Beyond Memorization: Testing LLM Reasoning on Unseen Theory of Computation Tasks

Shlok Shelat
Ahmedabad University
Gujarat, India

Jay Raval
Ahmedabad University
Gujarat, India

Souvik Roy
Ahmedabad University
Gujarat, India

Manas Gaur
University of Maryland
Baltimore County
Baltimore, MD, USA

Abstract

Large language models (LLMs) have demonstrated strong performance on formal language tasks, yet whether this reflects genuine symbolic reasoning or pattern matching on familiar constructions remains unclear. We introduce a benchmark for deterministic finite automata (DFA) construction from regular languages, comprising factual knowledge questions, seen construction problems from public sources, and two types of unseen problems: hand-crafted instances with multiple interacting constraints and systematically generated problems via Arden's theorem. Models achieve perfect accuracy on factual questions and 84-90% on seen tasks. However, accuracy drops sharply on unseen problems (by 30-64%), with failures stemming from systematic misinterpretation of language constraints, incorrect handling of Kleene-star semantics, and a failure to preserve global consistency. We evaluate a three-stage hint protocol that enables correction of shallow errors but does not reliably resolve globally inconsistent or structurally flawed automata. Our analysis across multiple prompting strategies (direct, Chain-of-Thought, Tree-of-Thought) reveals that errors persist regardless of prompting approach, exposing a fundamental gap between LLMs' ability to generate syntactically plausible DFAs and their capacity for semantically correct formal reasoning.

1 Introduction

Large language models (LLMs) have demonstrated remarkable performance on diverse reasoning benchmarks, from mathematical problem-solving (Lewkowycz et al., 2022; Welleck et al., 2021; Azerbayev et al., 2023) to code generation (Wu et al., 2022). However, a fundamental question remains unresolved: do these models perform genuine symbolic reasoning, or do they primarily rely on pattern matching over memorized examples? Recent work reveals persistent failures on tasks requiring structured symbolic manipulation (Katz et al., 2025; Yue et al., 2024), suggesting that strong benchmark performance may not reflect robust reasoning capabilities.

We address this question through the lens of *deterministic finite automata (DFA) construction from regular languages*, a core problem in the Theory of Computation (ToC). Moreover, this also depicts the task of lexical analyser which are solved by using tools like Lex, Flex with limitations (Aho et al., 2006). This task offers unique advantages as a reasoning probe: (1) correctness is formally verifiable through exhaustive testing, (2) solutions require multi-step symbolic manipulation with global consistency constraints, (3) the space of possible problem instances is combinatorially vast, and (4) DFAs represent the simplest non-trivial computational model, ensuring that failures cannot be attributed to problem complexity or ambiguous specifications. We focus on prompting-based evaluation (rather than fine-tuning) to reflect practical LLM usage. Critically, while existing ToC benchmarks (Golesteanu et al., 2024; Zahraei and Asgari, 2024) evaluate factual knowledge and proof verification, they do not systematically control for *memorization versus compositional generalization*, a model may succeed by recalling similar problems from training data rather than reasoning from first principles.

To isolate genuine reasoning capability, we introduce a carefully designed benchmark with three components: (1) a *knowledge-checking dataset* assessing foundational understanding of DFA definitions and properties, (2) a *seen construction dataset* comprising 90 publicly available DFA problems, and (3) an *unseen construction dataset* with 180 novel problems. The knowledge component is intended to establish that models possess the basic conceptual prerequisites for DFA construction, allowing subsequent performance differences to be attributed to generalization rather

Findings of the Association for Computational Linguistics: ACL 2026, pages 29904–29934
July 2-7, 2026

than missing foundational knowledge. The unseen dataset is generated via two complementary approaches. The first approach, *mathematical art*, manually constructs problems with multiple interacting constraints, forbidden substrings, and narrative-based specifications (e.g., encoding chess moves). The second approach, *mathematical engineering*, systematically generates problems via Arden's theorem (Sipser, 2013), producing structurally complex regular expressions unlikely to appear in training data. This seen/unseen split enables us to measure the performance gap attributable to memorization versus reasoning.

We evaluate frontier LLMs: GPT-5.1, Gemini-2.5-Flash, and Grok-4.1-fast-reasoning – across multiple prompting strategies including Chain-of-Thought (CoT) and Tree-of-Thought (ToT) (Wang and Zhou, 2024; Yao et al., 2023). We also introduce a *three-stage hint protocol* that progressively reveals construction errors, enabling us to assess whether LLMs can self-correct when guided.

Main Findings. Our results reveal a stark dissociation between knowledge and reasoning: all models achieve 100% accuracy on factual questions and 84–90% on seen construction tasks, but accuracy drops sharply on unseen problems (20.67–59.12% under direct prompting, representing 30–64 % point drops). Detailed error analysis shows systematic failure modes: incorrect simplification of Kleene star semantics, failure to preserve constraints under concatenation, and introduction of spurious states. Critically, these failures persist across all prompting strategies, and the hint protocol primarily corrects shallow errors while leaving globally inconsistent automata uncorrected.

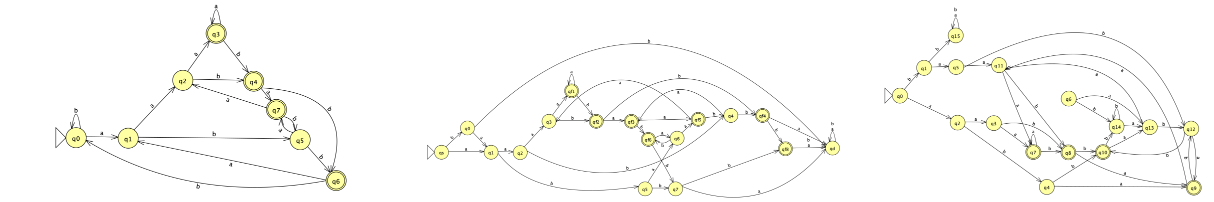
Contributions. This work makes the following contributions: (i) **Novel benchmark:** We introduce the first DFA construction benchmark with systematic seen/unseen splits, comprising 50 knowledge, 90 seen, and 180 unseen problems (60 hand-crafted, 120 via Arden's theorem). (ii) **Controlled memorization study:** By evaluating structurally similar seen/unseen pairs, we provide the first evidence that LLM success on ToC tasks primarily reflects memorization rather than compositional reasoning. (iii) **Comprehensive prompting evaluation:** We evaluate CoT, ToT (with four construction methods: direct, minimization, derivative-based, Thompson's algorithm), and a novel hint-based self-correction protocol. (iv) **Systematic failure taxonomy:** Through detailed analysis of 500+ incorrect DFAs, we identify six recurring failure modes (derivative normalization errors, constraint composition failures, etc.) that reveal fundamental limitations in symbolic state tracking. All datasets, prompts, evaluation code, and model outputs are released at `https://anonymous.4open.science/r/dfa-llm-evaluation-B82D/` to support reproducibility and future research on formal reasoning in LLMs.

2 Seen Dataset and Task Formulation

2.1 Task Definition

We evaluate LLMs on the task of constructing deterministic finite automata (DFAs) from formal language specifications. Given a target language L specified either as a regular expression (RE) or natural language description, the model must produce a DFA $D = \langle Q, \Sigma, \delta, q_0, F \rangle$ that recognizes exactly L. Here, Q is a finite set of states, Σ is the input alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of accepting states (Sipser, 2013). A DFA D accepts string $w = w_1 w_2 \cdots w_n \in \Sigma^*$ if there exists a sequence of states $s_0, s_1, \ldots, s_n$ such that (i) $s_0 = q_0$, (ii) $s_i = \delta(s_{i-1}, w_i)$ for all $1 \le i \le n$, and (iii) $s_n \in F$. We say D recognizes language L if $L = \{w \in \Sigma^* \mid D \text{ accepts } w\}$. A language is *regular* if there exists a DFA that recognizes it. Correct DFA construction requires models to: (1) parse and interpret formal language specifications, (2) identify the minimal state structure capturing all constraints, (3) design transitions ensuring acceptance of all and only valid strings, and (4) maintain global consistency across all states and symbols. Critically, there exist exponentially many invalid DFAs for any language, and small errors in state semantics or transition assignments can invalidate the entire construction.

Problem Instances. Each problem specifies the target language in one of two formats: *(a) Natural language:* L_1 = {Construct a DFA over $\{a, b\}$ that accepts all strings in which the third-to-last symbol from the end must be 'a'}. Here, if the problem format is given in natural language, it is not possible to design DFA using Lex, Flex (Aho et al., 2006). *(b) RE:* $L_1 = (a+b)^* a(a+b)(a+b)$. Both formats specify the same language (Figure 1(a)). Models must output a complete DFA specification including states, transitions, start state, and accepting states. We discuss minimal DFAs (fewest



(a) Correct minimal DFA for L_1 (b) Correct minimal DFA for L_2 (c) Gemini-2.5 output for L_2

Figure 1: (a): Minimum DFA for language L_1; (b) & (c): Comparison of correct minimal DFA for unseen language L_2 (see b) and representative incorrect output (see c). Notation: ◯ = state, ▷ = start state, ◎ = accepting state. Transition $q_0 \xrightarrow{a} q_1$ indicates that on input a from state q_0, the DFA transitions to state q_1(Please zoom for better readability).

states) as identifying state equivalence reflects an additional layer of symbolic reasoning. However, minimality is not part of the grading criterion: evaluation is based solely on semantic correctness (i.e., language equivalence), as requiring minimality would introduce an objective orthogonal to our primary focus on correct DFA construction.

2.2 Knowledge and Seen DFA Dataset

Before evaluating construction ability, we verify whether models possess the foundational knowledge required for automata reasoning. This dataset comprises 50 questions: 25 multiple-choice and 25 short-answer questions covering DFA definitions, regular expression semantics, and the relationship between DFAs and nondeterministic FAs (NFAs). The questions are drawn from standard ToC textbooks and publicly available materials, and span a range of conceptual, application-level, and multi-step reasoning difficulty aligned with the requirements of DFA construction tasks (see Appendix B). **Results:** All models achieve 100% accuracy (Table 1). This indicates that models possess the basic prerequisites for DFA construction; thus, subsequent errors are unlikely to stem from missing foundational knowledge.

The seen dataset comprises 90 DFA construction problems collected from online university problem sets, textbooks, and publicly accessible ToC resources. For each problem, we verified that both the problem statement and solution DFA are publicly available online, ensuring these represent patterns likely encountered during pretraining.

Language Characteristics. The seen dataset includes standard patterns such as: (i) Suffix/prefix constraints (e.g., "ends with ab"); (ii) Counting modulo k (e.g., "even number of a's"); (iii) Position-based constraints (e.g., "3rd-last symbol a"); and (iv) Boolean combinations (e.g., "contains aa or bb").

Dataset	Prompting	GPT-5.1	Grok-4.1	Gemini-2.5
Knowledge	Zero-Shot	100%	100%	100%
Seen DFA	Zero-Shot	84.2%	89.5%	85.0%

Table 1: Results of knowledge and seen DFA datasets.

Evaluation Protocol. We evaluate models using zero-shot direct prompting (no worked examples). Models are queried via official APIs with temperature set to 0 for deterministic decoding. Each model receives the problem specification and must output a complete DFA in structured JSON format specifying states, alphabet, transitions, start state, and accepting states (Appendix D).

Results. Table 1 shows that all models achieve strong performance: Grok-4.1-fast-reasoning achieves the highest success rate (89.5%), followed by Gemini-2.5-Flash (85.0%) and GPT-5.1 (84.2%). For the example language L_1, all three models produce the correct minimal DFA. These results suggest that models can successfully construct DFAs for *familiar* problem patterns.

The Memorization Question. High accuracy on seen problems does not imply genuine reasoning capability. Models may succeed by retrieving similar examples from training data rather than performing compositional symbolic manipulation. To isolate reasoning from memorization, we next introduce the unseen construction dataset with carefully controlled novelty.

2.3 The Memorization Question

Strong performance on the seen dataset (84–90% accuracy) might suggest that LLMs possess robust DFA construction capabilities. However, this conclusion is premature: models may succeed by retrieving memorized solution patterns rather than performing genuine symbolic reasoning. To distinguish these explanations, we must evaluate performance on *structurally novel* problems that require compositional generalization. **Motivating Example:** Consider two closely related languages: L_1 **(seen):** Accepts all strings over $\{a, b\}$ where the third-to-last symbol is 'a'; L_2 **(unseen):** Accepts all strings over $\{a, b\}$ where (i) the fourth-to-last symbol is 'a', *and* (ii) substring 'bb' does not appear before any 'a'. Language L_2 extends L_1 with one additional constraint (forbidding 'bb' before 'a') and a minor modification to the position constraint (4th-last instead of 3rd-last). Both languages require similar reasoning – tracking symbol positions while enforcing ordering constraints – yet L_2 is constructed to be absent from public problem sets and textbook solutions.

Empirical Evidence. Despite achieving 100% accuracy on L_1, *all models fail on* L_2 *under direct prompting*. Figure 1(c) shows Gemini-2.5-Flash's output: while the constructed DFA accepts some valid strings (e.g., 'aaaa', 'aaab'), it also accepts invalid strings such as 'aaa' (violates position constraint), and 'aabbaba' (violates both constraints). Similar systematic errors occur for GPT-5.1 and Grok-4.1-fast-reasoning (Appendix A). This failure is particularly revealing because L_1 and L_2 differ only in *constraint composition*, not in fundamental reasoning requirements. The models correctly construct DFAs for simpler positional constraints (seen in training) but fail when these constraints are combined with ordering restrictions – suggesting success on seen tasks reflects pattern retrieval rather than robust symbolic reasoning.

