SAME TASK, DIFFERENT REPRESENTATION: WHEN LLM SUCCESS FAILS TO TRANSFER

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) often solve tasks using familiar output representations, such as implementing Boolean logic in Verilog/Python or expressing chess moves in standard notation. Yet users may need the same task solved in a different representation, requiring models to apply acquired skills under new constraints. Does demonstrated success transfer when the required output representation changes, and can additional reasoning recover lost success? We thus introduce *Paired Representation Robustness* (PRR), which measures the fraction of problems a model already solves that remain correct under an alternate output representation. The semantic task stays fixed, and the alternate representation's rules are explicitly supplied. We evaluate five LLMs on 500 Boolean logic problems (Python code $\rightarrow$ Minecraft Redstone circuits) and 1,000 chess puzzles (standard move notation $\rightarrow$ a synthetic cipher notation, Syn-Chess). With CoT, and additional reasoning disabled, GPT-5.6-Sol solves 98.8% of logic problems in Python but retains only 1.01% of those successes in Redstone; it identifies the correct first move in 86.6% of chess puzzles but retains only 0.69% under Syn-Chess. On a held-out subset of 100 Syn-Chess puzzles, adaptive reasoning raises first-move PRR from below 4% to approximately 79% for both GPT-5.6-Sol and GPT-5.6-Luna, at higher inference cost. These findings reveal a barrier to applying demonstrated skills under new output requirements: transfer can collapse even with explicit CoT prompting, while adaptive reasoning recovers much of the lost success at over $10\times$ the baseline inference cost. Output representation thus emerges as a consequential factor in both the reliability and computational cost of LLM problem solving.

1 INTRODUCTION

Large language models (LLMs) have demonstrated strong problem-solving capability on reasoning tasks (Maslej et al., 2025). Typically these problems asks a model to solve an underlying *semantic task* and express its answer in a required *output representation*. For example, the task might be to determine a chess move, while the representation specifies how that move must be written. This distinction is important, as training on internet text may encourage LLMs to associate semantic tasks with familiar representations, such as chess moves with standard UCI notation or Boolean logic implementation with Verilog/Python code. Success under these conventions in standard benchmarks therefore leaves open whether the same ability transfers when the required output representation changes. This transfer is important because it shows evidence of a desirable property of intelligence: applying acquired skills under new constraints (Chollet, 2019; Lake et al., 2016).

This also matters in practice. Consider calculating total sales after excluding refunded orders when the data is spread across sheets: An LLM may solve the task in Python, while a user needs a spreadsheet formula that recalculates as the workbook is updated. Consider another example in a less studied domain of digital circuits: implementing the Boolean function $f(A, B) = A \vee B$. An LLM may readily produce Python code, while a Minecraft player needs a *Redstone* circuit to open a door when either of two levers is activated. Redstone is Minecraft's system of components for transmitting and manipulating signals (speedycube64; Wong, 2021). In each case, the model has already demonstrated the ability to solve the semantic task. The question is whether it can carry that success into the representation the user needs. If success is lost, can additional reasoning recover it, and at what cost? This motivates our central question:

How well does an LLM's success transfer across output representations, and how does additional inference compute affect this transfer?

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

We introduce *Paired Representation Robustness* (PRR) to measure this ability. For each model, we identify the problems it solves in a standard (*native*) representation, then evaluate those exact problems in an *alternate* representation with its complete rules supplied. PRR is the fraction of native successes that remain correct. Prior work studies sensitivity to formats and task conventions (Sclar et al., 2024; Fatemi et al., 2024; Wu et al., 2024) and correctness across paired variants (Li et al., 2024). Our distinctive focus is the loss of demonstrated success under new output requirements

We study two domains, illustrated in Figure 1. In Boolean logic, models implement the same function in Python and Minecraft Redstone. In chess tactics, models solve the same forced-mate puzzle using standard UCI move notation and a synthetic cipher notation, *Syn-Chess*. For example, UCI expresses a move from square g5 to square e6 as `g5e6`; Syn-Chess requires encoding the move through a different set of explicit rules. In each case, answers are checked against a common semantic target: the Boolean function's truth table or the chess puzzle's accepted solution moves.

