal DFAs of up to 18 states, a fixed bound of 6 is not sufficient to certify equivalence on its own. We therefore apply an *exact* test to every predicted DFA that passes the pre-filter: we minimise both the predicted and ground-truth DFAs using Hopcroft's algorithm (Appendix A) — first pruning any states unreachable from the start state, so that redundant or disconnected subgraphs (Problem 6) do not affect the minimal form — and test whether the two minimal DFAs are isomorphic. Since the minimal DFA for a regular language is unique up to relabeling of states, isomorphism of the minimised automata is an exact test of language equivalence, independent of any length bound. A prediction is counted correct if and only if it passes this isomorphism test. Applying the exact test produced results identical to the pre-filter on our evaluation set, confirming that the structural failure modes documented throughout this work are caught by short distinguishing strings.

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 Appendix G.

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.

F.2.2 Parser Validation

A potential concern with automated scoring is that a model using an unexpected output style could be under-credited by the parser rather than by genuine reasoning failure. Two aspects of our setup make this unlikely by design: the required output format is strictly specified in the prompt (Appendix C),

and DFA correctness is determined by semantic language-equivalence verification (Appendix F), not by matching surface format strings. Correctness therefore depends only on whether a valid DFA can be extracted and whether it recognizes the target language, not on stylistic differences in presentation.

To verify this empirically, we validated the parser against human judgment on a stratified sample of 360 outputs from the fine-tuned Qwen 2.5 models (60 per model size and supervision format, balanced across tiers). For each output, a human reviewer independently determined whether a valid DFA was present and whether it was correct, and we compared this to the parser's verdict. The parser agreed with the human judgment on all 360 outputs (100% agreement), with no false parse failures: there was no case in which a model produced a valid DFA that the parser failed to extract. In particular, the residual Tier 4 errors of the CoT models reflect genuine construction errors rather than format mismatch. We conclude that automated scoring does not conflate reasoning failure with format mismatch.

F.2.3 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:

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

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

F.2.4 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 set is a separate draw of 285 Tier 4 examples from the same generator (seed 42), disjoint from the training, validation, and test sets, and reserved exclusively for mechanistic analysis. Its baseline accuracy (85.3%) differs slightly from Tier 4 accuracy on the test set (82.9%) because it is an independent sample.

ADFA variant specificity. We use three related but distinct measurements, and state them separately to avoid confusion. (i) The *specificity score* $\text{Specificity}(l, i) = \frac{V_1+V_2}{2} - V_3$ (Section 5.2) is a *difference*, used to rank neurons. (ii) The *V3 bleed ratio*, the ratio of a neuron's mean activation on V3 to its mean activation on V1 and V2, is a *proportion*, used for a threshold test: a neuron is construction-specific if its bleed ratio is below 0.5, i.e., it fires at least twice as strongly on the construction-requiring inputs (V1, V2) as on the decoy (V3). No neuron in our 56,000-candidate pool meets this threshold; the lowest observed bleed ratio is 0.73. (iii) Separately, as an illustrative single-neuron statistic, the most selective candidate (Layer 27) fires at 91% of its *own peak activation* on V3; this is relative to the neuron's peak, not to V1/V2, and is not comparable to the bleed-ratio threshold.

Validity of the V1–V3 contrasts. Two clarifications address why these contrasts isolate construction-specific computation. First, V1 (alphabet renaming) does change token embeddings, and could shift activations for that reason alone; we therefore never treat V1 in isolation as evidence of specificity. Specificity is defined jointly, requiring high activation on both V1 and V2 together with low activation on V3. A neuron that merely tracked token identity would not remain strongly active across the alphabet swap, so surviving the surface change is the signal, not the existence of any difference. Second, V3 is a regex that appears complex but whose minimal DFA is trivial (2 states). The standard pipeline does nominally run subset construction on any regex before minimisation reveals the collapse, so V3 is not intended to test whether the procedure is invoked at all. Rather, it targets the genuinely hard part of Tier 4 construction: maintaining multiple simultaneously-active NFA states, the parallel bookkeeping that produces composite DFA states such as $q_2q_3q_4$. A regex that collapses to two states keeps the active state-sets small throughout and never enters this regime, even when the procedure is run. V3 therefore separates neurons that track parallel-state maintenance

from neurons that respond only to surface regex complexity, which is the distinction relevant to Tier 4 failures.

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). Because this subset is imbalanced, raw accuracy is dominated by the majority class: a constant "correct" predictor scores 85.3%. We therefore report, across the five cross-validation folds, both raw accuracy and metrics robust to this imbalance for the Layer-10 probe: accuracy 87.0%, AUROC 0.85, and balanced accuracy 0.845 (chance 0.50 for the latter two), with Matthews correlation 0.59. On the 42 failing instances the probe attains recall 0.81 (34 of 42 flagged), precision 0.54, and failure-class F1 0.65. The probe therefore discriminates failing from succeeding trajectories well above the majority prior, confirming that the eventual outcome is encoded in the early representation rather than reflecting the class balance.