Research Question. These observations motivate our central research question: *To what extent does LLM performance on formal reasoning tasks depend on memorization of training examples versus compositional symbolic reasoning?* Answering this question requires systematic evaluation on unseen problems that control for structural novelty while preserving task and reasoning requirements.

Unseen Dataset Design. To enable this controlled evaluation, we construct an unseen DFA dataset using two complementary approaches: (i) **Mathematical Art (60 problems):** Manually designed problems with multiple interacting constraints, forbidden substrings, positional patterns, and narrative-based specifications (e.g., encoding chess openings, maze navigation). These problems require creative constraint combination absent from standard curricula. (ii) **Mathematical Engineering (120 problems):** Systematically generated problems via Arden's theorem (Sipser, 2013). We construct random NFAs, derive their accepted languages through algebraic elimination, and use the resulting regular expressions (often highly nested and non-standard) as problem specifications. This approach ensures structural diversity and scalability.

All unseen problems are manually verified to be absent from public problem repositories, textbooks, and online course materials. We categorize problems by difficulty (easy, medium, hard) based on the number of constraints (Art) or nesting depth (Engineering), enabling fine-grained analysis of where reasoning breaks down. Detailed construction procedures and representative examples are provided in Section 3.

Evaluation Framework. Beyond measuring accuracy on unseen problems, we evaluate multiple prompting strategies (CoT, ToT with four construction branches) and introduce a three-stage hint protocol to assess self-correction capability. This comprehensive evaluation isolates whether failures stem from initial misinterpretation, inability to maintain symbolic consistency, or fundamental reasoning deficits that persist even with corrective guidance.

3 Unseen DFA Construction Dataset

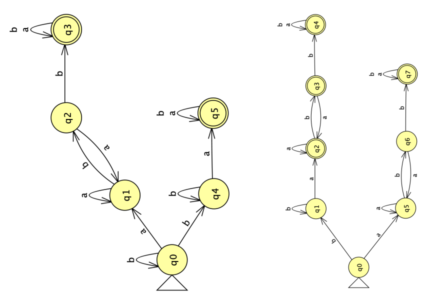
We employ two generation strategies: *manual constraint composition* (60 problems) for creative problem design, and *systematic Arden's theorem inversion* (120 problems) for scalable generation with guaranteed structural diversity.

3.1 Manual Constraint Composition

This approach extends seen DFA patterns through controlled increases in constraint complexity. Following the design principles illustrated by the $L_1 \rightarrow L_2$ transformation (Section 2.3), we systematically combine multiple constraints that rarely co-occur in standard curricula: (i) **Product constructions:** Multiple conjunctive/disjunctive constraints requiring state-space products (e.g., "strings starting with 'aba' OR 'bab' AND ending with their reverse"). (ii) **Interacting constraints:** Independent conditions that cannot be verified locally (e.g., "fourth-to-last symbol is 'a' AND substring 'bb' never precedes 'a'" – language L_2). (iii) **Structural restrictions:** Positional patterns and forbidden substrings (e.g., "XOR of first three bits equals final bit"; "every 4-bit window's product is divisible by 4"). (iv) **Narrative encodings:** Real-world scenarios requiring formalization (e.g., chess opening sequences encoded as moves; fruit-mixing recipes as syrup combinations; see Appendix B for additional examples). All 60 problems were manually verified to be absent from the top 100 Google search results, standard textbooks (Sipser, 2013; Hopcroft et al., 2001), and popular online repositories (`GitHub`, `StackOverflow`, course websites). This approach yields high-quality problems testing creative constraint integration, but scalability is limited by human effort.

3.2 Generation via Arden's Theorem

To address scalability while ensuring structural novelty, we employ a reverse-engineering approach: generate random NFAs, derive their accepted languages algebraically using Arden's theorem, and use the resulting regular expressions as problem specifications. This systematic procedure guarantees that each generated problem has a unique structure determined by random NFA topology rather than memorized patterns. Following is the generation procedure: (i) **Random NFA construction:** Random NFA with $5-8$ states, $2-3$ accepting states, and random transition density $0.3-0.6$. The nondeterministic transition function $\delta : Q \times \Sigma \to 2^Q$ permits multiple successor states per input symbol. (ii) **Language derivation via Arden's theorem:** Apply Arden's theorem (Sipser, 2013) – if regular expressions P and Q satisfy $R = Q + RP$ and P does not contain ϵ, then $R = QP^*$ – to eliminate states iteratively and derive a closed-form RE. (iii) **Minimal DFA construction:** Convert the NFA to a DFA via subset construction to obtain the ground-truth solution.



(a) (b)

Figure 2: **(a):** Random NFA over $\{a, b\}$. **(b):** Minimal DFA recognizing the derived language L_3.

Detailed Example: Deriving Language L_3. We now demonstrate the complete derivation process using the random NFA shown in Figure 2(a).

Step 1: State equations from NFA. Each state's language is expressed recursively based on incoming transitions of Figure 2(a):

$$q_5 = q_5a + q_5b + q_4a \tag{1}$$
$$q_4 = q_0b + q_4b \tag{2}$$
$$q_3 = q_3a + q_3b + q_2b \tag{3}$$
$$q_2 = q_1b \tag{4}$$
$$q_1 = q_0a + q_1a + q_2a \tag{5}$$
$$q_0 = \epsilon + q_0b \tag{6}$$

Here q_0 is the start state (accepting ϵ), and q_3, q_5 are accepting states. Our goal is to derive closed-form REs for q_3 and q_5 in terms of the input alphabet $\{a, b\}$ only.

Step 2: Solve for q_0 using Arden's theorem. Equation (6) has the form $R = Q + RP$ with $R = q_0$, $Q = \epsilon$, and $P = b$. Since b does not contain ϵ, Arden's theorem yields: $q_0 = \epsilon \cdot b^* = b^*$

Step 3: Solve for q_1. Substitute $q_2 = q_1b$ from Equation (4) into Equation (5):

$$\begin{aligned} q_1 &= q_0a + q_1a + q_2a = q_0a + q_1a + (q_1b)a \\ &= q_0a + q_1(a + ba) \end{aligned}$$

Applying Arden's theorem with $R = q_1, Q = q_0a$, $P = (a + ba)$: $q_1 = q_0a(a + ba)^* = b^*a(a + ba)^*$.

Following a similar application of Arden's theorem, we get the following REs for the final states: $q_3 = b^*a(a+ba)^*bb(a+b)^*$ and $q_5 = b^*bb^*a(a+b)^*$

Final language: Therefore, the language recognized by the NFA is: $L_3 = q_3 \cup q_5$. Hence,

$$L_3 = b^*a(a + ba)^*bb(a + b)^* + b^*bb^*a(a + b)^*$$

The resulting expression exhibits deep nesting (concatenations of unions containing Kleene stars) and non-standard structure unlikely to match textbook examples. Figure 2(b) shows the minimal DFA for L_3, obtained by converting the NFA via subset construction and state minimization. Using this procedure, we generated 120 problems spanning a range of structural complexities. Each random NFA seed produces a distinct RE, ensuring comprehensive coverage of expression patterns while maintaining verifiable correctness through algorithmic DFA construction[1].

[1]All datasets, file formats, and additional implementation details are provided in Appendix B.

Dataset Component	Count	Easy	Med	Hard
Unseen construction	**180**	**41**	**65**	**74**
Manual composition	60	15	22	23
Arden generation	120	26	43	51

Table 2: Dataset statistics and difficulty categorization.

3.3 Dataset Statistics and Difficulty level

We categorize all 180 unseen problems by difficulty based on structural complexity: (i) **Easy (**41 **problems,** 22.8**%):** Simple expressions with shallow nesting (e.g., $L_4 = b(a+b)^*ab$; 1 Kleene star, 1 union, 3 concatenations, minimal DFA has ≤ 5 states). (ii) **Medium (**65 **problems,** 36.1**%):** Moderate compositionality with $2-4$ nested operators (e.g., $L_5 = (a^*b + ((a^*a + a^*b)b^*)a + (((a^*a + a^*b)b^*a)b^*)b)a^*$; minimal DFA has $6-12$ states). (iii) **Hard (**74 **problems,** 41.1**%):** Deep nesting requiring global consistency tracking (e.g., $L_6 = (((a^*b)(a + b((\epsilon + a)(ba)^*(a + bb) + b)b^*a)^*)b((\epsilon + a)(ba)^*(a + bb) + b))b^*$; minimal DFA has ≥ 13 states). For manually composed problems, difficulty reflects the number of interacting constraints ($1-2$ for easy, $3-4$ for medium, $5+$ for hard). For Arden-generated problems, difficulty corresponds to RE nesting depth and the number of states in the DFA. Table 2 summarizes the complete benchmark. The difficulty distribution ensures comprehensive evaluation across complexity levels, with a slight bias toward hard problems (41%) to stress-test reasoning capabilities, for detailed classification see Appendix C.

Validation of Novelty. To ensure problems are truly unseen, we: (1) manually searched for exact and structurally similar matches in top Google results and online repositories, (2) verified that Arden-generated regular expressions do not match patterns in standard textbooks, and (3) confirmed that manual problems combine constraints in non-standard ways. While we cannot guarantee complete absence from all pretraining corpora, the controlled generation process and verification steps provide strong evidence of novelty.

4 Evaluation Methodology

Here, all models are accessed via official APIs with temperature set to 0 for deterministic decoding, and all experiments use identical prompt templates to ensure fair comparison [2]. We evaluate five prompting configurations spanning zero-shot, explicit reasoning, multi-branch exploration, and guided self-correction:

Direct Input-Output (Zero-Shot). Models receive only the problem specification (RE or natural language) and must output a complete DFA with no intermediate reasoning or worked examples (Kojima et al., 2022). This baseline isolates pure construction capability without scaffolding.

Chain-of-Thought (CoT). Models are instructed to reason step-by-step before producing the final DFA (Wang and Zhou, 2024). For DFA construction, this naturally decomposes into: (1) interpreting the language specification, (2) identifying required states and their semantic roles, and (3) designing transitions ensuring correct acceptance/rejection. For example, given $L_7 = (a + b)^*a(a+b)^*$, CoT prompting encourages models to explicitly reason: "State q_0 tracks strings without 'a'; state q_1 (accepting) tracks strings with at least one 'a'; transition $\delta(q_0, a) = q_1$ handles first 'a'; etc.". **Chain-of-Thought (One-Shot).** Extends CoT by providing a single worked example (problem + solution + reasoning trace) before the target problem. This tests whether explicit demonstration reduces ambiguity in reasoning structure.

Tree-of-Thought (ToT). Decomposes DFA construction into four distinct reasoning branches corresponding to standard automata-theoretic methods (Yao et al., 2023). Models are prompted to explore multiple construction approaches in parallel: (i) **Direct (Intuitive):** Interpret the regular expression semantically, identify necessary states based on language properties, and design transitions directly. Tests high-level symbolic reasoning without algorithmic scaffolding. (ii) **Minimization-Based:** Construct an initial DFA (possibly with redundant states), then merge equivalent states via partition refinement. Tests whether models understand state equivalence beyond local transition correctness. (iii) **Derivative-Based:** Apply Brzozowski's method (Sipser, 2013), where each state corresponds to a distinct derivative of the regular expression with respect to input prefixes. Tests symbolic algebraic manipulation and normalization of equivalent expressions. (iv) **Thompson's Construction:** Follow the algorithmic pipeline: convert regular expression to ϵ-NFA via structural recursion, then determinize via subset construction. Tests procedural correctness on multi-stage formal algorithms. Lexical analyser tools Lex, Flex follow this approach to automatically cre-

[2]Additional details regarding determinism controls, decoding settings, and the runtime environment are provided in Appendices I and J.

ate DFA from RE with limitations (Aho et al., 2006). These branches span fundamentally different reasoning styles (semantic interpretation, equivalence optimization, algebraic manipulation, algorithmic execution), enabling fine-grained diagnosis of where models succeed or fail. All prompt templates are provided in Appendix D.

Hint-Based Self-Correction Protocol. For problems answered incorrectly under direct prompting, we evaluate whether models can self-correct when provided structured feedback. We introduce a three-stage hint protocol with increasing levels of guidance. Hints are manually constructed following a fixed rubric and are validated by independent annotators in a model-blind setting to ensure consistency across instances and models. **Stage 1: Counterexample Feedback.** The model receives concrete counterexamples exposing errors in its DFA: **False negatives:** strings in L that the DFA rejects; **False positives:** strings not in L that the DFA accepts. Counterexamples are obtained from the evaluation pipeline (exhaustive testing up to length 6 with additional randomized sampling described in the validation and grading procedure) and three structurally distinct examples are selected to cover different behavioral paths of the automaton. **Stage 2: Error Localization.** If errors persist, the model is informed which reasoning stage is incorrect (language interpretation, state design, or transition) along with a high-level description of all identified error types, without disclosing exact transitions or states. **Stage 3: Explicit Error Disclosure.** If the model still fails, it receives an explicit description of all identified errors, including the exact faulty transitions or acceptance conditions (e.g., 'Missing transition: $\delta(q_2, a) \rightarrow q_4$ to handle aba'). At each stage, the model is provided with its previous DFA output to frame the task as iterative correction. Evaluation proceeds sequentially: Stage 1 is applied to all instances that fail direct prompting, and each subsequent stage is evaluated only on instances that remain unsolved from the previous stage. All stages use deterministic decoding (temperature $= 0.0$) with a maximum token budget of 4000, and up to three retries are allowed to handle occasional API-format failures. This progressive disclosure (counterexamples $\rightarrow$ high-level localization $\rightarrow$ explicit correction) standardizes the level of guidance across error types while enabling controlled analysis of self-correction behavior. While manual hint construction ensures consistency and diagnostic quality, it is not scalable; automated hint generation remains an important direction for future work. Appendix D provides representative templates.