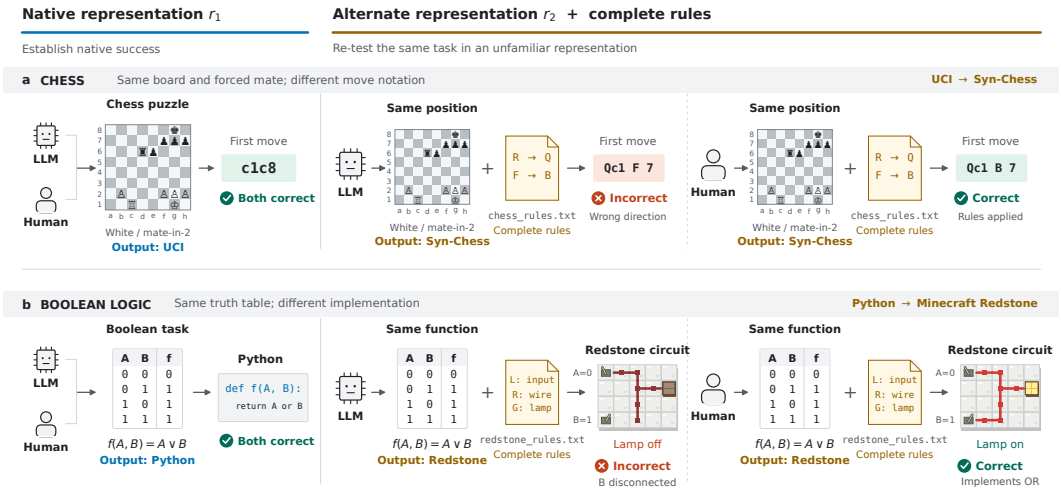

Figure 1: **Same semantic task, different output representation.** Top: moves for same chess position are expressed in UCI (standard notation) or Syn-Chess notation. Bottom: same Boolean OR function is implemented in Python or Minecraft Redstone.

Using these two domains, we construct a benchmark of 500 Boolean logic problems and 1,000 forced-mate chess puzzles and evaluate five LLMs. We first test transfer with additional reasoning disabled. Despite strong native performance, success often fails to transfer: GPT-5.6-Sol solves 98.8% of logic problems in Python but retains only 1.01% of those successes in Redstone. It identifies the correct first move in 86.6% of chess puzzles using UCI but retains only 0.69% under Syn-Chess. The difficulty also depends on the model and representation: Claude Opus 4.8 retains 66.71% of its native chess successes under Syn-Chess, but only 1.65% of its native logic successes in Redstone.

These losses lead us to ask whether the models can recover their demonstrated success by spending more inference time compute on each answer. On a held-out subset of 100 Syn-Chess puzzles, we use a router to select among four reasoning effort levels for GPT-5.6-Sol, GPT-5.6-Luna and Qwen3.8-27B. First-move PRR rises from 3.57% to 79.76% for Sol, from 1.79% to 78.57% for Luna, and from 0.00% to 14.70% for Qwen3.8-27B, albeit at greater than 10x higher inference cost.

Our contributions are:

- **Evaluation:** We introduce PRR to measure retained correctness on the exact problems each model already solves when the required output representation changes.
- **Benchmark settings and analysis:** We instantiate PRR in Boolean logic and chess with explicit representation rules and semantic correctness checks. A simple complexity analysis bounds the increase in computation caused by a reversible output change for a family of tasks, and shows that it can exceed a budget that sufficed natively.
- **Empirical findings:** Across five models, we identify severe transfer failures in Redstone and substantial variation in Syn-Chess at the base setting. Experiments with three models further quantify recovery through adaptive reasoning and its inference cost, complemented by error analysis and a human Syn-Chess usability study.

2 Related Works

Prior work examines sensitivity to task representation (McCoy et al., 2023; Sclar et al., 2024; Fatemi et al., 2024), performance under altered task conventions (Wu et al., 2024; O'Brien et al., 2025; Tang et al., 2023), and consistency or correctness across paired variants (Ribeiro et al., 2020; Elazar et al., 2021; Wang et al., 2023; Li et al., 2024). PRR builds on these perspectives to test the transfer of demonstrated problem-solving ability when the semantic task is fixed but the required output representation changes.

Task representation sensitivity. Equivalent multiplication inputs, prompt formats, and graph encodings can yield substantially different model accuracy (McCoy et al., 2023; Sclar et al., 2024; Fatemi et al., 2024). This sensitivity extends across input and output representations: planning evaluations compare natural-language and PDDL interfaces and test semantics-preserving name obfuscation (Valmeekam et al., 2023). MultiPL-E evaluates generated implementations across programming languages using translated unit tests (Cassano et al., 2022). Its prompts specify the task in the target language but do not supply that language's full rules. PRR builds on these representation-sensitivity studies by holding the task input fixed, supplying the alternate output representation's rules, and testing whether demonstrated problem-solving success transfers.

Altered task conventions. Altered task conventions test whether models can apply rules or symbol meanings that differ from familiar ones. *Reasoning or Reciting?* evaluates altered task conditions, including task-preserving coordinate and syntactic remappings, with explicit rules and comprehension checks (Wu et al., 2024). Related studies rename logical relations, rebind Python identifiers, or replace class labels with arbitrary symbols (Tang et al., 2023; Miceli-Barone et al., 2023; Wei et al., 2023b;a). MatheMagic compares rule induction from examples with deduction under explicit rules, combining arithmetic recodings with transformations that change the computation (O'Brien et al., 2025). Unlike conditions that change the semantic target or require rule induction, PRR holds the task input (the specified Boolean function or chessboard state) and semantic correctness criterion fixed and supplies the alternate rules. It therefore tests transfer to new output requirements without also requiring a different semantic solution or discovery of an unspecified rule. Furthermore, while much of this prior work focuses on arithmetic, programming, and NLP benchmarks, PRR broadens evaluation to less conventional settings: Boolean Logic to Redstone and Chess to Syn-Chess.

Paired robustness evaluation. Paired evaluations compare agreement or correctness across original and modified instances. CheckList tests behavioral invariance, ParaRel measures consistency and all-pattern correctness, and ReCode evaluates executable robustness (Ribeiro et al., 2020; Elazar et al., 2021; Wang et al., 2023). GSM-Plus reports aggregate performance drop (PDR) and jointly correct pairs (ASP) under math-question perturbations (Li et al., 2024). Table 6 in Appendix B summarizes the evaluation designs. PRR reports semantic correctness under new output requirements only on each model's native-correct instances. Its complementary collapse rate measures losses of demonstrated success without offsets from gains on other instances.