Activation patching. Activation patching tests whether correcting the model's internal representation repairs Tier 4 failures. For a failing example, we replace the residual-stream hidden state (3,584-dimensional) at a chosen layer, at a range of layers, or at all 28 layers at once with the mean hidden state of the correctly solved examples, computed over the prompt positions, and then allow generation to proceed without further intervention. Because patching sweeps every layer and layer-range, generation for this analysis is run in batches for efficiency; batched decoding yields a Tier 4 baseline of 85.3% (243 correct, 42 incorrect) on the mechanistic subset, so the patching analysis operates over the resulting 42 failing examples. We verified on a held-out sample that the batched and single-example pipelines produce identical correctness labels, so the difference reflects decoding batching alone and not a change in the verifier. As a control, the same patch is applied using the mean hidden state of the *incorrect* examples (leave-one-out). Patching the correct representation recovers no failing example at any single layer from 0 to 25 and only 4–5 at the final two layers; patching contiguous layer ranges recovers none except the final range (layers 24–27, five examples); and patching all 28 layers at once recovers none. The incorrect-source control recovers as many or more at these layers (seven versus five at the last layer), indicating that the few late-layer flips reflect perturbation of the near-final output rather than genuine repair. Correctness is checked with the same exact-match criterion as Appendix F. Because this patch injects a mean over structurally heterogeneous correct examples, it removes instance-specific conditioning and therefore speaks only to whether shifting toward an aggregate correctness direction repairs failures, not to whether an instance-accurate correct state would. To probe instance conditioning directly, we evaluated alphabet-swapped variants (V1) of identical DFA topology for the 42 failing examples (Section 5.2): 11 of 42 recover, so the failure is only partially tied to surface form. A clean instance-matched patch (injecting a correct twin's state into its failing base) would require a correct twin that is also token-aligned with the base; in our data the correct-twin and token-aligned sets do not overlap, so we do not make an instance-matched causal claim. We do not use the structural-sibling variants (V2) for this purpose either: as generated, these pair a base with a different expression of similar DFA size rather than a true isomorphic reformulation of the same automaton, so they do not hold instance structure fixed.

Steering intervention evaluation. All steering interventions in Section 5.2 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 Appendix F. 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).

F.2.5 Steering Robustness and Confidence Intervals

All steering interventions are evaluated on the same 285-example Tier 4 mechanistic subset (Appendix F), on which the CoT-trained 7B baseline achieves 85.3% accuracy (243 of 285 correct). On a set of this size, a single example corresponds to $100/285 \approx 0.35$ percentage points (pp). The reported deltas should be interpreted with this resolution in mind: a $+0.3$ pp change corresponds to

fewer than one additional example, a $+1.0$ pp change to roughly three, and the best result ($+2.4$ pp, VGNS) to seven.

We therefore emphasize that the fine-grained ordering among the sub-1 pp interventions is not statistically meaningful. Table 7 reports Wilson 95% confidence intervals for each condition. Because the interventions are evaluated on the same fixed example set with frozen weights and deterministic decoding (temperature 0.0), these intervals characterize the sampling uncertainty of the accuracy estimate on this subset.

Table 7: Wilson 95% confidence intervals for each intervention on the 285-example Tier 4 subset. All sub-1 pp interventions, including the random-neuron control, are statistically indistinguishable from one another and from the baseline.

Condition	**Correct**	**Acc. (%)**	**95% CI**
Baseline (no intervention)	243/285	85.3	[80.7, 88.9]
Probe direction	242/285	84.9	[80.3, 88.6]
Random neurons (control)	244/285	85.6	[81.1, 89.2]
Bad neurons ($\times$0.5)	244/285	85.6	[81.1, 89.2]
ActAdd	244/285	85.6	[81.1, 89.2]
Suppression ($\times$0.5)	244/285	85.6	[81.1, 89.2]
Good neurons ($\times$1.5)	245/285	86.0	[81.5, 89.5]
Functional antagonism	245/285	86.0	[81.5, 89.5]
SADI	246/285	86.3	[81.8, 89.8]
VGNS (4-round)	250/285	87.7	[83.4, 91.0]

The confidence intervals make the correct reading explicit. Every sub-1 pp intervention (ActAdd, suppression, bad-neuron scaling) produces exactly the same accuracy as the random-neuron control (85.6%, 244/285), and all of these intervals overlap almost completely with the baseline. No targeted intervention reliably exceeds random manipulation. Only VGNS produces a recovery ($+2.4$ pp, 7 examples) that moves outside the immediate single-example band, and even its interval overlaps the baseline, consistent with its modest absolute effect.

This does not weaken the paper's conclusion; it sharpens it. Our claim is not that any intervention meaningfully closes the Tier 4 gap, but the opposite: that no inference-time intervention does, because the residual gap is not recoverable by these inference-time interventions. The statistical indistinguishability of targeted interventions from random control is direct evidence for this claim. Reliable recovery of the residual gap would require a larger evaluation set and multiple seeds to resolve sub-percentage-point effects; we report the results here at their true resolution rather than implying a precision the sample size does not support.