Validation and Grading. All DFA outputs are validated using a two-stage pipeline designed to balance computational efficiency with exact correctness guarantees: (i) **Automated Validation:** A custom validator checks (a) *syntactic correctness* (valid JSON schema and well-formed transitions), (b) *totality* (exactly one outgoing transition per symbol per state), and (c) *behavioural equivalence* via finite-horizon testing. Behavioural equivalence is evaluated by exhaustive enumeration of all strings up to length 6, combined with random sampling of 2000 strings of length 7–15. If a counterexample is found, the DFA is marked *incorrect*. Otherwise, it is marked *likely correct* and passed to the second stage. In practice, incorrect DFAs almost always admit short distinguishing strings, making this stage an effective first-pass filter. (ii) **Exact Equivalence Checking:** All *likely correct* DFAs are validated using exact equivalence checking against the ground-truth DFA. Specifically, we minimize the model-generated DFA and compare it with the minimized ground-truth DFA; equivalently, this can be viewed as checking emptiness of the symmetric difference. A DFA is marked correct if and only if this check succeeds. The automated and exact evaluations differ in only 9 cases, where subtle structural errors produce distinguishing strings beyond the finite search horizon. These cases represent a small fraction of the total evaluations, confirming that the finite-horizon validator is highly reliable as a first-stage filter. Final accuracy is computed based on exact equivalence outcomes. For interpretability, a subset of outputs is manually inspected by independent reviewers in a model-blind setting. All evaluation is conducted in a model-blind setting to prevent bias (see Appendices E and F for implementation details).

5 Results and Analysis

Results. Table 3 reports DFA construction success rates across datasets, prompting strategies, and models. (i) **Seen vs. Unseen:** In contrast with the results of seen dataset (Section 2.2), performance drops sharply on the *Unseen DFA Construction* dataset across all models. Under direct prompting, Grok-4.1-fast-reasoning performs

Dataset	Prompting	GPT-5.1	Grok-4.1	Gemini-2.5
Seen DFA	Zero-Shot	84.2%	89.5%	85.0%
Unseen DFA	Zero-Shot	20.67%	59.12%	29.33%
	CoT	16.67%	51.90%	23.33%
	CoT (One-Shot)	19.33%	55.87%	28.10%
	ToT	24.10%	–	54.00%

Table 3: DFA construction success rates across datasets, prompting strategies, and models; "–" denotes timeouts.

best, followed by Gemini-2.5-Flash and GPT-5.1. This corresponds to a degradation of 30–64 % relative to seen tasks. Crucially, the task formulation is identical across seen and unseen settings. The observed performance gap therefore isolates a failure of *compositional generalization* rather than task understanding. (ii) **CoT Effects:** CoT consistently degrades performance on unseen DFA construction across all models. Here, explicit step-by-step reasoning increases effective state-space complexity and amplifies the impact of early semantic errors. To assess whether examples mitigate this failure mode, we evaluate *CoT (One-Shot)* prompting. Providing a single worked example improves performance relative to standard CoT. (iii) **ToT Effects:** ToT yields the strongest performance on complex unseen instances. Gemini-2.5, GPT-5.1 improve by 24.67, 3.43 % over direct prompting and 30.67, 7.43 % over CoT prompting respectively. In contrast, Grok-4.1-fast-reasoning frequently fails to return an output within the fixed inference-time budget [3].

Analysis. Across models and prompting strategies, the dominant failure modes arise from difficulties in maintaining *globally consistent symbolic structure*, rather than from a lack of formal knowledge. To illustrate, we analyze following representative outputs under the four ToT framework methods (which subsumes Direct and CoT). (i) **Thompson Construction:** Here, LLMs generally succeed on moderately complex REs with limited nesting because this method is fully algorithmic. However, for complex expressions involving deep nesting, the intermediate ε-NFA grows rapidly in size. This *state explosion* acts as a reason behind the failure of LLMs. Importantly, these failures do not reflect a misunderstanding of method, but rather a limitation in reliably managing large, densely connected intermediate structures. (ii) **Direct Construction:** Here, LLMs attempt to construct DFAs by informally reasoning about the language constraints. This setting exhibits following recurring failure modes. *(a) Partial or oversimplified language interpretation.* To produce a correct DFA, the first most logically important step demands correct understanding about the language. However, there are many evidences where LLMs lack in this step (e.g., for $L_8 = a^*(a+b)(a+b)^*a$, LLM develops wrong understanding 'ending with a', ignoring presence of $(a+b)$). The same is also applicable for 'Kleene star'. *(b) Inconsistency between a model's stated understanding and its final construction.* (e.g., starts with correct understanding 'strings ending with ab', but returns answer for 'ending with ba'). It indicates difficulty in maintaining symbolic invariants across construction. *(c) Failure to preserve constraints under concatenation.* (e.g., fails to handle L_8, but capable to handle unit components, namely a^*, $(a+b)$, $(a+b)^*$, and a). *(d) Over-acceptance of strings outside the target language.* Models often validate their constructions using a finite set of accepted strings while ignoring rejection cases. In addition, LLMs often introduce unreachable or redundant states, which further invalidate the answer [4]. (iii) **Minimization-Based Construction.** Here, most errors originate before minimization is applied and propagates during the algorithmic minimization approach. (iv) **Derivative-Based Construction.** LLMs frequently fail to normalize semantically equivalent RE derivatives, leading to the creation of spurious DFA states. (e.g., for $L_9 = b(a+b)^*ab$[5] the derivative $(a+b)^*ab + \varepsilon$ is a correct intermediate step reflecting nullability. However, LLMs frequently replace ε with $(a+b)^*$, yielding the incorrect intermediate RE $(a+b)^*ab + (a+b)^*$. This RE normalizes to $(a+b)^*$, thus discards the original language constraints). Importantly, this failure shows an inability to perform basic algebraic simplification and language containment reasoning.

Hint-Based Protocol Analysis. The hint-based protocol is designed to evaluate whether LLMs can recover from incorrect DFA constructions when provided with structured guidance. Table 4 summarizes model performance across difficulty levels and hint stages. (i) **Easy:** Most errors are resolved after the *first hint* across all models, indicating that failures at this level primarily arise from superficial misinterpretations rather than fun-

[3] A detailed analysis of timeout behaviour, inference cost, and deployability trade-offs is provided in Section 8.

[4] Appendix G contains a gallery of representative errors.

[5] see Appendix H for model-generated output.

Stage	**GPT-5.1**			**Grok-4.1**			**Gemini-2.5**		
	Easy	Medium	Difficult	Easy	Medium	Difficult	Easy	Medium	Difficult
First Hint	60.13%	10.13%	30.76%	85.25%	39.20%	52.93%	57.45%	49.13%	31.79%
Second Hint	15.50%	16.10%	15.38%	14.75%	27.67%	7.83%	42.55%	12.47%	6.88%
Final Hint	24.37%	36.17%	0%	0%	13.33%	0%	0%	12.37%	16.31%
Not Solved	0%	37.60%	53.86%	0%	19.80%	39.24%	0%	26.03%	45.02%

Table 4: Hint-based DFA construction performance.

damental reasoning errors. (ii) **Medium:** Here, recovery is distributed across multiple hint stages rather than concentrated at a single level of guidance. Moreover, a non-trivial fraction remain incorrect even after the final hint. (iii) **Difficult:** A large proportion of problems remain unsolved even after all hints are provided, particularly for GPT-5.1 and Gemini-2.5-Flash. The persistence of unsolved cases suggests that, for complex REs, guidance is insufficient to overcome underlying limitations in symbolic state-transition tracking, and global automaton structure. Overall, the hint-based analysis demonstrates that while all models can leverage guidance to correct shallow errors, their ability to recover degrades rapidly as DFA complexity increases. Hints primarily assist in resolving local misunderstandings, but they do not reliably enable correction of globally inconsistent or structurally flawed automata.

6 Concluding Remarks

This work evaluates LLMs on DFA construction tasks, emphasizing symbolic correctness over surface-level pattern matching. While models reproduce familiar constructions, performance degrades on complex inputs. Failures are systematic, persist across prompting strategies, and indicate structural weaknesses in symbolic reasoning rather than prompt-specific issues. The hint-based framework shows limited self-correction for simpler errors, but deeper inconsistencies are rarely resolved. While our primary evaluation focuses on frontier models, we also conducted supplementary experiments with representative open-source instruction-tuned models. These models exhibit very low performance even on seen DFA construction tasks (LLaMA-3.1-8B: 4.8%, LLaMA-3.1-14B: 9.4%, Qwen-2.5-7B: 3.6%, Qwen-2.5-14B: 7.1%), suggesting that they struggle with the core construction procedure itself rather than only compositional generalization. Moreover, fine-tuning LLaMA-3.1-8B on the TuringQ dataset (Zahraei and Asgari, 2024) does not improve performance, indicating that these limitations are not solely due to lack of training data. For this reason, we prioritize frontier models to study whether these limitations persist even in the strongest available systems.

	Easy	Easy-Medium	Medium-Hard	Hard
Seen	36.67%	35.56%	23.33%	4.44%
Unseen	30.55%	37.77%	14.46%	17.22%

Table 5: Difficulty distribution (DFA states)

Difficulty	**GPT-5.1**		**Grok-4.1**		**Gemini-2.5**	
	Seen	Unseen	Seen	Unseen	Seen	Unseen
Easy	87.87%	29.09%	90.90%	61.81%	96.97%	38.18%
Easy-Medium	84.37%	17.64%	90.62%	58.82%	81.25%	29.41%
Medium-Hard	80.95%	23.07%	85.71%	53.84%	76.19%	26.93%
Hard	75.00%	9.60%	100.00%	58.06%	75.00%	16.12%

Table 6: Difficulty-matched DFA construction accuracy.

A natural question is whether the observed performance drop from seen to unseen tasks can be attributed to increased problem difficulty rather than limited generalization. To investigate this, we introduce a unified difficulty stratification based on the number of states in the minimized DFA, which serves as a proxy for structural complexity. Specifically, we define: Easy (states $\leq$ 5), Easy-Medium (6–9 states), Medium-Hard (10–12 states), and Hard (states $\geq$ 13). This choice enables consistent comparison across the seen dataset (natural language specifications) and unseen dataset (regular expressions), where operator-based measures are not directly comparable. Under this criterion, the overall difficulty distribution between seen and unseen splits is broadly similar, with easy and easy-medium instances comprising the majority in both cases (Table 5). We further evaluate models under matched difficulty levels (Table 6). Across all models, we observe substantial performance degradation from seen to unseen tasks even within the same difficulty tiers, including easy and easy-medium categories. This indicates that the drop in accuracy cannot be explained solely by increased structural complexity, but instead reflects limited compositional generalization to structurally novel language specifications.

7 Acknowledgments

The authors thank Professor Krishnaprasad Thirunarayan, Professor Amit Nanavati, Professor Mehul Raval, the students of CSE 525 and CSE 406 at Ahmedabad University, and the anonymous reviewers.

8 Limitations

This study has several limitations that should be considered when interpreting the results.

Model access and evaluation constraints. All experiments were conducted via public APIs or standard model interfaces with fixed inference-time budgets. As a result, some models—most notably Grok-4.1-fast-reasoning under Tree-of-Thought prompting—frequently failed to return outputs within the allotted budget. Such cases were treated as unsuccessful attempts, reflecting practical interface and deployability constraints rather than definitive limitations of the underlying reasoning capabilities.

Token and state-space scalability. DFA construction from complex regular expressions often induces large intermediate representations, particularly under Chain-of-Thought and Tree-of-Thought prompting. Token budget exhaustion and implicit state-space explosion limit the reliability of these methods for highly nested or compositional expressions, even when the underlying construction is theoretically well defined.

Finite validation horizon. Behavioral equivalence between generated DFAs and ground-truth regular expressions is tested using exhaustive enumeration only up to a bounded string length, supplemented by randomized testing for longer strings. While this approach provides strong empirical assurance, it does not constitute a formal proof of equivalence for all possible strings. As a result, rare counterexamples beyond the tested length range may remain undetected, potentially leading to overestimation of construction correctness.

Scope of tasks and formalisms. The experiments focus exclusively on regular expressions, DFA/NFA construction, and Arden's theorem–based inverse synthesis. Consequently, the reported results may not generalize to richer automata models, non-regular languages, or other formal systems. Accordingly, our findings should not be generalized to richer formalisms such as context-free grammars, pushdown automata, or Turing-complete models.

Model variability across interfaces. Identical prompts issued via APIs and web interfaces can yield different outputs due to undocumented system-level differences. All reported results correspond to API-based evaluations and should not be interpreted as exact replicas of web interface behavior.

Absence of fine-tuning or adaptive prompting. No model-specific fine-tuning, adaptive decoding, or prompt optimization was applied beyond the fixed prompting protocols described in the paper. While this ensures controlled comparability across models, it may underestimate achievable performance under specialized or tuned evaluation settings.

Tool-Augmented Construction. Our evaluation focuses on assessing intrinsic symbolic reasoning in LLMs without external tools. We do not consider settings where models generate and execute algorithms such as Thompson's construction via an external interpreter. While such tool-augmented approaches would likely improve performance, they shift the task from symbolic reasoning to program synthesis and execution, and are therefore outside the scope of this work.

These limitations primarily affect the external validity of our conclusions, but do not alter the observed relative performance trends across models and prompting strategies.

9 Ethical Considerations

This work evaluates the capability of large language models (LLMs) to perform formal language–theoretic reasoning tasks, specifically deterministic finite automaton (DFA) construction and related inverse problems. The study does not involve human subjects, personal data, or sensitive information.