3 Paired Representation Robustness

As discussed earlier, this work studies whether an LLM's demonstrated problem-solving ability transfers when the required output representation changes. We first define task-preserving pairs and their output correctness criterion, then describe the evaluation protocol wherein we measure Paired Representation Robustness (PRR) to articulate the transfer ability.

3.1 Problem Setup

A problem specifies a semantic task z-what must be solved-and an output representation r-how the answer must be expressed. Two problems form a *task-preserving pair* when they share same semantic task and correctness criterion but require different output representations. Both representations must admit a correct answer. For example, Python programs and Redstone circuits can implement the same Boolean function; correctness requires matching its truth table on every input.

Let $D_r(y; z)$ interpret an output y in representation r, using task context when needed. Invalid outputs map to a failure symbol $\perp$. Let $V(z, a)$ equal one when the semantic answer a is accepted for task z, and zero otherwise, with $V(z, \perp) = 0$. This criterion allows multiple correct answers, such as different accepted mating continuations.

Let $M_\pi(z, r)$ denote the output of model M under inference policy π, given the task, required representation, and its rules. A policy specifies how inference is performed, including the reasoning

effort used. We now define correctness as

$$C_M(z, r; \pi) = V(z, D_r(M_\pi(z, r); z)) . \tag{1}$$

The same semantic acceptance criterion V applies to both representations. Table 1 summarizes the pairs used in our evaluation.

Table 1: **Task-preserving evaluation pairs.** Each pair shares the same semantic task and correctness target denoted by ❄, while changing the required output representation.

Domain	Semantic task z ❄	Reference target $g(z)$ ❄	Native representation r_1 ⇄	Alternate representation r_2 ⇄
Boolean logic	Implement Boolean function f	Truth table of f	Python code	Minecraft Redstone circuit
Chess	Find a forced mating continuation	Accepted solution under the scoring criterion	UCI move notation	Syn-Chess notation

3.2 Paired Evaluation Protocol

We evaluate each model using three steps. The base inference policy, denoted π_0, uses reasoning effort `None`. Thus, disabling additional inference scaling does not remove prompted reasoning.

Step 1: Alternate-representation control. We evaluate the alternate representation without supplying its rules, using the base policy π_0. This measures success without explicit rule instruction.

Step 2: Native competence. Let $\mathcal{Z}$ denote the benchmark's task set. We evaluate each task in the native representation r_1 under π_0 and retain the tasks solved correctly:

$$\mathcal{Z}_M = \{z \in \mathcal{Z} \mid C_M(z, r_1; \pi_0) = 1\} . \tag{2}$$

This native competence set stays fixed across subsequent inference policies. Every paired evaluation therefore tests tasks the same model has already solved at the base setting.

Step 3: Paired Representation Robustness. We supply the complete rules of the alternate representation r_2 and evaluate the tasks in $\mathcal{Z}_M$. For a nonempty competence set, *Paired Representation Robustness* (PRR) is the fraction of native successes retained under inference policy π:

$$\mathrm{PRR}_{r_1 \to r_2}(M; \pi) = \frac{1}{|\mathcal{Z}_M|} \sum_{z \in \mathcal{Z}_M} C_M(z, r_2; \pi). \tag{3}$$

A value of one means all native successes are retained; zero means none is retained. Our base evaluation uses $\pi = \pi_0$. Inference-scaling experiments change the alternate-representation policy while retaining the competence set established in Step 2.

3.3 Why Representation Transfer Can Be Difficult

Changing the output representation can introduce additional work even when the semantic task remains fixed. A solver must find the answer, construct its required representation, and coordinate these operations. We express their joint effect on resource cost as $\mathrm{cost}_M(z, r) = \Phi_M(s(z),\, e(z, r),\, h_M(z, r))$. Here, $s(z)$ is the difficulty of finding the semantic answer; $e(z, r)$ is the difficulty of expressing an already-available answer in representation r; and $h_M(z, r)$ describes what solver M must retain, track, or recompute when solving and expressing the answer together. The function Φ_M describes how these factors jointly determine resource cost

Different factors can dominate under different representation changes. We use two elementary bit-string problems in circuit and streaming computation to illustrate these complexity differences. These examples study resource requirements. We choose these settings because their resource requirements can be stated exactly for deterministic computation.

Encoding Dominated Problem. Consider tasks whose outputs are XORs of selected input bits, where XOR returns one when an odd number of its inputs are one. Such a task can be written as $y = Au$, where u is the input bit string and the binary matrix A specifies which input bits contribute to each output. An invertible binary matrix L defines an alternate output representation: the solver must produce $Ly = LAu$, from which y can be recovered exactly.

Let $D(A)$ denote the minimum circuit depth needed to compute Au when each gate combines at most two values. We show that $-D(L^{-1}) \le D(LA) - D(A) \le D(L)$. The upper bound follows by

computing the answer and then encoding it. The additional depth can be unavoidable. For example, suppose the task returns the parity of each of p blocks containing q input bits. If the alternate representation replaces the first output with the parity of all p outputs, that output depends on all pq input bits. The minimum depth increases from $\lceil \log_2 q \rceil$ to $\lceil \log_2(pq) \rceil$. When p and q are powers of two, this increase equals the encoding depth $\log_2 p$ (Appendix A.1).