F.3 Semantic Interpretation Checks

These checks are applied during human evaluation of the understanding-evaluation 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)).

F.4 Reproducibility

All evaluation scripts, DFA equivalence checkers, parsing tools, prompting templates, and training scripts are provided in the artifact repository and indexed in Appendix J, with usage instructions in the repository README.

F.5 Per-String Normalisation of Tasks 2 and 3

The whole-task failure rates in Figure 2 cannot be read directly as a difficulty ranking between generation (Task 2) and recognition (Task 3), because the two tasks differ in how many strings they score and how failure is aggregated (Appendix F). Task 2 is marked failed if any of its 20 generated strings is misclassified *or* if the accepted set does not cover all structural branches of the regex; Task 3 is failed if any of 3 curated edge cases is wrong. All-or-nothing aggregation over 20 strings rather than 3 amplifies any per-string difference into a larger whole-task gap, so a higher Task 2 whole-task rate is expected on cardinality grounds alone. This appendix separates the two effects that the whole-task rate conflates: per-string classification error, and branch coverage.

Per-string classification error. Table 8 reports classification accuracy at the level of individual strings, with coverage scored separately and excluded here. For Task 2 this is the fraction of all $180 \times 20 = 3{,}600$ generated strings that are correctly placed (an "accepted" string that is genuinely in the language, or a "rejected" string genuinely outside it), checked with the language-equivalence verifier of Appendix F; for Task 3 it is the fraction of all $180 \times 3 = 540$ edge cases labeled correctly. Normalised this way, the two tasks are comparable: for the three frontier models Task 2 and Task 3 per-string error differ by at most a few points (e.g. GPT-5.2, 34.2% vs. 31.5%), and for all three Qwen models Task 2 per-string error is in fact *lower* than Task 3. The large whole-task gap in Figure 2 therefore does not reflect worse string-level classification in Task 2; it is a product of item count and the coverage requirement.

Why per-string error does not favour recognition. The comparable per-string accuracy across the two tasks is consistent with the coverage failure below rather than at odds with it: Task 3's edge cases are chosen to be difficult, whereas the strings a model generates in Task 2 tend to be the easy, high-frequency members of the language it can already handle. The model is not worse at judging individual strings when it writes them; it is selective about which strings it writes.

Coverage is the structural signal. The dominant contributor to Task 2 failure is the branch-coverage requirement, which is independent of string count and of per-string classification: models generate locally valid strings that populate only the easy branch of the language and omit the harder branch entirely. This is Simple-Path Bias, mode (vi), and it is the most frequent Task 2 failure for every model (Table 1; 102 of 180 problems for GPT-5.2). Coverage failure concerns *which* strings the model chooses to generate, not how many are scored or whether each is individually correct, so it persists under any per-string normalisation. It is the component of the Task 2 peak that is genuinely structural.

What the protocol establishes. The Cross-Consistency Protocol is designed to detect *inconsistency* between a model's stated description and its behaviour (descriptive–operational dissonance), not to measure an absolute generation-versus-recognition difficulty ratio. Its robust conclusions do not depend on the whole-task Task 2/Task 3 ordering: per-string classification is comparable across the two tasks, the dominant Task 2 failure is coverage, and the eleven failure modes recur across every model family and scale. We accordingly report whole-task rates as reflecting task stringency and coverage, and rest the structural claim of Section 3 on these normalisation-invariant findings.

Table 8: Per-string classification accuracy on Tasks 2 and 3 over all 180 problems (Task 2: 3,600 generated strings; Task 3: 540 edge cases), with branch coverage scored separately and excluded. The large whole-task Task 2 peak in Figure 2 is not reflected at the per-string level, where the two tasks are comparable.

	Task 2 (generation)		Task 3 (recognition)	
Model	**Acc. (%)**	**Err. (%)**	**Acc. (%)**	**Err. (%)**
GPT-5.2	65.8	34.2	68.5	31.5
Grok-4.1	95.3	4.7	95.0	5.0
Gemini-2.5	78.4	21.6	80.0	20.0
Qwen-1.5B	36.8	63.2	28.1	71.9
Qwen-7B	49.0	51.0	39.3	60.7
Qwen-14B	60.2	39.8	51.3	48.7

F.6 Isolated-Context Control for Descriptive–Operational Dissonance

The Cross-Consistency Protocol presents Tasks 1–3 in a single prompt (Section 3), so a causal language model generating Task 2 or Task 3 attends to its own Task 1 description. One might therefore ask whether the descriptive–operational dissonance we report is a property of the model or an artifact of this shared context: does the joint prompt *create* contradictions that would not otherwise appear? We test this directly.

Setup. For each of the 180 problems and each model, we regenerate Task 1, Task 2, and Task 3 in three *separate* prompts, each conditioned only on the raw regex (Task 3 using the same edge cases as the joint run). We then apply the identical dissonance criterion used for the joint condition (Appendix F): a problem is counted inconsistent if the model's Task 2 or Task 3 behaviour contradicts the constraints stated in its own Task 1 description. Scoring rule, annotators, problem set, edge cases, and decoding (temperature 0) are held fixed across the two conditions, so the only variable is whether the tasks share a context. Because the isolated Task 1 is generated afresh, the reference description is the model's own isolated Task 1, not the joint one; we thus measure cross-context consistency between the model's independent outputs.