Dataset sourcing. The datasets used in this study consist exclusively of symbolic, mathematical objects, including regular expressions, DFAs, NFAs, transition tables, and multiple-choice or true/false questions.

The Knowledge Checking dataset and the Seen DFA Construction dataset were sourced from publicly available educational materials and standard automata theory problem sets accessible online. These materials are commonly used for teaching and assessment and do not carry restrictive licenses.

The Unseen DFA Construction dataset, including the Mathematical Engineering and Mathematical Art subsets, was manually constructed by the authors and was not sourced from online repositories. All datasets are released with an explicit data license in the accompanying repository.

All newly created datasets are released under the Creative Commons Attribution 4.0 International (CC BY 4.0) license. Publicly sourced educational datasets are provided for research and evaluation purposes only, with original copyrights retained by their respective owners. Full license text and terms of use are provided in the accompanying repository.

Model usage and evaluation. All evaluated models were accessed exclusively through their official public APIs under standard usage conditions. No attempts were made to extract proprietary model internals, training data, or hidden reasoning traces. Prompts were designed solely to elicit task performance and did not request personal, sensitive, or copyrighted information.

Reproducibility and transparency. To promote transparency and responsible research practice, we release the full set of prompts, experimental scripts, validation code, and representative outputs. Automated validation and targeted human inspection were used to reduce the risk of reporting spurious or misleading results.

Potential risks and misuse. The tasks studied are purely formal and mathematical in nature, and the released artifacts are not expected to enable harmful applications or deployment-facing systems. A plausible risk lies in misinterpretation or overgeneralization of the results, for example if observed failures or successes are taken as evidence of general reasoning ability or deficiency across broader linguistic or real-world domains. We explicitly limit our claims to regular-language formalisms and symbolic automata construction and caution against extrapolation beyond these settings.

Use of AI assistants. AI assistants were used in a limited capacity during the development and debugging of auxiliary code. They were not used to generate datasets, ground-truth solutions, experimental results, analyses, or conclusions, and did not contribute to the scientific claims of this work.

Overall, we identify no significant ethical risks associated with this study and adhere to conference guidelines on data sourcing, transparency, and responsible reporting.

References

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

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

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

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2001. *Introduction to Automata Theory, Languages, and Computation*, 2 edition. Addison-Wesley.

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

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Preprint*, arXiv:2205.11916.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. 2022. Solving quantitative reasoning problems with language models. In *Proceedings of the 36th Conference on Neural Information Processing Systems (NeurIPS)*.

Michael Sipser. 2013. *Introduction to the Theory of Computation*, 3 edition. Course Technology.

Xuezhi Wang and Denny Zhou. 2024. Chain-of-thought reasoning without prompting. *Preprint*, arXiv:2402.10200.

Sean Welleck, Jiacheng Liu, Ronan Le Bras, Hannaneh Hajishirzi, Yejin Choi, and Kyunghyun Cho. 2021. Naturalproofs: Mathematical theorem proving in natural language. *Preprint*, arXiv:2104.01112.

Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with large language models. In *Proceedings of the 36th Conference on Neural Information Processing Systems (NeurIPS)*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. In *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS)*.

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

Pardis Sadat Zahraei and Ehsaneddin Asgari. 2024. Turingq: Benchmarking AI comprehension in theory of computation. *Preprint*, arXiv:2410.06547.

A Appendix : Model-Specific DFA Outputs

This appendix reports the exact DFA constructions produced by each evaluated model for the seen task (L_1) and the unseen task (L_2). All outputs are shown exactly as returned by the models, without post-processing or correction.

A.1 Seen Problem (L_1)

A.1.1 GPT-5.1 Output for L_1


```
{
  "states": [
    "q0",
    "q1",
    "q2",
    "q3",
    "q4",
    "q5",
    "q6",
    "q7"
  ],
  "alphabet": [
    "a",
    "b"
  ],
  "start_state": "q0",
  "accept_states": [
    "q4",
    "q5",
    "q6",
    "q7"
  ],
  "transitions": {
    "q0": {
      "a": "q1",
      "b": "q0"
    },
    "q1": {
      "a": "q3",
      "b": "q2"
    },
    "q2": {
      "a": "q5",
      "b": "q4"
    },
    "q3": {
      "a": "q7",
      "b": "q6"
    },
    "q4": {
      "a": "q1",
      "b": "q0"
    },
    "q5": {
      "a": "q3",
      "b": "q2"
    },
    "q6": {
      "a": "q5",
      "b": "q4"
    },
    "q7": {
      "a": "q7",
      "b": "q6"
    }
  }
}
```


A.1.2 Grok-4-1-fast-reasoning Output for L_1


```
{
  "states": [
    "q0",
    "q1",
    "q2",
    "q3",
    "q4",
```

```
    "q5",
    "q6",
    "q7"
  ],
  "alphabet": [
    "a",
    "b"
  ],
  "start_state": "q0",
  "accept_states": [
    "q4",
    "q5",
    "q6",
    "q7"
  ],
  "transitions": {
    "q0": {
      "a": "q1",
      "b": "q0"
    },
    "q1": {
      "a": "q3",
      "b": "q2"
    },
    "q2": {
      "a": "q5",
      "b": "q4"
    },
    "q3": {
      "a": "q7",
      "b": "q6"
    },
    "q4": {
      "a": "q1",
      "b": "q0"
    },
    "q5": {
      "a": "q3",
      "b": "q2"
    },
    "q6": {
      "a": "q5",
      "b": "q4"
    },
    "q7": {
      "a": "q7",
      "b": "q6"
    }
  }
}
```

A.1.3 Gemini-2.5-flash Output for L_1

```
{
  "states": [
    "q0",
    "q1",
    "q2",
    "q3",
    "q4",
    "q5",
    "q6",
    "q7"
  ],
  "alphabet": [
    "a",
    "b"
  ],
  "start_state": "q0",
  "accept_states": [
    "q4",
    "q5",
    "q6",
    "q7"
  ],
  "transitions": {
    "q0": {
      "a": "q1",
      "b": "q0"
    },
    "q1": {
      "a": "q2",
      "b": "q3"
    },
    "q2": {
      "a": "q4",
      "b": "q5"
    },
    "q3": {
      "a": "q6",
      "b": "q7"
    },
    "q4": {
      "a": "q4",
      "b": "q5"
    },
    "q5": {
      "a": "q6",
      "b": "q7"
    },
    "q6": {
      "a": "q2",
      "b": "q3"
    },
    "q7": {
      "a": "q1",
      "b": "q0"
    }
  }
}
```

A.2 Unseen Problem (L_2)

A.2.1 GPT-5.1 Output for L_2

```
{
  "states": [
    "q0",
    "q1",
    "q2",
    "q3",
    "q4",
    "q5",
    "q6",
    "q7",
    "q8",
    "q9",
    "q10",
    "q11",
    "q12",
    "q13",
    "q14",
    "q15",
    "q16",
    "q17",
    "q18",
    "q19",
    "q20",
    "q21",
    "q22",
    "q23",
    "q24",
    "q25",
    "q26",
    "q27",
    "q28",
    "q29",
    "q30",
    "q31",
    "qDead"
  ],
  "alphabet": [
    "a",
    "b"
  ],
  "start_state": "q0",
  "accept_states": [
    "q24",
    "q25",
    "q26",
    "q27",
    "q28",
    "q29",
    "q30",
    "q31"
  ],
  "transitions": {
    "q0": {
      "a": "q1",
      "b": "q2"
    },
```

```
    "q1": {
      "a": "q3",
      "b": "q4"
    },
    "q2": {
      "a": "q1",
      "b": "qDead"
    },
    "q3": {
      "a": "q5",
      "b": "q6"
    },
    "q4": {
      "a": "q3",
      "b": "q7"
    },
    "q5": {
      "a": "q8",
      "b": "q9"
    },
    "q6": {
      "a": "q5",
      "b": "q10"
    },
    "q7": {
      "a": "q6",
      "b": "q11"
    },
    "q8": {
      "a": "q12",
      "b": "q13"
    },
    "q9": {
      "a": "q8",
      "b": "q14"
    },
    "q10": {
      "a": "q9",
      "b": "q15"
    },
    "q11": {
      "a": "q10",
      "b": "q16"
    },
    "q12": {
      "a": "q17",
      "b": "q18"
    },
    "q13": {
      "a": "q12",
      "b": "q19"
    },
    "q14": {
      "a": "q13",
      "b": "q20"
    },
    "q15": {
      "a": "q14",
      "b": "q21"
    },
    "q16": {
      "a": "q15",
      "b": "q22"
    },
    "q17": {
      "a": "q23",
      "b": "q24"
    },
    "q18": {
      "a": "q17",
      "b": "q25"
    },
    "q19": {
      "a": "q18",
      "b": "q26"
    },
    "q20": {
      "a": "q19",
      "b": "q27"
    },
    "q21": {
      "a": "q20",
      "b": "q28"
    },
    "q22": {
      "a": "q21",
      "b": "q29"
    },
    "q23": {
      "a": "q23",
      "b": "q30"
    },
    "q24": {
      "a": "q23",
      "b": "q31"
    },
    "q25": {
      "a": "q24",
      "b": "q31"
    },
    "q26": {
      "a": "q25",
      "b": "q31"
    },
    "q27": {
      "a": "q26",
      "b": "q31"
    },
    "q28": {
      "a": "q27",
      "b": "q31"
    },
    "q29": {
      "a": "q28",
      "b": "q31"
    },
    "q30": {
      "a": "q30",
      "b": "q31"
    },
    "q31": {
      "a": "q30",
      "b": "q31"
    },
    "qDead": {
      "a": "qDead",
      "b": "qDead"
    }
  }
}
```


A.2.2 Grok-4-1-fast-reasoning Output for L_2


```
{

  "states": [
    "q0",
    "q1",
    "q2",
    "q3",
    "q4",
    "q5",
    "q6",
    "q7",
    "q8",
    "q9",
    "q10",
    "q11",
    "q12",
    "q13",
    "q14",
    "q15",
    "q16",
    "q17",
    "q18",
    "q19",
    "q20",
    "q21",
    "q22",
    "q23",
    "q24",
    "q25",
    "q26",
    "q27",
    "q28"
  ],
  "alphabet": [
    "a",
    "b"
  ],
  "start_state": "q0",
  "accept_states": [
    "q13",
```

```
    "q14",
    "q15",
    "q16",
    "q17",
    "q18",
    "q19",
    "q20"
  ],
  "transitions": {
    "q0": {
      "a": "q1",
      "b": "q2"
    },
    "q1": {
      "a": "q3",
      "b": "q4"
    },
    "q2": {
      "a": "q5",
      "b": "q12"
    },
    "q3": {
      "a": "q6",
      "b": "q7"
    },
    "q4": {
      "a": "q8",
      "b": "q9"
    },
    "q5": {
      "a": "q10",
      "b": "q11"
    },
    "q6": {
      "a": "q13",
      "b": "q14"
    },
    "q7": {
      "a": "q15",
      "b": "q16"
    },
    "q8": {
      "a": "q17",
      "b": "q18"
    },
    "q9": {
      "a": "q19",
      "b": "q20"
    },
    "q10": {
      "a": "q21",
      "b": "q22"
    },
    "q11": {
      "a": "q23",
      "b": "q24"
    },
    "q12": {
      "a": "q12",
      "b": "q12"
    },
    "q13": {
      "a": "q13",
      "b": "q14"
    },
    "q14": {
      "a": "q15",
      "b": "q16"
    },
    "q15": {
      "a": "q17",
      "b": "q18"
    },
    "q16": {
      "a": "q19",
      "b": "q20"
    },
    "q17": {
      "a": "q21",
      "b": "q22"
    },
    "q18": {
      "a": "q23",
      "b": "q24"
    },
    "q19": {
      "a": "q25",
      "b": "q26"
    },
    "q20": {
      "a": "q27",
      "b": "q28"
    },
    "q21": {
      "a": "q13",
      "b": "q14"
    },
    "q22": {
      "a": "q15",
      "b": "q16"
    },
    "q23": {
      "a": "q17",
      "b": "q18"
    },
    "q24": {
      "a": "q19",
      "b": "q20"
    },
    "q25": {
      "a": "q21",
      "b": "q22"
    },
    "q26": {
      "a": "q23",
      "b": "q24"
    },
    "q27": {
      "a": "q25",
      "b": "q26"
    },
    "q28": {
      "a": "q27",
      "b": "q28"
    }
  }
}
```


A.2.3 Gemini-2.5-flash Output for L_2


```
{
  "states": [
    "q0",
    "q1",
    "q2",
    "q3",
    "q4",
    "q5",
    "q6",
    "q7",
    "q8",
    "q9",
    "q10",
    "q11",
    "q12",
    "q13",
    "q14",
    "q15"
  ],
  "alphabet": [
    "a",
    "b"
  ],
  "start_state": "q0",
  "accept_states": [
    "q7",
    "q8",
    "q9",
    "q10"
  ],
  "transitions": {
    "q0": {
      "a": "q2",
      "b": "q1"
    },
    "q1": {
      "a": "q5",
      "b": "q15"
    },
    "q2": {
      "a": "q3",
      "b": "q4"
    },
```

```
    "q3": {
      "a": "q7",
      "b": "q8"
    },
    "q4": {
      "a": "q9",
      "b": "q10"
    },
    "q5": {
      "a": "q11",
      "b": "q12"
    },
    "q6": {
      "a": "q13",
      "b": "q14"
    },
    "q7": {
      "a": "q7",
      "b": "q8"
    },
    "q8": {
      "a": "q9",
      "b": "q10"
    },
    "q9": {
      "a": "q11",
      "b": "q12"
    },
    "q10": {
      "a": "q13",
      "b": "q14"
    },
    "q11": {
      "a": "q7",
      "b": "q8"
    },
    "q12": {
      "a": "q9",
      "b": "q10"
    },
    "q13": {
      "a": "q11",
      "b": "q12"
    },
    "q14": {
      "a": "q13",
      "b": "q14"
    },
    "q15": {
      "a": "q15",
      "b": "q15"
    }
  }
}
```


B Appendix : Datasets

This appendix documents the datasets used in our experiments, including their composition, sizes, formats, and public availability. Detailed dataset construction procedures, design motivations, and validation protocols are described in the main paper.