Coordination Dominated Problem The semantic task is again to return an n-bit string, but its bits now arrive one at a time and cannot be revisited unless stored. Output can only be appended and cannot be read back. The native problem requires the original order; the alternate problem requires reverse order. Copying arriving bits uses $O(1)$ working memory. Reversing an answer already supplied as a read-only array uses $O(\log n)$ additional memory for an index. Yet producing the reversed answer from the stream requires $\Theta(n)$ memory: the last input bit must be output first, so the preceding bits must be retained. Here, coordinating input processing with output construction determines the cost.

These examples establish that a resource budget sufficient for a native representation may be insufficient for an alternate one. Increasing the budget may remove this computational obstruction. They motivate testing whether additional reasoning improves transfer.

4 Experimental Setup

We measure PRR in two domains: Boolean logic and chess tactics. In Boolean logic, models implement the same function in either Python or Minecraft Redstone. In chess, models solve the same forced-mate position using either standard UCI notation or our synthetic notation, Syn-Chess. Table 1 summarizes the corresponding semantic tasks, reference targets, and representations. Below we state -(i) Description of the domains (ii) Scoring criteria for the domains (iii) Models used for evaluation and their corresponding setup.

4.1 Domain Descriptions

Boolean Logic → Minecraft Redstone. Each task specifies a Boolean function. In the native representation, the model implements the function in Python. In the alternate representation, it constructs a two-dimensional Minecraft Redstone circuit as a matrix of block symbols. Levers (`L`) provide input signals, Redstone dust (`R`) carries signals, Redstone torches (`T`) can invert signals, quartz blocks (`Q`) provide structural support, and a lamp (`G`) represents the output. A Python program and a Redstone circuit solve the same task if they realize the same target truth table.

The benchmark contains 500 tasks from two subsets. The *Natural* subset contains 50 manually curated logic circuits, including XNOR gates, half-adders, and D-latches. The *Combinational* subset contains 450 expressions generated by recursively composing AND, OR, XOR, and NOT operators to depth at most two. Each generated expression uses all four variables (A, B, C, D), is non-constant, and has a truth table distinct from every other generated expression.

Chess → Syn-Chess. Each task specifies a chess position with a forced checkmate, the side to move, and the mate depth. The model returns only the attacking side's moves; the evaluator accounts for the defender's possible replies. The native representation uses UCI notation, in which a move is written using its source and destination squares, such as `g5e6`. Syn-Chess changes only how moves are expressed: standard chess rules and board states remain unchanged. First, piece symbols undergo a cyclic substitution, such as $N \to B$ and $B \to R$. Second, the displacement from the source square is expressed relative to the attacking side and rotated by $180°$. A Syn-Chess move therefore contains a ciphered piece symbol, the source square, an inverted direction, and the distance moved.

We construct $1{,}000$ tasks from forced-mate positions in the Lichess puzzle database. Because PRR conditions on native success, we prioritize lower-rated puzzles to obtain sufficiently large competence sets. The benchmark contains 334 Mate-in-2, 333 Mate-in-3, and 333 Mate-in-4 puzzles. A Mate-in-k puzzle requires the attacking side to force checkmate on its k-th move.

4.2 Scoring Criteria

Both representations in each domain are scored against the same representation-independent target.

Boolean logic. Correctness is determined by the target truth table. A generated Python program or Redstone circuit is correct only if it produces the required output for every input assignment.
Redstone evaluation proceeds in two stages. First, deterministic checks test whether the generated

matrix is parseable and satisfies structural requirements such as valid component counts, supported torches, and connected inputs and outputs. Matrices that pass these checks are converted into `.mcfunction` files and executed in Minecraft. A human evaluator then tests every input assignment and compares the observed lamp state with the target truth table.

Chess. UCI and Syn-Chess answers are decoded into chess moves and evaluated against the same precomputed directed acyclic graph (DAG) of optimal mating continuations. We construct this graph using exact bounded minimax: attacker nodes retain every move that achieves the shortest forced mate, while defender nodes retain every reply that maximizes survival. Stockfish 18 cross-checks 20 initial positions at depth 18 but does not determine the graph's edges. For each submitted attacker move, the evaluator decodes the notation at every compatible current node, rejects invalid moves, and expands all retained defender replies. *First-move accuracy*, our primary chess metric, requires the submitted first move to be accepted at the root. *Full-line accuracy* requires exactly k attacker moves for a Mate-in-k puzzle and at least one compatible path ending in checkmate. Figure 2 illustrates this procedure.

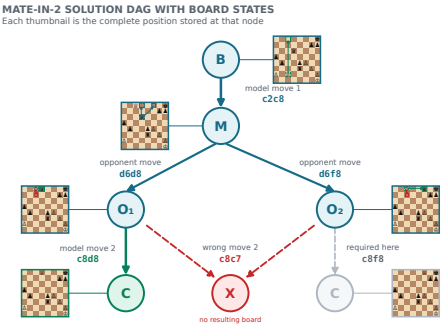

Figure 2: **DAG-based chess evaluation.** The model moves are checked against all valid opponent continuations; full-line credit requires at least one compatible path to checkmate.