Result. Table 9 reports the two counts. Dissonance does not diminish under isolation; for every model it is as high or higher, rising from 37 to 52 for GPT-5.2, from 24 to 29 for Gemini-2.5, from 93 to 126 for Qwen-1.5B, and by smaller margins for the remaining models. The joint context therefore does not manufacture contradictions. If anything it suppresses them: when the model can see its own Task 1 output, it is somewhat more likely to remain consistent with it, so the joint dissonance counts reported in the main text are a conservative floor rather than an inflated figure.

Interpretation. Two effects contribute to the increase, and our conclusion holds under either. The first is the suppression just described: visible context gives the model an explicit description to stay faithful to. The second is that three independent generations have more freedom to diverge than a single coherent one, so some of the rise reflects sampling variation across calls rather than a change in the model's competence. We do not attempt to separate these, because both are consistent with the claim the control is meant to establish: the dissonance is not an artifact of joint prompting. Under the setting that removes all cross-task attention, self-contradiction persists and increases. We conclude that descriptive–operational dissonance is a property of the models rather than of the protocol's prompt structure.

G Additional Results

This appendix presents supplementary qualitative and quantitative material that complements the understanding, construction, and mitigation analyses. 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,

Table 9: Descriptive–operational dissonance under the joint single-prompt protocol versus an isolated-context control, out of 180 problems, scored by the same criterion in both conditions. Dissonance is as high or higher under isolation for every model, so the joint counts are a conservative floor.

Model	Joint	Isolated	Δ
GPT-5.2	37	52	+15
Grok-4.1	4	5	+1
Gemini-2.5	24	29	+5
Qwen-1.5B	93	126	+33
Qwen-7B	74	81	+7
Qwen-14B	48	52	+4

- zero-shot and few-shot baseline results omitted from the main paper for brevity (Appendix G).

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 behaviours in detail.

G.1 Additional Failure Mode Examples

Except where noted, the transcripts below are outputs of frontier models (GPT-5.2 and Gemini-2.5); the shared example in Appendix G is from Qwen 2.5-7B. That these failures appear in the strongest systems evaluated underscores that they are not an artifact of limited model capacity.

The understanding evaluation 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 model's 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.

G.1.1 Anchor Hallucination

Model: Qwen 2.5-7B-Instruct, temperature 0.0, Cross-Consistency prompt (Appendix D).

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.

G.1.2 NULLABILITY NEGLECT

Model: GPT-5.2, temperature 0.0, Cross-Consistency prompt (Appendix D).

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 three 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 three consecutive `a`'s after the initial prefix, 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.

G.1.3 Atomic Unit Blindness

Model: Qwen 2.5-7B-Instruct, temperature 0.0, Cross-Consistency prompt (Appendix D). This transcript is the same model output analysed in Appendix G (Anchor Hallucination); we re-examine it here for a co-occurring failure mode, consistent with the non-exclusivity of modes noted in Table 1.

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 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`, `baabababba`, `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 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*.

G.1.4 Scope and Nesting Confusion

Model: Gemini-2.5, temperature 0.0, Cross-Consistency prompt (Appendix D).

Regex.

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

Note that the leading factor $(a \mid b \mid ab)^*$ denotes Σ^*, so $L(R)$ is exactly the set of strings that end in b or end in aa. The structured tail $(bb^*a \mid aaa \mid \epsilon)(ba)^*b$ imposes no constraint beyond its final b, because any material before that b can be absorbed by the prefix star with the tail taking its ϵ alternative.

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 model's Task 1 description is locally accurate: it correctly identifies the two disjuncts and the shape of the structured tail. What it never recognises is where the scope of the prefix star ends and the scope of the tail begins. Because $(a \mid b \mid ab)^*$ generates every string over $\{a, b\}$, the tail's $(bb^*a \mid aaa \mid \epsilon)(ba)^*$ portion is optional in effect as well as in form; the only surviving constraint is the final symbol. The model instead treats the tail's internal structure as a genuine restriction on the characters preceding the last b, as if that material

were outside the reach of the prefix star. This misplaced scope boundary produces two kinds of error in the output above.

First, it causes systematic under-acceptance. The model rejects `bbb`, `bab`, and `ababb` in Task 2, and `aaababbb` in Task 3, with the justification that "the remaining suffix cannot match either branch." Every one of these strings ends in `b` and is in $L(R)$: the prefix star consumes all but the last symbol and the tail takes its ϵ alternative. The model is applying the tail's constraint to characters that the grammar assigns to the prefix.

Second, when the model does accept a `b`-final string in Task 3, it does so by an illegal parse. For `babbabaab` it asserts that "the ending `aa` followed by `b` is achieved by taking the final alternative," attaching the final `b`, which is scoped inside the left disjunct, to the standalone `aa` of the right disjunct. In effect the model has regrouped

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

as

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

The verdict is correct only by coincidence; the derivation it offers is not one the grammar admits.