B.1 Overview

We evaluate large language model (LLM) performance using four datasets spanning factual knowledge, seen DFA construction, and two categories of unseen DFA construction tasks. All datasets are grounded in formal language theory and deterministic finite automata. Ground-truth answers are verified through manual construction and automated validation procedures described in the main paper.

B.2 Dataset Composition and Sizes

- **Knowledge Checking Dataset** (50 questions):
  This dataset consists of multiple-choice and short-answer questions assessing foundational knowledge of core Theory of Computation concepts. The questions cover key topics required for DFA construction, including DFA definitions and acceptance semantics, regular expression semantics (including Kleene-star behavior), closure properties of regular languages, NFA–DFA equivalence and subset construction, DFA minimization and state equivalence, and basic language containment reasoning.
  To ensure coverage beyond trivial recall, the dataset spans multiple levels of difficulty: 64% conceptual (recall-level), 28% application-level, and 8% multi-step reasoning questions. This distribution reflects the range of conceptual understanding and reasoning required for DFA construction tasks.
- **Unseen DFA Construction Dataset** (180 questions):
  This dataset is designed to evaluate generalization beyond memorized patterns and is intentionally constructed to avoid overlap with common instructional examples. It is divided into two subsets that differ in their generative principles:
  - **Mathematical Art Subset** (60 questions):
    This subset consists of problems generated by combining multiple interacting constraints, including structural restrictions on automata, conditional dependencies between symbols, and narrative-based task formulations. The resulting languages are intentionally highly irregular and non-canonical, designed to stress symbolic consistency, constraint integration, and long-horizon reasoning.
    To design the languages in this unseen subset, we employ the following principled construction strategies:
    (a) **Use of product construction.** Multiple constraints are combined using product-style constructions (e.g., repeated use of conjunctions and disjunctions to increase constraint complexity); for example,

L_a = { Construct a DFA that accepts the set of all strings over {a,b} where it starts with either 'aba' or 'bab' and ends with 'bab' or 'aba' respectively. }

L_b = { Construct a DFA that accepts the set of all strings over {a,b} where 'ab' should be followed by 'ba' and 'aa' should be followed by 'bb' and the total count of 'a's and 'b's is even. }

(b) **Multiple interacting constraints.** Independent constraints (e.g., prefix conditions paired with counting or exclusion constraints) are imposed simultaneously, preventing purely local reasoning from yielding a correct construction; for example,

L_c = { Construct a DFA over {a,b} that accepts all strings in which the fourth-last symbol from the end must be 'a' and the substring 'bb' does not appear before any 'a'. }

L_d = { Construct a DFA over {a,b,c} such that there exists a block of A's of length no more than 5, immediately followed by a block of B's that is twice the length of the A block; the character C may appear anywhere before or after these two consecutive blocks with $|C| \equiv 0 \pmod 3$, and no C appears between the two AB blocks. }

(c) **Structural restrictions.** Structural constraints such as adjacency rules, forbidden substrings, or positional patterns are enforced, requiring precise tracking of symbol relationships; for example,

L_e = { Construct a DFA over {0,1} that accepts a string w such that the digit obtained by the XOR operation of the first three digits is equal to the digit present at the end of the string w. }

L_f = { Construct a DFA over {0,1} such that the string length is greater than or equal to 4, and in each substring of length 4 the first two bits are interpreted as a decimal number and the last two bits are interpreted as another decimal number, and their multiplication is divisible by 4. }

(d) **Narrative-based tasks.** Narrative-driven problems are formulated using real-world or game-based scenarios, which must be translated into precise symbolic automaton constraints; for example,

L_g = { In chess, consider the Ruy-Lopez opening. States correspond to positions of pieces on the board. The input alphabet is defined as Sigma = {pxyi}, where p denotes the player (white or black), x denotes the piece, y the square alphabet, and i the square number. A move such as a black knight moving to c6 is written as bkc6. Assuming standard chess rules with alternating turns and a fixed initial board configuration, construct a minimized DFA whose accepting paths correspond to the Ruy-Lopez opening

Figure 3: Initial board configuration before the Ruy-Lopez opening sequence.

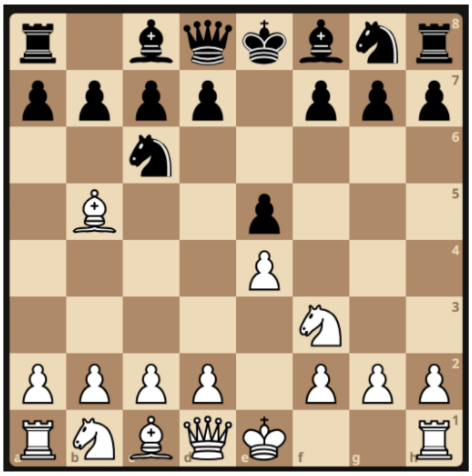


Figure 4: Illustration of the Ruy-Lopez opening sequence considered in L_g.

`achieved in exactly five moves, excluding the two initial moves. }`

L_h `= { You are a fruit juice manufacturing company owner with three possible flavors: sweet, sour, and mild. Five syrups are available: 0-Orange, 1-Lemon, 2-Mango, 3-Banana, and 4-Grape. The flavor combinations are defined as follows: Sweet = Mango + Grape; Sour = Orange + Lemon + Grape; Mild = Banana + Mango. Construct a DFA that accepts exactly those strings that constitute a particular flavor according to these rules. }`

Across these strategies, we include concrete instances such as maze-solving analogies, real-world scenario encodings, and composite logical requirements to illustrate their effect on DFA construction difficulty. Each strategy is represented by multiple example languages in the dataset.

– **Mathematical Engineering Subset** (120 questions):
  This subset is constructed using Arden's theorem as a core generative tool. Regular expressions are derived from systematically designed DFA and NFA transition structures, yielding symbolically precise but nonstandard DFA construction problems that emphasize algebraic manipulation and formal reasoning rather than surface familiarity.

B.3 Data Format

All DFA construction datasets (seen and unseen) are provided in **PDF and JSON formats** for both human and machine readability. Each problem instance includes:

- The problem statement (regular expression or formal language specification),
- The corresponding DFA state transition table,
- A visual DFA diagram used for reference and verification.

The Knowledge Checking dataset is provided in both PDF and JSON formats and includes questions with explicitly labeled correct answers.

B.4 Ground Truth and Validation

All ground-truth DFAs and corresponding answers were manually constructed by the authors and validated through formal reasoning and automated behavioral equivalence testing, as described in the main paper. Dataset construction and validation were conducted over a period of more than three months in the context of advanced Theory of Computation coursework. The process can be summarized as follows:

- **Course Context:** Dataset development was carried out within two advanced Theory of Compu-

tation courses:
  - **CSE 406 (Theory of Computation):** 193 undergraduate students.
  - **CSE 525 (Advanced Theory of Computation):** 107 students, comprising both undergraduate and graduate students.

  In total, instructional materials involved contributions across multiple course offerings from **300 students**.
- **Nature of Student Contributions:** Student contributions occurred strictly within the normal scope of coursework and assessment activities. No student-generated data was collected, analyzed, or annotated specifically for the purposes of this research, and students were not treated as research participants.
- **Instructional Review and Validation:** Instructional materials underwent systematic review and validation involving:
  - **Five teaching assistants** responsible for intermediate checking, consistency verification, and instructional review.
  - **Two faculty members** who performed final checking and validation.
  - A dedicated **team of research assistants** supporting curation, standardization, and cross-verification.

All final problem instances and ground-truth solutions included in the datasets were subsequently curated, standardized, and independently verified by the authors to ensure correctness, consistency, and suitability for research use.

For the Seen DFA Construction dataset, the repository additionally includes supporting documentation demonstrating the presence of structurally equivalent questions in publicly accessible online sources. This serves as explicit evidence of prior availability and confirms that these instances were not newly introduced for this work.

B.5 Availability and Licensing

All datasets, prompts, experimental scripts, validation code, evidence of internet availability for seen questions, and representative sample outputs are publicly released at:

https://anonymous.4open.science/r/dfa-llm-evaluation-B82D/

The repository includes:

- Dataset PDFs and JSON files for all four datasets,
- Machine-readable metadata files describing dataset properties,
- Prompt templates and experimental scripts,
- Automated validation and evaluation code,
- Proof of public availability for seen dataset questions,
- Sample model outputs and validation reports.

All datasets are released under an explicit data license included in the repository, permitting research use and reproducibility in accordance with conference guidelines.

C Appendix : Difficulty Labeling Criteria

C.1 Mathematical Engineering Dataset

For the regular-expression–based dataset, difficulty labels are assigned using a quantitative *structural complexity score* derived directly from the syntax of each regular expression. The score combines the following five measurable properties:

- *Maximum nesting depth*, capturing hierarchical structure and long-range dependencies;
- *Number of union operators* ($|$), reflecting branching and nondeterminism;
- *Number of Kleene stars* ($*$), indicating unbounded repetition;
- *Implicit concatenations*, representing sequential composition complexity;
- *Expression length*, capturing overall description size (log-scaled).

These features are linearly combined into a single scalar score. Regular expressions are then partitioned into *easy*, *medium*, and *difficult* categories using dataset-wide tertiles of this score. This procedure yields adaptive and reproducible difficulty labels without the use of manually tuned thresholds. The full implementation used in our experiments is shown below.

```
import math
import json

# -------- Regex Analysis --------

def analyze_regex(regex):
    length = len(regex)

    union = 0
    star = 0
    concat = 0

    nesting = 0
    max_nesting = 0

    prev = None

    for c in regex:
        if c == '(':
            nesting += 1
            max_nesting = max(max_nesting,
  nesting)
```

```
        elif c == ')':
            nesting -= 1
        elif c == '|':
            union += 1
        elif c == '*':
            star += 1
        elif prev and prev not in '(|' and c
    not in '|)*':
            concat += 1

        prev = c

    return {
        "length": length,
        "union": union,
        "star": star,
        "concat": concat,
        "nesting": max_nesting
    }

# -------- Difficulty Score --------

def difficulty_score(m):
    return (
        1.5 * m["nesting"] +
        1.0 * m["union"] +
        0.5 * m["star"] +
        0.2 * m["concat"] +
        math.log1p(m["length"])
    )

# -------- Dataset Classification --------

def classify_dataset(entries):
    analyzed = []

    for e in entries:
        metrics = analyze_regex(e["regex"])
        score = difficulty_score(metrics)
        analyzed.append((e, metrics, score))

    scores = sorted(score for _, _, score in
    analyzed)

    q1 = scores[int(0.33 * len(scores))]
    q2 = scores[int(0.66 * len(scores))]

    output = []
    for e, metrics, score in analyzed:
        if score <= q1:
            difficulty = "easy"
        elif score <= q2:
            difficulty = "medium"
        else:
            difficulty = "difficult"

        entry = e.copy()
        entry["metrics"] = metrics
        entry["score"] = round(score, 2)
        entry["difficulty"] = difficulty
        output.append(entry)

    return output

# -------- Example Usage --------

if __name__ == "__main__":
    dataset = [
        {
            "id": "r001",
            "regex":
    "((b*a)|(b*a(a|b)*(a|b)))a*(a|b)a*aa*",
            "alphabet": ["a", "b"]
        }
    ]

    classified = classify_dataset(dataset)
    print(json.dumps(classified, indent=2))
```

C.2 Mathematical Art Dataset

The *Mathematical Art* dataset consists of natural-language DFA construction tasks characterized by multiple interacting constraints. Unlike regular-expression–based problems, these tasks are not defined by formal syntax alone; difficulty is therefore assigned based on semantic structure rather than surface form.

Specifically, difficulty labels are assigned as follows:

- **Easy and medium** tasks are distinguished based on the number of explicit constraints and the extent of their interaction (e.g., positional constraints, counting constraints, and forbidden substrings).
- **Difficult** tasks involve narrative-based formulations, such as grid navigation, game scenarios, or real-world analogies. These problems require multi-step interpretation, implicit state tracking, and translation from informal descriptions into formal automata.

This labeling scheme reflects the additional reasoning burden imposed by narrative grounding and constraint integration, which consistently leads to higher error rates across evaluated models.

D Appendix: Prompt Templates

This appendix provides the exact prompts used in all experiments. Prompts are reported verbatim to ensure reproducibility.