4.3 Models and Evaluation Protocol

We evaluate three proprietary models—GPT-5.6-Sol, GPT-5.6-Luna, and Claude Opus 4.8—and two open-weight models—Gemma-4-31B and Qwen3.6-27B. The proprietary models are queried through their official provider APIs. We serve the open-weight models locally using vLLM on a single AMD Instinct MI300X GPU with 192 GB of memory. The Qwen3.8-27B routing experiments in RQ2 use the Groq API.

We apply the three-step protocol from Section 3.2 separately in each domain. Step 1 evaluates every benchmark instance in the alternate representation without supplying its complete rules; we retain only the format information required to parse the response. Step 2 evaluates every instance in the native representation and constructs the model-specific competence set $\mathcal{Z}_M$. Step 3 supplies the complete rules of the alternate representation and evaluates exactly the instances in $\mathcal{Z}_M$. Consequently, the denominator of PRR is the number of instances solved by that same model in Step 2, rather than the full benchmark or a competence set shared across models.

Each model produces one deterministic generation for every instance in each applicable step. We use the same task content and scoring target across the paired native and alternate conditions. The complete prompts for all three steps are provided in Appendix C.

5 Results and Analysis

We organize the results around three questions: whether native success transfers at the base reasoning setting, whether additional inference compute recovers that success, and what the observed failures reveal. RQ1 and RQ3 use the base setting, with reasoning effort set to `none`. RQ2 evaluates adaptive reasoning on a 100-puzzle Syn-Chess subset.

> ***RQ1:*** *Does success in the native representation transfer at the base reasoning setting?*

Table 2 reports the three evaluation steps. Without the alternate representation's rules, Step 1 accuracy is 0.2–0.8% for Redstone and 0% for Syn-Chess. These controls show little success under the alternate output requirements before the rules are supplied.

Transfer to Redstone. All five models solve at least 96.4% of the Boolean tasks in Python. Yet, after receiving the Redstone rules, they retain only 0.41–1.65% of these native successes. Thus, every evaluated model loses more than 98% of its native successes when the same Boolean functions must be implemented in Redstone.

Table 2: **Native competence does not imply robustness to a change in output representation.** Step 1 evaluates the alternate representation without rules. Step 2 reports native accuracy, followed by $|\mathcal{Z}_M|/N$ in parentheses. Step 3 reports PRR, followed by retained/native-correct counts. Step 3 supplies the complete alternate-representation rules; chess results use first-move correctness. Cell shading indicates collapse, with darker red denoting greater collapse.

Model	Logic Gates: Python → Redstone			Chess: UCI → Syn-Chess		
	Control Step 1	**Native** Step 2	**PRR** Step 3	**Control** Step 1	**Native** Step 2	**PRR** Step 3
Proprietary models						
GPT-5.6-Sol	0.8%	98.8% (494/500)	**1.01%** (5/494)	0.0%	86.6% (866/1000)	**0.69%** (6/866)
GPT-5.6-Luna	0.2%	96.8% (484/500)	**0.41%** (2/484)	0.0%	64.1% (641/1000)	**1.72%** (11/641)
Claude Opus 4.8	0.8%	96.8% (484/500)	**1.65%** (8/484)	0.0%	73.9% (739/1000)	**66.71%** (493/739)
Open-weight models						
Gemma-4-31B	0.8%	98.2% (491/500)	**0.81%** (4/491)	0.0%	14.4% (144/1000)	**13.19%** (19/144)
Qwen3.6-27B	0.4%	96.4% (482/500)	**0.41%** (2/482)	0.0%	11.1% (111/1000)	**0.00%** (0/111)

Collapse: 0% 25% 50% 75% 100%

Transfer to Syn-Chess. First-move retention varies substantially across models. GPT-5.6-Sol, GPT-5.6-Luna, and Qwen3.6-27B retain at most 1.72% of their native successes; Gemma-4-31B retains 13.19%; and Claude Opus 4.8 retains 66.71%. Native chess accuracy does not explain this ordering: GPT-5.6-Sol has the highest native accuracy (86.6%) but retains only 0.69%, whereas Claude has lower native accuracy (73.9%) but the highest retention. Across two additional runs, GPT-5.6-Sol retains 1.04% and 0.81%, and GPT-5.6-Luna retains 1.87% and 2.03%, confirming that their low PRR is reproducible.

Takeaway: *At the base reasoning setting, strong native performance does not ensure transfer: all five models retain less than 2% of their native successes in Redstone, while Syn-Chess retention varies substantially across models.*

***RQ2:** How much native success can additional inference compute recover, and at what cost?*

We next examine whether selecting a reasoning effort level for each query improves transfer. Following the accuracy–efficiency perspective of OptimalThinkingBench (Aggarwal et al., 2026), we compare retained correctness and inference cost for GPT-5.6-Sol, GPT-5.6-Luna, & Qwen3.8-27B on 100 Syn-Chess puzzles.

Table 3: **Adaptive reasoning on 100 Syn-Chess puzzles.** The router selects among four reasoning effort levels using a classifier calibrated on 200 separate queries. PRR measures first-move correctness retained on each model's native-correct instances within the 100 puzzles.