Both errors have the same source. The model reasons correctly about each subexpression in isolation but misassigns the boundaries between them, both between the prefix star and the tail, and between the two disjuncts of the tail. This is a direct instance of **Scope and Nesting Confusion**: an internal suffix is treated as globally available outside its parenthesised scope, while material that the outer star legitimately covers is treated as if it were bound by the inner constraint.

G.1.5 Pseudo-Structural Hallucination

Model: GPT-5.2, temperature 0.0, Cross-Consistency prompt (Appendix D).

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:

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

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 $\texttt{ba}\star$. 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.

G.1.6 SIMPLE-PATH BIAS

Model: GPT-5.2, temperature 0.0, Cross-Consistency prompt (Appendix D).

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 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 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 nondeterministic structure into a single, simpler path, ignoring the deeper branch and preferring the most familiar or repetitive option.

G.1.7 COMPLEXITY AVERSION

Model: GPT-5.2, temperature 0.0, Cross-Consistency prompt (Appendix D).

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 $\mathbf{b}$), 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. The same simplification produces the false rejection of ba in Task 2 and the Task 3 claim that "all branches require the string to begin with a": both ignore that a^* in $(a^*a \mid a^*b)$ may be empty, so the second and third branches can begin with b.

G.1.8 TRACE FABRICATION

Model: Gemini-2.5, temperature 0.0, Cross-Consistency prompt (Appendix D).

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 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.

G.1.9 Greedy Parsing Failures

Model: GPT-5.2, temperature 0.0, Cross-Consistency prompt (Appendix D). This transcript is the same model output analysed in Appendix G (Simple-Path Bias); we re-examine it here for a co-occurring failure mode.

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 behaviour 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. The same non-backtracking commitment explains the model's other false rejections in Task 2: b and aab are in $L(R)$ via the trailing b^* with zero blocks, and abba via a single block $b \cdot \epsilon \cdot b \cdot a$, yet in each case the model tries to open a block on the first b and, finding no closing b, rejects.

G.1.10 Indexing and Positional Drift

Model: Gemini-2.5, temperature 0.0, Cross-Consistency prompt (Appendix D).

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. This is the same output analysed as Trace Fabrication in Appendix G; the two modes co-occur here, as Table 1 notes they may. 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.

G.1.11 DESCRIPTIVE–OPERATIONAL DISSONANCE

Model: GPT-5.2, temperature 0.0, Cross-Consistency prompt (Appendix D).

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`$\star$. 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`$\star$.
> `101101001` → REJECTED: Final symbol is `1`; suffix must be `00`$\star$.
> `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

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

G.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 near-zero baseline accuracies are driven overwhelmingly by parse failures rather than by well-formed DFAs that fail language equivalence: every zero-shot output is unparseable (100% PF), so zero-shot accuracy measures format compliance, not construction. In the few-shot setting the non-monotonic 7B-versus-14B pattern is likewise a formatting effect: the 14B model fails to parse on 93.5% of outputs versus 75.0% for 7B, so its lower accuracy (4.0% vs 20.0%) reflects weaker adherence to the terse output schema; because most 14B outputs are unparseable, these baselines are uninformative about its construction ability, which the fine-tuning results (Section 5.1) instead measure. This is consistent with a tension between the capacity to follow format demonstrations and the strength of pretrained generative priors, which grows with scale. The parse-failure flag is a purely structural check (whether the output contains a schema-valid transition table), not a semantic judgment, so it does not require the human-agreement validation applied to the equivalence check in Appendix F.

Table 10: Zero-shot and few-shot baseline accuracy across tiers, with parse-failure rate (PF%: fraction of outputs not parseable into a valid DFA). At these baselines, low accuracy is dominated by parse failures rather than by well-formed DFAs that fail language equivalence: every zero-shot output and the large majority of few-shot outputs are not schema-valid.

Training	Model	Overall	T1	T2	T3	T4	PF%
Zero-shot	1.5B	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%
	7B	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%
	14B	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%
Few-shot (3)	1.5B	0.0%	0.0%	0.0%	0.0%	0.0%	100.0%
	7B	20.0%	52.6%	19.1%	13.7%	2.4%	75.0%
	14B	4.0%	2.6%	6.4%	5.5%	0.0%	93.5%