D.1 Prompts Used for Regular Expression to DFA Construction

D.1.1 Direct Input–Output Prompt

```
SYSTEM:
You are an expert in formal languages and
     automata.

USER:
Given the following regular expression and
     alphabet:

REGEX: {{REGEX}}
ALPHABET: {{ALPHABET}}

Task:
Construct a correct deterministic finite
     automaton (DFA) that recognizes exactly
the language denoted by the regular expression.

Constraints:
- The DFA must be total (every state has exactly
     one transition per symbol).
- Use short state names: q0, q1, q2, ...
- Do NOT include explanations, reasoning,
     derivations, or intermediate steps.
- Do NOT include any text outside the JSON object.

OUTPUT ONLY a single JSON object matching EXACTLY
     this schema:

{
  "states": ["q0", "q1", "..."],
  "alphabet": ["a", "b", "..."],
  "start_state": "q0",
  "accept_states": ["q1", "..."],
  "transitions": {
    "q0": { "a": "q1", "b": "q0" },
```

```
    "q1": { "a": "q1", "b": "q2" }
  }
}
```

D.1.2 Chain-of-Thought (CoT) Prompt

```
SYSTEM:
You are an expert in formal languages and finite
     automata. When asked to produce a
DFA transition table, THINK step-by-step
     internally (Chain-of-Thought) but DO NOT
output any reasoning. Output only a single valid
     JSON object that exactly matches
the schema described below.

USER:
Given the following regular expression and
     alphabet:

REGEX: {{REGEX}}
ALPHABET: {{ALPHABET}}

Task:
1. Internally (chain-of-thought) derive a correct
     deterministic finite automaton
   (DFA) that recognizes the language of the
     regular expression.
2. Do not output any intermediate reasoning or
     explanation.
3. OUTPUT ONLY a single JSON object that contains
     the DFA transition table and
   nothing else. The JSON must match the exact
     schema (keys and types) below.

Required JSON schema (exact keys and types):
{
  "states": ["q0", "q1", ...],
  "alphabet": ["a", "b", ...],
  "start_state": "q0",
  "accept_states": ["q1", ...],
  "transitions": {
    "q0": { "a": "q1", "b": "q0" },
    "q1": { "a": "q1", "b": "q2" }
  }
}

Formatting rules and constraints (must follow
     exactly):
- The JSON object must contain only the five keys
     above and nothing else.
- The DFA must be total: for every state and for
     every symbol in "alphabet",
  there must be exactly one target state.
- Use short state names like "q0", "q1", "q2", ...
- All values must be valid JSON types.
- Do not include comments, trailing commas, or
     extra text.
- If construction fails, output the following
     machine-readable failure JSON:
  {"error":"cannot_construct","reason":"<one-
line_reason>"}

Edge-case guidance (zero-shot):
- Output ONLY the JSON object (or the failure
     JSON above).
- No diagnostic text before or after the JSON.
- The experiment script retries up to three times.
```

D.1.3 Chain-of-Thought One-Shot Prompt

```
SYSTEM:
You are an expert in formal languages and finite
     automata. When asked to produce a DFA
     transition table, THINK step-by-step
     internally (Chain-of-Thought) but DO NOT
     output any reasoning. Output only a single
     valid JSON object that exactly matches the
     schema described below.



USER:
Below is an example (one-shot) followed by the
     real input. Read the example carefully -- it
     demonstrates exact JSON schema, formatting,
     and totality rules. After the example,
     produce the DFA JSON for the real input only.



=== EXAMPLE (one-shot) ===
Regular expression: (a|b)*
Alphabet: ["a","b"]

Correct DFA (JSON only):
{
  "states": ["q0"],
  "alphabet": ["a", "b"],
  "start_state": "q0",
  "accept_states": ["q0"],
  "transitions": {
    "q0": { "a": "q0", "b": "q0" }
  }
}


=== END EXAMPLE ===

Now the actual input:

REGEX: {{REGEX}}
ALPHABET: {{ALPHABET}}

Task:
1. Internally (chain-of-thought) derive a correct
     deterministic finite automaton (DFA) that
     recognizes the language of the regular
     expression.
2. Do not output any intermediate reasoning or
     explanation.
3. OUTPUT ONLY a single JSON object that contains
     the DFA transition table and nothing else.
     The JSON must match the exact schema (keys
     and types) below.

Required JSON schema (exact keys and types):
{
  "states": ["q0", "q1", ...],
  "alphabet": ["a", "b", ...],
  "start_state": "q0",
  "accept_states": ["q1", ...],
  "transitions": {
    "q0": { "a": "q1", "b": "q0" },
    "q1": { "a": "q1", "b": "q2" }
  }
}



Formatting rules and constraints (must follow
     exactly):
- The JSON object must contain only the five keys
     above and nothing else.
- The DFA must be total: for every state and for
     every symbol in "alphabet", there must be
     exactly one target state.
- Use short state names like "q0", "q1", "q2",
     ... (no spaces, no special characters).
- All values must be valid JSON types (arrays,
     strings, objects). Do not include comments
     or trailing commas.
- Do not include any text before or after the
     JSON object (no backticks, no code fences,
     no extra explanation).
- If you cannot produce a correct DFA, DO NOT
     output {}. Instead, attempt to produce a
     compact but valid DFA. Only if you truly
     cannot construct any DFA, output a
     machine-readable failure JSON:
  {"error":"cannot_construct","reason":
  "<one-line_reason>"}
  (This is allowed but discouraged - prefer
     producing a compact DFA.)
- If the DFA is large, produce a compact encoding
     using short state names (q0,q1,q2,...) and
     include every transition for each symbol.


Edge-case guidance (one-shot):
```

```
- The above example demonstrates the EXACT output
     shape and formatting (including
     spacing/newlines are not important, but
     content must be valid JSON).
- The experiment script will retry up to 3 times
     if the first output is invalid. OUTPUT ONLY
     the JSON object (or the small error JSON) --
     no additional text.

IMPORTANT: OUTPUT ONLY the JSON object (or the
     small error JSON). Do not output any other
     text.
```

D.1.4 Tree-of-Thought (ToT) Direct-Branch Prompt

```
SYSTEM:
You are an expert in formal languages and finite
     automata. When asked to produce a DFA
     transition table, THINK step-by-step
     internally (Chain-of-Thought) but DO NOT
     output any reasoning. Output only a single
     valid JSON object that exactly matches the
     schema described below.

USER:
Given the following regular expression and
     alphabet:

REGEX: {{REGEX}}
ALPHABET: {{ALPHABET}}

Task:
1. Internally derive a correct deterministic
     finite automaton (DFA) that recognizes
     exactly the language denoted by the regular
     expression.
2. Use any sound formal reasoning internally, but
     do not output any intermediate steps.
3. OUTPUT ONLY a single JSON object that contains
     the DFA transition table and nothing else.

Required JSON schema (exact keys and types):
{
  "states": ["q0", "q1", ...],
  "alphabet": ["a", "b", ...],
  "start_state": "q0",
  "accept_states": ["q1", ...],
  "transitions": {
    "q0": { "a": "q1", "b": "q0" },
    "q1": { "a": "q1", "b": "q2" }
  }
}

Formatting rules and constraints:
- Output only the JSON object, with no extra text.
- The DFA must be total.
- Use short state names q0, q1, q2, ...
- Do not include comments, explanations, or
     trailing commas.
- If construction is difficult, still output a
     compact but valid DFA rather than an empty
     object.
```

D.1.5 Tree-of-Thought (ToT) Minimization Prompt

```
SYSTEM:
You are an expert in deterministic finite
     automata. Internally aim to construct a
     correct DFA that recognizes the given
     regular expression and is close to minimal
     in the number of states. You may internally
     apply minimization techniques such as state
     equivalence or partition refinement. Do NOT
     output any reasoning.

USER:
Given the following regular expression and
     alphabet:

REGEX: {{REGEX}}
ALPHABET: {{ALPHABET}}

Task:
1. Internally construct a correct DFA for the
     regular expression.
2. Internally minimize the DFA if possible.
3. Ensure the DFA is total.
4. OUTPUT ONLY the final minimized DFA as a JSON
     object matching the schema below.

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" }
  }
}

Formatting constraints:
- Output only the JSON object.
- No explanations or comments.
- DFA must be total.
```

D.1.6 Tree-of-Thought (ToT) Derivative Method Prompt

```
SYSTEM:
You are an expert in formal language theory.
     Internally construct a DFA using regular
     expression derivatives (Brzozowski or
     Antimirov derivatives). Each DFA state
     corresponds to a derivative of the original
     regular expression. Do NOT output any
     intermediate reasoning.

USER:
Given the following regular expression and
     alphabet:

REGEX: {{REGEX}}
ALPHABET: {{ALPHABET}}

Task:
1. Internally compute the set of regular
     expression derivatives with respect to the
     alphabet.
2. Internally build the corresponding DFA from
     these derivatives.
3. Ensure the DFA is total.
4. OUTPUT ONLY the final DFA as a JSON object
     matching the exact schema below.

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.
```

D.1.7 Tree-of-Thought (ToT) Thompson's Construction Prompt

```
SYSTEM:
You are an expert in automata theory. Internally
     follow a Thompson-style construction: first
     convert the regular expression to an
     $\varepsilon$-NFA, then apply subset
```

```
     construction to obtain a DFA, and finally
     ensure the DFA is total. Do NOT output any
     intermediate structures or reasoning.


USER:
Given the following regular expression and
     alphabet:

REGEX: {{REGEX}}
ALPHABET: {{ALPHABET}}

Task:
1. Internally construct an $\varepsilon$-NFA
     using Thompson's construction.
2. Internally determinize the NFA using subset
     construction.
3. Internally make the DFA total by adding a sink
     state if necessary.
4. OUTPUT ONLY the final DFA as a single JSON
     object matching the schema below.

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" }
  }
}

Formatting rules:
- Output ONLY the JSON object.
- DFA must be total.
- No explanations, comments, or extra text.
- Use short state names q0, q1, q2, ...
```

D.2 Hint-Based Framework Prompts

This section lists the hints used to evaluate whether models can correct their own errors under guided feedback.

D.2.1 Initial Prompt (No Hints)

```
Construct a deterministic finite automaton (DFA)
     that recognizes exactly the following
     language over the alphabet {a, b}:

L = (b)*(a)(a + ba)*(bb)(a + b)* + (b)*b(b)*a(a +
     b)*

Requirements:
- The DFA must be total.
- Use short state names (q0, q1, q2, ...).
- Specify the start state, accept states, and
     transitions.
- Output only the final DFA.
```

D.2.2 Hint 1:Counterexamples protocol

```
You are given the following language:

L = (b)*(a)(a + ba)*(bb)(a + b)* + (b)*b(b)*a(a +
     b)*

The DFA you previously constructed does not
     correctly recognize this language.

In particular, the following strings belong to L
     but are rejected by your DFA:
- ba
- bba
- baa

Revise or reconstruct the DFA so that it accepts
     all valid strings in L while continuing to
     reject invalid strings. Output only the
     corrected DFA.
```

D.2.3 Hint 2: Error localization

```
You are given the following language:

L = (b)*(a)(a + ba)*(bb)(a + b)* + (b)*b(b)*a(a +
     b)*

Your revised DFA still contains an error.

At least one transition does not correctly
     reflect the language definition, leading to
     incorrect acceptance or rejection of some
     strings.

Re-examine the DFA structure and transition
     assignments, and provide a corrected DFA
     that recognizes exactly L. Output only the
     DFA.
```

D.2.4 Final Hint: Explicit error disclosure

```
You are given the following language:

L = (b)*(a)(a + ba)*(bb)(a + b)* + (b)*b(b)*a(a +
     b)*

The remaining error arises from an incomplete
     interpretation of the regular expression.

In particular, the Kleene-star subexpression (a +
     ba)* permits arbitrary repetitions and
     combinations that must be fully captured by
     the automaton.

As a consequence of this incomplete
     interpretation, your DFA incorrectly accepts
     the string:

aabab

Reconstruct the DFA with a correct interpretation
     of all subexpressions and their
     interactions. Output only the final
     corrected DFA.
```

E Appendix: Evaluation and Validation Code

Automated Validation Pipeline. The automated validation pipeline provides a principled and reproducible mechanism for assessing the correctness of LLM-generated DFAs. It combines (i) strict structural validation, ensuring syntactic correctness, totality, and well-formed transitions, with (ii) behavioral equivalence testing against the ground-truth regular expression. Behavioral validation is performed using exhaustive enumeration of all strings up to a fixed length and is supplemented with randomized sampling of longer strings, enabling detection of both local and global acceptance errors. This design ensures that reported correctness reflects semantic language equivalence rather than surface-level plausibility.

Parameter	GPT-5.1	Gemini-2.5	Grok-4.1
API endpoint	Public	Public	Public
Model variant	GPT-5.1	Gemini-2.5-Flash	Grok-4.1-fast-reasoning
Temperature	0.0	0.0	0.0
Max output tokens	4000	4000	4000
Reasoning toggle	Not exposed	Not exposed	Not exposed
Prompt template	Identical	Identical	Identical
Retries	3	3	3
Timeout	120s	120s	120s

Table 7: API parameters used across model providers. No provider-specific reasoning or deliberation modes were enabled.

The validator is model-agnostic and deterministic, producing identical results given the same DFA and regular expression. It records explicit counterexamples whenever a mismatch is detected, supporting transparent error analysis and independent verification. All validation artifacts, including per-instance reports and aggregated summaries, are saved to disk to facilitate reproducibility and post-hoc inspection.

At the same time, the validation procedure has inherent limitations. Exhaustive testing is bounded by a maximum string length, and while random sampling extends coverage, formal equivalence between a DFA and a regular expression is undecidable via finite testing alone. As a result, DFAs classified as correct should be interpreted as *likely correct* within the tested bounds rather than formally proven equivalent. Additionally, the validator assumes the correctness of the regular expression semantics as interpreted by the host regex engine, which may differ from theoretical automata semantics in edge cases. These limitations are mitigated in the main evaluation by complementary human expert inspection, but they remain important considerations when interpreting automated results.