Model	Policy	Cost ($)	PRR
GPT-5.6-Sol	`None` reasoning	0.220	3.57%
	Router	2.518	79.76%
	Oracle	2.684	100.00%
GPT-5.6-Luna	`None` reasoning	0.010	1.79%
	Router	0.154	78.57%
	Oracle	0.189	98.21%
Qwen3.8-27B	`None` reasoning	0.071	0.00%
	Router	1.503	14.70%
	Oracle	1.683	58.82%

Routing procedure. We use NVIDIA's prompt task and complexity classifier[1] to assign each query a reasoning score between zero and one. This score estimates the reasoning effort required by the prompt. We fit KMeans with four clusters to the scores of 200 separate calibration syn-chess puzzles. The clusters are ordered by their centroids and mapped to `none`, `low`, `medium`, and `high` LLM reasoning levels, respectively. For each held-out query, we compute its score, assign it to the nearest fitted centroid, and invoke the solver at the corresponding reasoning level. The centroids remain fixed during evaluation.

[1] `https://huggingface.co/nvidia/prompt-task-and-complexity-classifier`

Table 4: **Redstone Step 3 failure diagnostics.** Entries give percentages of each model's native-correct set, with diagnostic counts in parentheses. "Checker-valid" denotes structurally valid circuits that compute the wrong function. Categories are non-exclusive.

Failure diagnostic	GPT-5.6-Sol $n = 494$	GPT-5.6-Luna $n = 484$	Gemma-4-31B $n = 491$	Qwen3.6-27B $n = 482$
Checker-valid	10.32% (51)	0.21% (1)	2.44% (12)	0.41% (2)
Generation failed	3.85% (19)	38.43% (186)	13.24% (65)	78.63% (379)
Empty matrix	4.66% (23)	47.93% (232)	3.67% (18)	1.24% (6)
Component-count mismatch	30.36% (150)	10.12% (49)	46.23% (227)	15.56% (75)
Isolated lever	22.87% (113)	0.00% (0)	2.65% (13)	2.49% (12)
Isolated lamp	0.61% (3)	0.62% (3)	7.33% (36)	0.00% (0)
Torch at an unsupported edge	1.42% (7)	0.00% (0)	0.00% (0)	0.00% (0)
Torch without quartz support	19.23% (95)	0.83% (4)	20.57% (101)	0.62% (3)
Broken circuit	5.67% (28)	1.45% (7)	3.05% (15)	0.62% (3)

Failure percentage: 0% 25% 50% 75% 100%

Comparison policies. The *None reasoning* policy uses `none` for every query. The *Router* selects a level using the procedure above. We also report an *Oracle* router which selects the lowest successful LLM reasoning level.

We evaluate transfer on each model's native-correct instances at `none` reasoning, and report first-move PRR alongside the total cost across 100 puzzles.

Recovery and cost. Table 3 shows substantial recovery for both models. Routing raises GPT-5.6-Sol's PRR from 3.57% to 79.76%, while cost rises from \$0.220 to \$2.518. For GPT-5.6-Luna, PRR rises from 1.79% to 78.57%, while cost rises from \$0.010 to \$0.154. Qwen3.8-27B reaches 14.70% PRR at a cost of \$1.503. The oracle reaches 100% PRR for GPT-5.6-Sol and 98.21% for GPT-5.6-Luna, leaving gaps of 20.24 and 19.64 percentage points relative to routing. Thus, the practical router recovers much of the lost success but remains significantly below the oracle reference. Together, these results show that the severe failure at the base setting is not an absolute limit on transfer in this evaluated Syn-Chess subset.

Takeaway: *On the held-out Syn-Chess subset, adaptive reasoning raises PRR to approximately* 79% *for the GPT models albeit at* 10*x higher inference cost, while also leaving a roughly* 20*-percentage-point gap to the oracle.*

***RQ3:** What fails at the base reasoning setting?*

We examine the base-setting failures in both domains. Appendix E provides a complementary human usability evaluation on a subset of 50 Syn-Chess puzzles.

Redstone: structural validity and functional correctness. Table 4 separates malformed outputs from structurally valid circuits that compute the wrong function. Failure modes differ by model. Qwen3.6-27B fails to produce a usable generation on 78.63% of its Step 3 instances. GPT-5.6-Luna often produces either a failed generation (38.43%) or an empty matrix (47.93%). GPT-5.6-Sol and Gemma-4-31B more often produce circuit-like outputs with invalid topology, including incorrect component counts, disconnected inputs, and unsupported torches.

Structural validity is nevertheless insufficient. GPT-5.6-Sol produces 56 checker-valid circuits, but only five implement the target truth table. The corresponding valid/correct counts are $3/2$ for GPT-5.6-Luna, $16/4$ for Gemma-4-31B, and $4/2$ for Qwen3.6-27B. Manual inspection often finds a simpler function, particularly OR, in place of the target. Thus, passing the structural checks does not establish that a circuit solves the Boolean task.

Syn-Chess: failure across mate depths. Table 5 shows low first-move PRR even on Mate-in-2 puzzles for GPT-5.6-Sol, GPT-5.6-Luna, and Qwen3.6-27B. Their low aggregate retention therefore also occurs on the shortest mating sequences in the benchmark. Gemma-4-31B's retention declines from 19.4% on Mate-in-2 to 5.6% on Mate-in-4.