G.3 Sample-and-Check Baseline

Because DFA correctness is exactly verifiable at inference, a natural strong baseline for any inference-time recovery method is sample-and-check (best-of-n / rejection sampling): draw several independent generations and keep any that the verifier accepts. Our proposed VGNS is itself a verification-guided procedure, so we compare against this baseline to separate the contribution of *steering* from the contribution of simply *attempting more times*. **Protocol.** For each of the 285 Tier 4 mechanistic examples, we sample the CoT-trained 7B model five times independently at temperature 0.8 and check each generation against the exact language-equivalence verifier (Appendix F). An example is counted as recovered if at least one of the five attempts produces a DFA that passes verification. Five attempts is comparable to the number of escalation rounds VGNS uses, so the two methods operate at similar inference-time cost. **Result.** Sample-and-check recovers *no* additional examples beyond the greedy baseline. On every example the model fails greedily, all five sampled attempts also fail: the model produces a different incorrect DFA on each attempt, but no attempt passes verification. By contrast, VGNS recovers 7 of the 285 examples. While this recovery is small and, on the accuracy metric, within the noise floor discussed in Appendix F, the paired comparison against sample-and-check (7 versus 0 on the same examples) isolates the effect of the intervention from that of additional sampling. Whatever benefit VGNS provides therefore arises from the intervention itself, not from additional sampling. **Interpretation.** This result sharpens the central finding of the mechanistic analysis. If the model's Tier 4 failures were stochastic, near a decision boundary and resolvable by chance, then drawing several independent samples would recover a meaningful fraction of them. Instead, although the sampled outputs vary, none is ever correct: the model explores different incorrect constructions but does not reach a correct one within this sampling budget. This is consistent with the finding that Tier 4 outcomes are predictable early in the computation (Section 5.2): the failure is present before generation resolves and does not vary in a way that resampling can exploit. Sampling more attempts does not help, because the failure does not vary in a way resampling can exploit; recovering any of these cases requires an intervention that alters the computation rather than merely repeating it.

Temperature control for the sample-and-check baseline. The sample-and-check comparison above draws five samples per failing example at temperature 0.8. Because Tier 4 targets are long, structured traces, this temperature is high enough to perturb even outputs the model produces correctly: of the 243 examples solved under greedy decoding, 77 became incorrect when resampled at $T = 0.8$, showing that $T = 0.8$ degrades well-formed outputs rather than only exploring alternatives. To ensure the zero-recovery result is not an artifact of an overly high temperature, we repeated the baseline at $T = 0.2$ with sampling enabled: for each of the 42 greedy-failing examples we drew five samples and checked each against the exact verifier (Appendix F). The model still recovers no additional examples — it produces a different incorrect DFA on each of the five low-temperature attempts but never a correct one. We therefore do not claim the correct solution has zero probability under the model, since five samples cannot establish that; rather, it is not recoverable within a modest sampling budget at either temperature, whereas VGNS, which alters the computation rather than resampling it, recovers 7 of the same examples. This isolates the recovery to the intervention, not to additional sampling.

H 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 and Training Methods 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. Our training interventions are also limited in kind: we evaluate parameter-efficient supervised fine-tuning (LoRA with frozen base weights) and curriculum learning, but not full-weight fine-tuning or reinforcement learning with verifiable rewards (RLVR). The latter is a natural fit for this domain, since regex-to-DFA correctness is exactly and cheaply checkable and could serve directly as a reward signal that penalises state-transition violations rather than imitating static traces. Whether either direction closes the residual Tier 4 gap is open: a negative result would more firmly establish the structural account, while a positive one would bound its scope to parameter-efficient supervised adaptation. Our conclusions should therefore be read as applying to the training regimes we evaluate, not as a claim that no optimisation method can remediate these failures.

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 a held-out set of 285 Tier 4 examples. Deep single-model analysis is standard in mechanistic interpretability, where the cost of activation collection, per-neuron scoring, and representation probing makes broad multi-model studies impractical; prior work localizes and edits computation within one or two models (Meng et al., 2022; Wang et al., 2023; Nanda et al., 2023). We follow this practice, focusing on Tier 4 as the tier that most sharply isolates procedural competence and on the 7B model as a representative midpoint of our scaling ladder. While the findings — distributed representation of subset construction, Layer 10 outcome encoding at 87.0% 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 or constraint injection at the decoding level are not evaluated here and may recover additional performance. A further class we do not evaluate is *external counterexample-guided refinement*. Our interventions act only on the model's internal activations and never return information to the model. Because DFA equivalence checking deterministically yields a distinguishing string whenever a predicted automaton is wrong, that string could be fed back to the model in a counterexample-guided or verbal-reflection loop (CEGAR-style refinement or Reflexion (Shinn et al., 2023)), a distinct pathway that supplies external corrective information rather than nudging internal representations. We do not run this loop ourselves. However, Shelat et al. (2026) evaluate exactly such a protocol on this benchmark: their first stage feeds the model concrete counterexamples (false-positive and false-negative strings drawn from the verifier), followed by error-localisation and explicit-correction stages. They report that this guidance corrects shallow errors but that a large fraction of the hardest instances remain unsolved even after all stages (39–54% across the three frontier models on the difficult tier), concluding that feedback does "not reliably enable correction of globally inconsistent or structurally flawed automata." Counterexample-guided refinement is therefore not merely an untested alternative: on this task it has been shown to leave the structural failures we study largely intact, consistent with our finding that these failures are representational rather than resolvable at inference time. 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. Among representation-steering methods, we evaluate ActAdd and SADI but not Contrastive Activation Addition (CAA), which derives a steering direction from contrastive correct/incorrect activation pairs. Because our failing Tier-4 examples do not form a single cluster (Section 5.2), a single contrastive direction faces the same limitation as ActAdd, which likewise applies one shared push; SADI, which we do evaluate, already tailors the intervention per example and recovers at most $+1.0$ pp. We therefore expect CAA to be subject to the same ceiling, though a dedicated evaluation remains a reasonable addition for future work.