E.1 Automated DFA Validation Pipeline

```
#!/usr/bin/env python3
"""
validate_dfa_outputs.py

Automated DFA validation pipeline used in
     experiments.

Validation protocol:
1. Schema validation (JSON structure + required
     fields).
2. Totality validation (every state has one
     transition per symbol).
3. Behavioral equivalence testing against the
     ground-truth regular expression:
   - Exhaustive enumeration up to
     MAX_EXHAUSTIVE_LEN.
   - Large-scale randomized testing for longer
     strings.

Outputs:
- Per-DFA detailed validation reports (JSON).
- Aggregated summary table (CSV + JSON).
This script is model-agnostic and
     dataset-agnostic.
"""

import os
import json
import re
import random
import csv
from itertools import product
from typing import List, Dict, Tuple

# ============================
# CONFIGURATION (EDIT HERE)
# ============================

TABLES_DIR = "outputs/tables"
RAW_DIR = "outputs/raw"
VALID_DIR = "outputs/validation"

MAX_EXHAUSTIVE_LEN = 6        # exhaustive
     testing for strings of length 0..6
N_RANDOM = 2000               # number of random
     test strings
MAX_RANDOM_LEN = 15           # maximum length of
     random strings
MAX_COUNTEREXAMPLES = 100     # cap stored
     counterexamples per DFA
RANDOM_SEED = 42              # reproducibility

# ============================
# INITIALIZATION
# ============================

os.makedirs(VALID_DIR, exist_ok=True)
random.seed(RANDOM_SEED)

# ============================
# UTILITIES
# ============================

def load_json(path: str):
    with open(path, "r", encoding="utf-8") as f:
        return json.load(f)

def save_json(path: str, obj):
    with open(path, "w", encoding="utf-8") as f:
        json.dump(obj, f, indent=2,
     ensure_ascii=False)

def is_valid_schema(dfa: Dict) -> Tuple[bool,
     str]:
    required = {"states", "alphabet",
     "start_state", "accept_states",
     "transitions"}
    if not isinstance(dfa, dict):
        return False, "not_json_object"
    if not required.issubset(dfa.keys()):
        return False, "missing_required_keys"
    if not isinstance(dfa["states"], list):
        return False, "states_not_list"
    if not isinstance(dfa["alphabet"], list):
        return False, "alphabet_not_list"
    if dfa["start_state"] not in dfa["states"]:
        return False, "invalid_start_state"
    if not isinstance(dfa["accept_states"], list):
        return False, "accept_states_not_list"
    if not isinstance(dfa["transitions"], dict):
```

```
        return False, "transitions_not_dict"
    return True, "ok"

def check_totality(dfa: Dict):
    missing = []
    invalid_targets = []

    for s in dfa["states"]:
        if s not in dfa["transitions"]:
            missing.append({"state": s, "reason":
     "no_transition_block"})
            continue
        for a in dfa["alphabet"]:
            if a not in dfa["transitions"][s]:
                missing.append({"state": s,
     "symbol": a})
            else:
                tgt = dfa["transitions"][s][a]
                if tgt not in dfa["states"]:

     invalid_targets.append({"state": s,
     "symbol": a, "target": tgt})

    return missing, invalid_targets

def simulate_dfa(dfa: Dict, word: str) -> bool:
    cur = dfa["start_state"]
    for ch in word:
        if ch not in dfa["alphabet"]:
            return False
        cur = dfa["transitions"].get(cur,
     {}).get(ch)
        if cur is None:
            return False
    return cur in dfa["accept_states"]

def generate_exhaustive_strings(alphabet:
     List[str], max_len: int) -> List[str]:
    strings = [""]
    for L in range(1, max_len + 1):
        for tup in product(alphabet, repeat=L):
            strings.append("".join(tup))
    return strings

def generate_random_strings(alphabet: List[str],
     n: int, max_len: int) -> List[str]:
    out = set()
    while len(out) < n:
        L = random.randint(0, max_len)
        out.add("".join(random.choice(alphabet)
     for _ in range(L)))
    return list(out)

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

def main():
    table_files = [
        os.path.join(TABLES_DIR, f)
        for f in os.listdir(TABLES_DIR)
        if f.endswith(".json")
    ]

    summary = []

    for table_path in table_files:
        base = os.path.splitext(os.path.basename
        (table_path))[0]
        print(f"[VALIDATING] {base}")

        report = {
            "id": base,
            "schema_valid": False,
            "schema_error": None,
            "totality_missing": [],
            "totality_invalid_targets": [],
            "tested_strings": 0,
            "counterexamples": [],
            "verdict": ""
        }

        try:
            dfa = load_json(table_path)
        except Exception as e:
            report["schema_error"] =
     f"json_load_error: {e}"
```

```
        report["verdict"] = "invalid_json"
        save_json(os.path.join(VALID_DIR,
base + "_report.json"), report)
        continue

    ok, reason = is_valid_schema(dfa)
    report["schema_valid"] = ok
    report["schema_error"] = None if ok else
reason

    if not ok:
        report["verdict"] = "schema_invalid"
        save_json(os.path.join(VALID_DIR,
base + "_report.json"), report)
        continue

    missing, invalid = check_totality(dfa)
    report["totality_missing"] = missing
    report["totality_invalid_targets"] =
invalid

    # Locate matching raw file for regex
    regex = None
    for rf in os.listdir(RAW_DIR):
        if rf.startswith(base):
            raw =
load_json(os.path.join(RAW_DIR, rf))
            regex = raw.get("regex")
            break

    if regex is None:
        report["verdict"] =
"no_ground_truth_regex"
        save_json(os.path.join(VALID_DIR,
base + "_report.json"), report)
        continue

    try:
        pattern = re.compile(f"^({regex})$")
    except Exception as e:
        report["verdict"] =
f"regex_compile_error: {e}"
        save_json(os.path.join(VALID_DIR,
base + "_report.json"), report)
        continue

    exhaustive = generate_exhaustive_strings
    (dfa["alphabet"], MAX_EXHAUSTIVE_LEN)
    random_tests =
generate_random_strings(dfa["alphabet"],
N_RANDOM, MAX_RANDOM_LEN)
    tests = exhaustive + [s for s in
random_tests if s not in exhaustive]

    for w in tests:
        dfa_accept = simulate_dfa(dfa, w)
        regex_accept = pattern.fullmatch(w)
is not None
        report["tested_strings"] += 1

        if dfa_accept != regex_accept:
            report["counterexamples"].append({
                "string": w,
                "dfa_accepts": dfa_accept,
                "regex_accepts": regex_accept
            })
            if len(report["counterexamples"])
>= MAX_COUNTEREXAMPLES:
                break

    if missing or invalid:
        report["verdict"] = "not_total"
    elif report["counterexamples"]:
        report["verdict"] = "incorrect"
    else:
        report["verdict"] = "likely_correct"

    save_json(os.path.join(VALID_DIR, base +
"_report.json"), report)

    summary.append({
        "id": base,
        "schema_valid": ok,
        "totality_ok": not (missing or
invalid),
        "tested_strings":
report["tested_strings"],
```

```
            "num_counterexamples":
     len(report["counterexamples"]),
            "verdict": report["verdict"]
        })

    # ============================
    # WRITE SUMMARY TABLES
    # ============================

    with open(os.path.join(VALID_DIR,
     "summary.csv"), "w", newline="",
     encoding="utf-8") as f:
        writer = csv.DictWriter(
            f,
            fieldnames=["id", "schema_valid",
     "totality_ok",
                        "tested_strings",
     "num_counterexamples", "verdict"]
        )
        writer.writeheader()
        for row in summary:
            writer.writerow(row)

    save_json(os.path.join(VALID_DIR,
     "summary.json"), summary)

    print("[DONE] DFA validation completed.")

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

F Appendix: Experiment Execution Code

Experiment Execution Code. We provide the full experiment execution code used to query language models and collect DFA outputs. The script is model-agnostic and was used uniformly across all evaluated models and prompting strategies, differing only in the model identifier and prompt templates. It enforces deterministic decoding, robust retry logic, strict JSON extraction, and complete logging of raw outputs, parsed DFAs, and metadata. All runs write outputs regardless of success or failure, enabling reproducible analysis of both correctness and failure modes.

Strengths and Limitations. The execution pipeline ensures reproducibility through deterministic decoding and uniform evaluation settings, and robustness through retries and structured output validation. However, it does not attempt to correct or post-process invalid DFAs, and relies on downstream automated and human validation for semantic correctness. Timeout events and malformed outputs are treated as unsuccessful attempts, reflecting practical deployment constraints rather than purely theoretical capability.

F.1 Experiment Execution Code

```
#!/usr/bin/env python3
"""
run_experiment_tot.py

General Tree-of-Thought (ToT) experiment runner
     for LLM-based DFA construction.

Features:
- Model-agnostic (OpenAI / API-based LLMs)
- Deterministic decoding (temperature = 0)
- Robust JSON extraction and retry logic
- Strict output logging (raw, parsed, metadata)
- Always writes outputs (success or failure)
- Reproducible and configurable via constants

This script is used uniformly across models and
     prompting strategies by
changing only the MODEL identifier and prompt
     templates.
"""

import os
import json
import time
import re
from datetime import datetime
from typing import Optional
from openai import OpenAI, OpenAIError

# ============================
# CONFIGURATION (MODEL-AGNOSTIC)
# ============================

MODEL = os.environ.get("LLM_MODEL", "gpt-5.1")
TEMPERATURE = 0.0
MAX_TOKENS = 4000

DATASET_PATH = "data/analysis_data.json"

PROMPT_FILES = {
    "direct":     "prompts/tot_direct.txt",
    "minimal":    "prompts/tot_minimal.txt",
    "derivative": "prompts/tot_derivative.txt",
    "thompson":   "prompts/tot_thompson.txt",
}

OUTPUT_ROOT = "outputs/tot_experiments"

RETRIES = 3
RETRY_SLEEP = 1.5
RATE_LIMIT_SLEEP = 0.3

# ============================
# INIT
# ============================

client = OpenAI()

TIMESTAMP =
     datetime.utcnow().strftime("%Y%m%dT%H%M%
SZ")
OUTPUT_RAW_DIR   = os.path.join(OUTPUT_ROOT,
     MODEL, "raw")
OUTPUT_TABLE_DIR = os.path.join(OUTPUT_ROOT,
     MODEL, "tables")
OUTPUT_META_DIR  = os.path.join(OUTPUT_ROOT,
     MODEL, "meta")

for d in [OUTPUT_RAW_DIR, OUTPUT_TABLE_DIR,
     OUTPUT_META_DIR]:
    os.makedirs(d, exist_ok=True)

PROJECT_ROOT =
     os.path.abspath(os.path.join(os.path.
dirname(__file__), ".."))

# ============================
# UTILITIES
# ============================

def now_ts():
    return datetime.utcnow().strftime("%Y%m%dT%H%M
%SZ")

def safe_filename(s: str) -> str:
    return re.sub(r"[^0-9A-Za-z._-]", "_",
     s)[:200]

def load_json(path):
    with open(path, "r", encoding="utf-8") as f:
        return json.load(f)

def load_prompt(path):
    with open(path, "r", encoding="utf-8") as f:
```

```
        return f.read()

def save_json(path, obj):
    with open(path, "w", encoding="utf-8") as f:
        json.dump(obj, f, indent=2,
     ensure_ascii=False)

def try_parse_json(s: str) -> Optional[dict]:
    if not s:
        return None
    s = re.sub(r"```(?:json)?", "", s,
     flags=re.IGNORECASE).strip("` \n")
    try:
        return json.loads(s)
    except Exception:
        pass
    m = re.search(r"\{[\s\S]*\}", s)
    if m:
        try:
            return json.loads(m.group(0))
        except Exception:
            pass
    return None

# ============================
# TOKEN DISPATCH
# ============================

def completion_kwargs():
    if MODEL.startswith("gpt-5"):
        return {"max_completion_tokens":
     MAX_TOKENS}
    return {"max_tokens": MAX_TOKENS}

# ============================
# MODEL ACCESS CHECK
# ============================

def check_model_access():
    try:
        r = client.chat.completions.create(
            model=MODEL,
            messages=[{"role": "user", "content":
     "ping"}],
            temperature=0.0,
            **completion_kwargs()
        )
        actual = getattr(r, "model", None)
        if actual is None or not
     actual.startswith(MODEL.split("-")[0]):
            raise RuntimeError(f"Model mismatch:
     requested={MODEL}, actual={actual}")
    except OpenAIError as e:
        raise RuntimeError(f"Model access failed:
     {e}")

# ============================
# PROMPTING
# ============================

def build_messages(template, regex, alphabet):
    filled = (
        template
        .replace("{{REGEX}}", regex)
        .replace("{{ALPHABET}}",
     json.dumps(alphabet))
    )
    return [
        {
            "role": "system",
            "content": (
                "You are an expert in formal
     languages and automata. "
                "Internally reason as needed, but
     OUTPUT ONLY the DFA JSON."
            )
        },
        {"role": "user", "content": filled}
    ]

def call_model(messages):
    r = client.chat.completions.create(
        model=MODEL,
        messages=messages,
        temperature=TEMPERATURE,
        **completion_kwargs()
    )
```

```
    text = r.choices[0].message.content or ""
    meta = {
        "requested_model": MODEL,
        "actual_model": getattr(r, "model", None),
        "finish_reason":
     r.choices[0].finish_reason,
        "usage": getattr(r, "usage", None)
    }
    return text, meta

# ============================
# CORE LOOP
# ============================

def run_single_branch(entry, branch,
     prompt_template):
    regex_id = entry["id"]
    regex = entry["regex"]
    alphabet = entry.get("alphabet", [])
    ts = now_ts()

    messages = build_messages(prompt_template,
     regex, alphabet)

    raw_text, meta, parsed = "", None, None
    attempts = 0

    while attempts < RETRIES:
        attempts += 1
        try:
            raw_text, meta = call_model(messages)
        except Exception as e:
            meta = {"error": str(e)}
            time.sleep(RETRY_SLEEP)
            continue

        parsed = try_parse_json(raw_text)
        if parsed:
            break

        messages.append({
            "role": "user",
            "content": "OUTPUT ONLY the DFA JSON.
     No explanations."
        })
        time.sleep(RETRY_SLEEP)

    base =
     safe_filename(f"{regex_id}_{branch}_{ts}")

    save_json(os.path.join(OUTPUT_RAW_DIR, base +
     ".json"), {
        "id": regex_id,
        "branch": branch,
        "regex": regex,
        "alphabet": alphabet,
        "raw_output": raw_text,
        "attempts": attempts,
        "meta": meta,
        "model": MODEL,
        "timestamp": ts
    })

    save_json(os.path.join(OUTPUT_META_DIR, base
     + "_meta.json"), meta)

    table_obj = parsed if parsed else {
        "error": "GENERATION_FAILED",
        "reason": "Invalid or non-JSON output",
        "regex_id": regex_id,
        "branch": branch,
        "model": MODEL,
        "attempts": attempts
    }

    save_json(os.path.join(OUTPUT_TABLE_DIR, base
     + ".json"), table_obj)

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

def main():
    check_model_access()
    dataset =
     load_json(os.path.join(PROJECT_ROOT,
     DATASET_PATH))
```

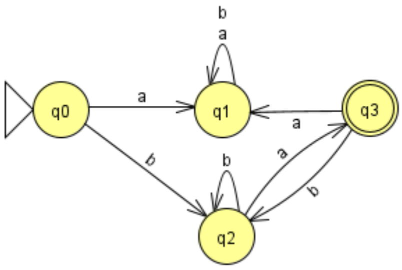


Figure 5: DFA generated via direct construction for $L = b(a \mid b)^*ab$.