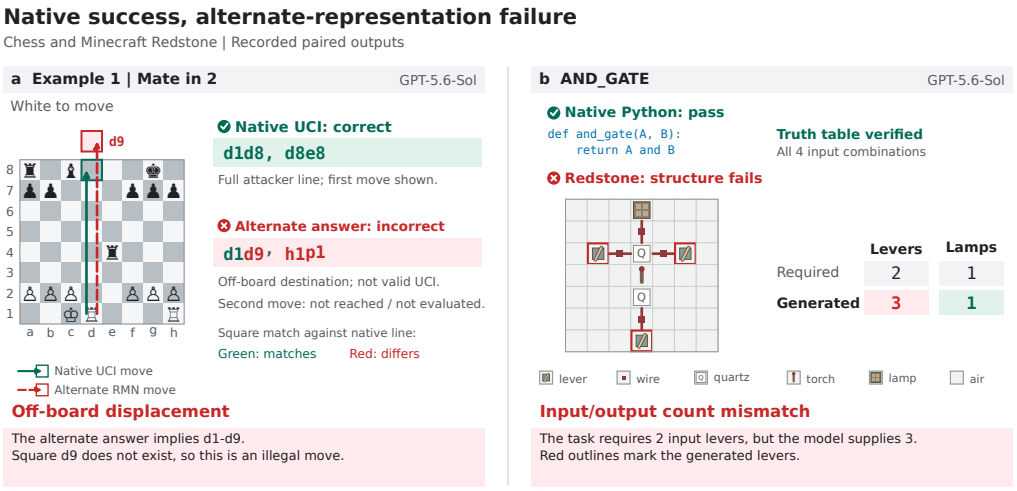

Figure 3: **Recorded paired failures at the base setting.** GPT-5.6-Sol solves both native tasks but (a) generates an off-board first move in Syn-Chess and (b) supplies three levers for a two-input AND gate in Redstone.

Full-line retention is even lower. Among puzzles for which each model produces the complete mating line in UCI, Step 3 retains $0/99$ for GPT-5.6-Sol, $0/1$ for GPT-5.6-Luna, $1/25$ for Gemma-4-31B, and $0/17$ for Qwen3.6-27B. These denominators differ from those for first-move PRR because they condition on native full-line correctness.

Syn-Chess: errors in applying the notation. Direction mismatch is the most frequent component-level error signature. It appears in 92.67%, 72.86%, 36.00%, and 84.68% of the incorrect Step 3 first moves from GPT-5.6-Sol, GPT-5.6-Luna, Gemma-4-31B, and Qwen3.6-27B, respectively. These failures often involve other components as well. After restricting to moves with the correct ciphered piece, source square, distance, and promotion, direction-only errors account for $95/860$ (11.05%), $15/630$ (2.38%), $8/125$ (6.40%), and $11/111$ (9.91%) of failures, respectively. GPT-5.6-Luna also uses the unciphered piece symbol in 29.84% of its failures, while Qwen3.6-27B uses the non-inverted direction in 81.98%. These diagnostics show that the failures involve multiple components of the required representation.

Table 5: **Syn-Chess first-move PRR.** Each cell gives retained/native-correct counts and PRR.

Depth	GPT-5.6 Sol	GPT-5.6 Luna	Gemma 4-31B	Qwen3.6-27B
Mate-in-2	1/314 (0.3%)	4/267 (1.5%)	14/72 (19.4%)	0/50 (0.0%)
Mate-in-3	2/318 (0.6%)	5/244 (2.0%)	3/36 (8.3%)	0/37 (0.0%)
Mate-in-4	3/234 (1.3%)	2/130 (1.5%)	2/36 (5.6%)	0/24 (0.0%)

Collapse: 0% 25% 50% 75% 100%

Figure 3 illustrates GPT-5.6-Sol's native successes and alternate-representation failures in both domains. Further examples appear in Appendix F.

Takeaway: *Base-setting failures include both invalid outputs and incorrect solutions. Syn-Chess failures occur even on Mate-in-2 puzzles and involve multiple notation components*

6 Conclusion

We introduced Paired Representation Robustness (PRR) to investigate whether high benchmark scores in Large Language Models reflect representation-invariant task competence or reliance on familiar data distributions. By keeping the underlying task instances strictly fixed, we tested SOTA models across native representations (Python, UCI) and logically equivalent but unfamiliar alternatives (Redstone, Syn-Chess). The results reveal a near-total performance collapse: despite receiving complete alternate representations rules, models such as GPT-5.6-Sol retain less than 2% of their demonstrated native competence. These findings demonstrate that output representation is a consequential factor in both the reliability and computational cost of LLM problem solving.