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. The eleven failure modes and their ordering by frequency are consistent across all evaluated models, including both frontier and open-weight families. While we cannot directly verify whether the mechanistic findings from Qwen 2.5 transfer to frontier model internals, this cross-family consistency at the

behavioural level is at least suggestive that the underlying difficulty is architecture-independent rather than idiosyncratic to Qwen 2.5.

Evolving Model behaviour. 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.

Circuit-Level Analysis. Our mechanistic analysis tests for localization and repairability at several levels. First, individual neurons, via Activation Difference Scoring over 56,000 neuron–layer pairs. Second, top-scored neuron groups, via joint amplification and suppression of the highest-ranked sets. Third, whole-layer representations, via activation patching, where we replace the model's internal state at each layer, at ranges of layers, and at all layers at once with the state from correctly solved examples. Fourth, targeted steering of those representations (ActAdd, probe-direction, SADI, and our iterative VGNS scheme). None of these repairs the failure: no neuron or small group carries out subset construction, correcting the representation at any layer or combination of layers recovers no failing case beyond a matched control, and steering recovers at most 2.4 percentage points. Layer-wise probing further shows the outcome is linearly decodable well before execution. Together these give causal, not merely correlational, evidence that the computation is not carried by any single neuron or small group, and that the failure cannot be repaired by correcting or steering the representation the model holds. The neuron-level result holds for the sampled population (2,000 randomly sampled neurons per layer, roughly 10% of the 18,944 MLP neurons in each); a construction-specific neuron among the unsampled 90% cannot be excluded, though the uniformity of the result across all 28 layers and 56,000 candidates makes it unlikely that such neurons are common. Our analyses also do not constitute an exhaustive search over all possible neuron subsets, which is combinatorially large, and our patching injects a single averaged correct representation rather than an instance-matched one. An instance-matched counterfactual would require correct/incorrect pairs that are both topology- and token-aligned; in our data the correct-twin and token-aligned sets do not overlap, so we report patching as evidence that shifting toward aggregate correctness does not repair failures and do not claim that no instance-matched intervention could. A distributed circuit spread across many neurons in a pattern our ranking does not surface, or a more targeted patch, therefore cannot be fully ruled out. Our claim is therefore that subset construction is not *localized* in an individually identifiable component, and is not *repairable* by correcting or steering the representation as we do here, not that no circuit for it exists anywhere in the network. Identifying such a circuit directly would require dedicated circuit-discovery methods such as path patching or automated circuit discovery, which we leave to future work.

Coverage of Frontier Reasoning Models. A limitation of our work is that we have not tested new, proprietary, math-oriented models such as Gemini Deep Think (Luong & 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 whether 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. We did not evaluate Claude models because sustained API evaluation at the scale of the 180-problem benchmark was outside our budget. 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 superficial patterns.

Program-Synthesis Baselines. Our evaluation elicits DFAs by direct generation of the transition table (or its full construction trace). An alternative we do not evaluate is program-aided generation

(PAL / Program-of-Thought), in which the model emits executable code and the DFA is obtained by running that code rather than by the model producing the automaton itself. We omit this by design, not merely for scope, because it does not bear on our research question. The object of this study is not whether an LLM-based *system* can produce a correct DFA — a question already answered affirmatively by classical tooling, since Thompson's construction, subset construction, and Hopcroft minimisation are standard algorithms that any correct implementation executes deterministically. Our question is narrower and different: whether the model itself performs the symbolic reasoning, using regex-to-DFA conversion only as an exactly-checkable testbed for that reasoning. A model that writes a Python script invoking these algorithms and obtains a correct DFA by executing it demonstrates competence at *code synthesis* and at recalling which library or procedure to call; it demonstrates nothing about whether the model can itself track NFA state sets, propagate nullability, or normalise residuals — the operations whose failure we document. Indeed, a correct program-aided result would be fully consistent with our central finding, since it would show only that the model can delegate the construction to an external interpreter, not that the construction is carried out in the model's own computation. Program-synthesis evaluation therefore answers a distinct question — can models orchestrate external symbolic tools? — that is complementary to, but disjoint from, the question of intrinsic symbolic reasoning we study here. We leave it to future work.

I 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. For the cross-consistency evaluation, annotators assess model outputs across three tasks of the Cross-Consistency Protocol for semantic correctness and cross-task consistency. For the DFA construction evaluation, 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 about how to apply the rubric to borderline cases were resolved through

adjudication by a senior reviewer for the cross-consistency evaluation, and by discussion among annotators for the construction evaluation. All mitigation 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 clock 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, construction evaluation, mechanistic analysis, steering vector extraction, and DFA correctness verification has been released in the artifact repository and indexed in Appendix J. 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.