```
    prompts = {
        k: load_prompt(os.path.join(PROJECT_ROOT,
     v))
        for k, v in PROMPT_FILES.items()
    }

    for entry in dataset:
        for branch, tmpl in prompts.items():
            run_single_branch(entry, branch, tmpl)
            time.sleep(RATE_LIMIT_SLEEP)

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

G Appendix: Common Problems Identified from LLM-Generated DFAs

This appendix summarizes recurring error patterns observed in deterministic finite automaton (DFA) constructions produced by large language models (LLMs) across different prompting strategies. Each problem type is illustrated using a concrete example and a corresponding DFA diagram. The goal is diagnostic: to characterize systematic construction failures rather than isolated errors.

Summary of Identified Problem Types

Problem 1: Disregard for Kleene Star Semantics

Figure 5 shows a DFA generated via direct construction for the language $L = b(a \mid b)^*ab$. The model correctly identifies the terminal pattern `ab` but fails to account for the Kleene star $(a \mid b)^*$, resulting in a DFA that accepts strings ending in `ba` rather than enforcing the required suffix `ab`. This error reflects an incomplete semantic interpretation of unbounded repetition under the Kleene star.

Problem 2: Violation of Brzozowski Derivative Semantics

Construction method: Derivative-based construction. Figure 6 shows a derivative-based DFA generated for the language $L = b(a \mid b)^*ab$. The construction fails to normalize semantically equivalent Brzozowski derivatives and incorrectly reintroduces consumed prefixes. As a result, distinct residual languages are treated as separate states, violating the formal semantics of regular expression derivatives and yielding an incorrect DFA.

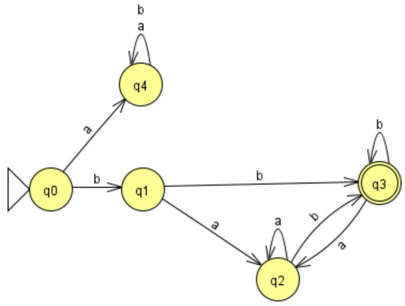


Figure 6: Derivative-based DFA generated for $L = b(a \mid b)^*ab$.

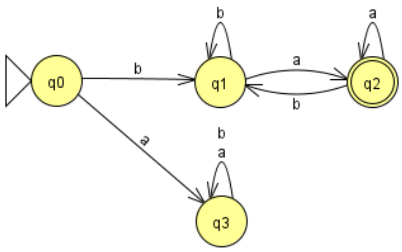


Figure 7: DFA produced prior to minimization for $L = b(a \mid b)^*ab$.

Problem 3: Errors in Initial DFA Construction

Construction method: Minimization-based construction. Figure 7 shows a DFA produced prior to minimization for the language $L = b(a \mid b)^*ab$. The initial DFA is incorrectly constructed due to faulty state semantics and transition assignments. Subsequent minimization amplifies these errors by merging states that should remain distinct, resulting in an invalid minimized DFA.

Problem 4: Over-Acceptance of Strings

Construction method: Derivative-based construction. Figure 8 shows an over-accepting DFA generated for the language

$$L = ((a \mid b)^*(aa^*bb^* \mid b(b \mid ab)^*aaa^*bb^*) \\ (aa^*bb^*)^* \\ \mid (a \mid b)^*b(b \mid ab)^*a)aa^*.$$

The constructed DFA accepts strings that do not belong to the target language: formally, there exists a string $w \in \Sigma^*$ such that $w \in L(\text{DFA})$ while $w \notin L$. This over-acceptance arises when syntactically similar but semantically distinct deriva-

Table 8: Summary of Common DFA Construction Problems

Problem	**Identified Issue**
Problem 1	Disregard for Kleene star semantics
Problem 2	Violation of Brzozowski derivative semantics
Problem 3	Errors in initial DFA construction before minimization
Problem 4	Over-acceptance of strings outside the target language
Problem 5	Failure to preserve constraints under concatenation
Problem 6	Introduction of redundant or unreachable states

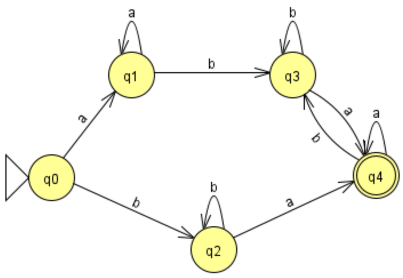


Figure 8: Over-accepting DFA generated for the target language.

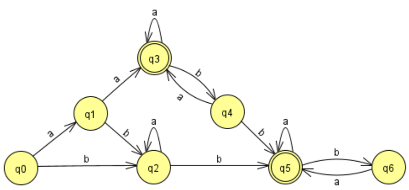


Figure 9: DFA failing to preserve constraints under concatenation.

tives are merged without establishing true language equivalence. In this example, strings of the form bb^*a are incorrectly accepted.

Problem 5: Failure to Preserve Constraints under Concatenation

Construction methods: Direct and Thompson constructions. Figure 9 shows a DFA that fails to preserve constraints under concatenation for the language

$$L = ((a^*ba^*b \mid (a^*ba^*b \mid a^*a)(b \mid a)^*a))a^*.$$

The constructed DFA correctly enforces individual sub-constraints such as a^*ba^*b and a^*a in isolation. However, when these constraints are composed via concatenation, global boundary conditions are not preserved.

Problem 6: Introduction of Redundant or Unreachable States

Construction methods: Direct, Thompson, derivative-based, and minimization-based constructions.

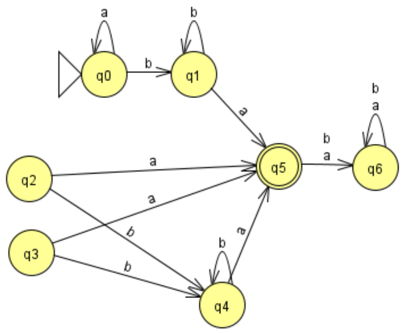


Figure 10: DFA containing redundant and unreachable states.

Figure 10 shows a DFA containing redundant and unreachable states for the language

$$L = ((a^*b \mid (a^*a)a^*b \mid (a^*b)b^*b \\ \mid ((a)^*ba \mid (a^*a)a^*ba \mid (a^*b)b^*ba) \\ (aa \mid b)^*a)(a \mid b) \mid (a^*b)b^*)a.$$

The constructed DFA contains multiple states that are unreachable from the start state and therefore do not contribute to language recognition. In some examples, some of these states are accepting despite having no incoming paths. This indicates that the model introduces states corresponding to syntactic components of the regular expression without performing reachability or usefulness analysis, resulting in redundant and semantically irrelevant states.

For an overall view, Figure 11 shows the common mistakes for the Language L_9 under the ToT methods.

H Appendix : Derivative-Based DFA Output for L_9

This appendix reports the exact output produced by the derivative-based DFA construction for language L_9. The content is shown verbatim as returned by the model, without normalization, correction, or post-processing.

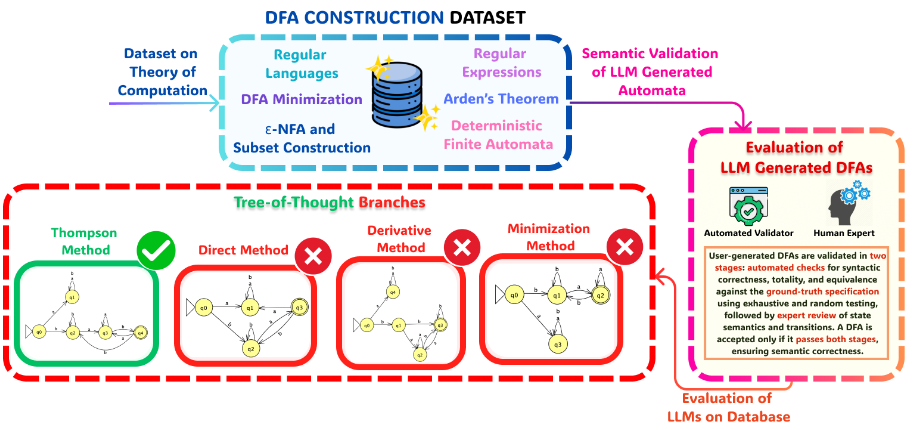


Figure 11: DFA construction dataset and evaluation pipeline, with representative DFA outputs generated by ToT prompting following construction strategies for the language $L_9 = b(a+b)^*ab$.

H.0.1 Model Output (Derivative Construction for L_9)


```
{
  "method": "derivative",
  "derivatives": {
    "q0": {
      "regex": "b(a|b)*ab",
      "nullable": false
    },
    "q1": {
      "regex": "(a|b)*ab",
      "nullable": false
    },
    "q2": {
      "regex": "b(a|b)*ab+(a|b)*b",
      "nullable": false
    },
    "q3": {
      "regex": "(a|b)*ab+(a|b)*",
      "nullable": true
    },
    "q4": {
      "regex": "\phi",
      "nullable": false
    }
  },
  "dfa": {
    "states": [
      "q0",
      "q1",
      "q2",
      "q3",
      "q4"
    ],
    "alphabet": [
      "a",
      "b"
    ],
    "start_state": "q0",
    "accept_states": [
      "q3"
    ],
    "transitions": {
      "q0": {
        "a": "q4",
        "b": "q1"
      },
      "q1": {
        "a": "q2",
        "b": "q3"
      },
      "q2": {
        "a": "q2",
        "b": "q3"
      },
      "q3": {
        "a": "q2",
        "b": "q3"
      },
      "q4": {
        "a": "q4",
        "b": "q4"
      }
    }
  }
}
```


I Appendix: Determinism and Randomness Control

All experiments were designed to be as deterministic and reproducible as possible, subject to the constraints of black-box LLM APIs.

Decoding determinism. For all models and prompting strategies, decoding temperature was fixed to 0.0, and no stochastic sampling parameters were modified. This ensures that, for a given prompt and model endpoint, the generation process is deterministic to the extent supported by the provider.

Prompt determinism. Each prompt template is fixed and version-controlled in the repository. For Tree-of-Thought and Chain-of-Thought settings, a fixed, strategy-specific prompt template is reused across all runs. No adaptive prompt modification or dynamic hint generation is performed during a single model invocation.

Retry policy. To mitigate transient API failures and formatting errors, each query is retried up to a fixed number of attempts using an identical prompt. Retries are only triggered when the output is invalid (e.g., non-JSON or schema-violating) and do not introduce additional randomness or alternative prompts.

Validation randomness control. Behavioral equivalence testing uses a hybrid validation strategy consisting of: (i) exhaustive enumeration of all strings up to a fixed maximum length, and (ii) randomized testing over longer strings. All randomized testing is performed with a fixed random seed, ensuring that validation results are fully reproducible.

Non-determinism across API calls. Despite the above controls, repeated API calls to the same model may still produce different outputs due to undocumented provider-side nondeterminism (e.g., model updates, inference-time optimizations, or distributed serving). Such variation is treated as an inherent property of deployed LLM systems rather than experimental noise.

Reporting policy. All reported results correspond to the actual outputs returned by the API during evaluation and are not post-selected. Timeouts, formatting failures, and invalid outputs are logged explicitly and counted according to the evaluation protocol described in the main paper.

J Appendix: Runtime Environment

All experiments were executed using custom Python scripts on a local workstation. The implementation was written in Python (version 3.10+) and relied exclusively on standard libraries and official model APIs. No proprietary or undocumented tooling was used.

All LLM interactions were performed via API calls using deterministic decoding settings (temperature set to 0). Experiments were conducted sequentially with explicit rate-limiting and retry logic to ensure stability and reproducibility. Each model invocation was stateless: no conversational context was carried across calls. This includes the hint-based framework, where the full problem specification was re-provided at every stage.

Model outputs, metadata (including token usage and finish reasons), parsed DFA tables, and validation reports were logged to disk in structured JSON format. The runtime environment enforced strict output handling: results were recorded for every attempt, including malformed outputs, timeouts, or generation failures.

Validation and evaluation were performed offline using a separate automated pipeline that exhaustively and randomly tested DFA behavior against the ground-truth regular expressions. All randomness used in validation (e.g., random string generation) was controlled via fixed random seeds.

No fine-tuning, model-side configuration changes, or system-level optimizations were applied. Differences in performance therefore reflect intrinsic model behavior under identical runtime conditions rather than environmental variability.