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.

J 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/`

J.1 Code Availability

All training code, inference scripts, two-branch prompting drivers, correctness checkers, mechanistic analysis scripts, steering intervention scripts, and data preprocessing utilities used in this work are provided in the artifact repository; this appendix lists each component and its role.

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_ads_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*).
- Activation patching pipeline that replaces the hidden state at a chosen layer, range of layers, or all layers at once with the mean hidden state of correctly solved examples, with a leave-one-out incorrect-source control (*run_activation_patching.py*, *run_multilayer_patching.py*, *merge_patching_results.py*).
- Two-branch parallel zero-shot construction 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.

J.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-weight licenses as specified in the official Qwen 2.5 technical report (Qwen et al., 2025). The diagnostic benchmark from Shelat et al. (2026) is used consistent with its public research release.

Diagnostic Benchmark. 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. 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)

J.3 Model Architectures

All fine-tuning and mechanistic analysis experiments use the Qwen 2.5-Instruct family (Qwen et al., 2025). Full architectural details are provided in Appendix D. Frontier model evaluations (understanding and construction) use GPT-5.2, Grok-4.1, and Gemini-2.5 accessed via their respective APIs at temperature 0.0.

J.4 Training Hyperparameters

All LoRA fine-tuning hyperparameters are documented in full in Appendix D. 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.

J.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.

J.6 Evaluation Methodology

Full evaluation protocols are documented in Appendix F. All mitigation evaluations use programmatic exact match verification with no human annotation. Understanding and construction evaluations combine human annotation with automatic DFA equivalence checking. The DFA exact match verifier uses a bounded-length pre-filter ($|w| \leq 6$) followed by an exact isomorphism test on the minimised automata (Appendix F).

J.7 Mechanistic Analysis Scripts

ADS Neuron Scoring (*compute_ads_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*, *ads_layer_summary.json*.

ADFA Variant Generator (*build_adfa_variants.py*). Generates four example types per positive Tier 4 example. Three are the specificity contrasts used in Section 5.2: V1 (same DFA, symbols renamed), V2 (different regex, DFA of similar structure), and V3 (complex-looking regex producing a trivial 2-state DFA decoy). The generator also emits a fourth type, V4 (the same example with step-level correctness annotations), which was produced for a preliminary verification-feedback design and is *not* used in any experiment reported in this paper. The VGNS results (Section 5.2, Table 2c) use only internal activation interventions—functional antagonism, SADI, and their combination—with escalation gated by the exact-match verifier, and do not consume V4. We retain V4 in the released generator for completeness.

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*.

Activation Patching (*run_activation_patching.py*, *run_multilayer_patching.py*). Registers a prompt-only forward hook on the residual-stream hidden state (3,584-dimensional for the 7B model) at one or more layers. On the prompt forward pass it overwrites the real prompt positions (those with attention mask 1, leaving left-padding untouched) with a source vector, then lets generation proceed without further intervention. The source is the mean hidden state over the correctly solved Tier 4 examples at that layer; the leave-one-out mean over incorrect examples serves as the control. *run_activation_patching.py* sweeps each of the 28 layers individually and runs the control at chosen

layers; *run_multilayer_patching.py* patches arbitrary sets of layers at once (single layers, contiguous windows, or all 28 layers), each layer receiving its own source vector. Correctness is checked with the same exact-match DFA verifier used elsewhere. Generation is batched with left padding for efficiency; a batched-versus-unbatched equivalence check confirms identical results. Outputs *split.json* (baseline correct/incorrect), a per-layer or per-set recovery record, and recovery counts.

J.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.

J.9 Dataset Generator

The full dataset generator script (*generate_dataset.py*) is provided in the released repository. 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.

J.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. The full train_qwen7b_lora_cot.py is provided in the released repository.

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. The full train_lora.py is provided in the released repository.

Curriculum SLURM Scripts. The full submit_curriculum_phasewise.sh is provided in the released repository.

J.11 Mechanistic Analysis Implementation

ADFA Variant Generator. The full build_adfa_variants.py is provided in the released repository.

ADS Neuron Scoring. The full compute_ads_scores.py is provided in the released repository.

Steering Vector Extraction. The full extract_steering_vectors.py is provided in the released repository.

Steering Evaluation and VGNS. The full run_steering_eval.py is provided in the released repository.

Activation Patching (Single-Layer Sweep). The full run_activation_patching.py is provided in the released repository.

Activation Patching (Multi-Layer). The full run_multilayer_patching.py is provided in the released repository.

Patching Result Aggregation. The full merge_patching_results.py is provided in the released repository.

J.12 Construction Experiment Setup

Templates. All construction prompting templates (Intuitive, Derivative) used for frontier models are provided in the released repository.

Experiment Driver. The exact script used to run the construction experiments is provided in the released repository, 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.

The full construction driver script is provided in the released repository.

J.13 Prompts for All Methods

Prompt Templates. The full Intuitive, Derivative-Based, and Cross-Consistency prompt templates are reproduced verbatim in Appendix D; they are also included in the released repository.