AI USE STATEMENT

In this work, we used generative AI tools for assistance in writing proofs, implementing methods, interpreting results, and supporting qualitative and thematic data analysis. We have not used generative AI tools for generating synthetic datasets, developing theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, proposing or refining hypotheses, designing or providing feedback on research methodology or experiments, or cleaning and reformatting datasets, and translation tasks are not applicable to this work. Additionally, we used generative AI tools for creating or editing software code, creating or modifying scientific figures or images, summarizing or analyzing existing literature, sourcing and searching for information, editing the paper to improve readability, and identifying relevant literature. We did not use generative AI tools to format references, suggest the paper's structure, or propose a title or keywords. We have reviewed all AI-assisted work by manually validating the proofs, crosschecking the code and the interpretation of the results, as well as its claims of the data analysis. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

We conducted a small-scale usability evaluation in the chess domain, with two voluntary participants. We recorded their consent and collected their responses through an anonymized Google Form, where in we asked them to solve 50 puzzles. We did not seek formal ethics review.

REPRODUCIBILITY STATEMENT

We describe the paired evaluation protocol in Section 3.2 and the benchmark construction, scoring, evaluated models, and serving setup in Section 4. We provide the evaluation prompts in Appendix C, the theoretical assumptions and proof in Appendix A, and the human evaluation procedure in Appendix E. Our code and generated evaluation assets are not publicly available during review; we commit to releasing them after the review decision, as detailed in Appendix D, which also documents asset sources and licensing.

REFERENCES

Pranjal Aggarwal, Seungone Kim, Jack Lanchantin, Sean Welleck, Jason E Weston, Ilia Kulikov, and Swarnadeep Saha. Optimalthinkingbench: Evaluating over and underthinking in llms. In *International Conference on Learning Representations*, volume 2026, pp. 8886–8918, 2026.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*, 2022.

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

Yanai Elazar, Nora Kassner, Shauli Ravfogel, Abhilasha Ravichander, Eduard Hovy, Hinrich Schütze, and Yoav Goldberg. Measuring and improving consistency in pretrained language models. *Transactions of the Association for Computational Linguistics*, 9:1012–1031, 2021.

Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. Talk like a graph: Encoding graphs for large language models. 2024:43909–43934, 2024.

Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *arXiv preprint arXiv:1604.00289*, 2016.

Qintong Li, Leyang Cui, Xueliang Zhao, Lingpeng Kong, and Wei Bi. Gsm-plus: A comprehensive benchmark for evaluating the robustness of llms as mathematical problem solvers. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2961–2984, 2024.

Nestor Maslej, Loredana Fattorini, Raymond Perrault, Yolanda Gil, Vanessa Parli, Njenga Kariuki, Emily Capstick, Anka Reuel, Erik Brynjolfsson, John Etchemendy, et al. Artificial intelligence index report 2025, 2025.

R Thomas McCoy, Shunyu Yao, Dan Friedman, Matthew Hardy, and Thomas L Griffiths. Embers of autoregression: Understanding large language models through the problem they are trained to solve. *arXiv preprint arXiv:2309.13638*, 2023.

Antonio Valerio Miceli-Barone, Fazl Barez, Shay B Cohen, and Ioannis Konstas. The larger they are, the harder they fail: Language models do not recognize identifier swaps in python. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 272–292, 2023.

Dayyán O'Brien, Barry Haddow, Emily Allaway, and Pinzhen Chen. Mathemagic: Generating dynamic mathematics benchmarks robust to memorization. *arXiv preprint arXiv:2510.05962*, 2025.

Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of nlp models with checklist. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pp. 4902–4912, 2020.

Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. In *International Conference on Learning Representations*, volume 2024, pp. 25055–25083, 2024.

speedycube64. Speedy's redstone computer. GitHub repository. URL `https://github.com/speedycube64/spredcom`.

Xiaojuan Tang, Zilong Zheng, Jiaqi Li, Fanxu Meng, Song-Chun Zhu, Yitao Liang, and Muhan Zhang. Large language models are in-context semantic reasoners rather than symbolic reasoners. *arXiv preprint arXiv:2305.14825*, 2023.

Karthik Valmeekam, Sarath Sreedharan, Matthew Marquez, Alberto Olmo, and Subbarao Kambhampati. On the planning abilities of large language models (a critical investigation with a proposed benchmark). 2023.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. Recode: Robustness evaluation of code generation models. pp. 13818–13843, 2023.

Jerry Wei, Le Hou, Andrew Lampinen, Xiangning Chen, Da Huang, Yi Tay, Xinyun Chen, Yifeng Lu, Denny Zhou, Tengyu Ma, et al. Symbol tuning improves in-context learning in language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 968–979, 2023a.

Jerry Wei, Jason Wei, Yi Tay, Dustin Tran, Albert Webson, Yifeng Lu, Xinyun Chen, Hanxiao Liu, Da Huang, Denny Zhou, et al. Larger language models do in-context learning differently. *arXiv preprint arXiv:2303.03846*, 2023b.

William G. Wong. 1-Hz, 8-bit RISC micro built using Minecraft Redstone. Electronic Design, December 2021. URL `https://www.electronicdesign.com/technologies/embedded/video/21212550/electronic-design-chungus-2-a-very-powerful-1-hz-minecraft-cpu`.

Zhaofeng Wu, Linlu Qiu, Alexis Ross, Ekin Akyürek, Boyuan Chen, Bailin Wang, Najoung Kim, Jacob Andreas, and Yoon Kim. Reasoning or reciting? exploring the capabilities and limitations of language models through counterfactual tasks. In *Proceedings of the 2024 conference of the north american chapter of the association for computational linguistics: Human language technologies (volume 1: Long papers)*, pp. 1819–1862, 2024.

A Resource Costs of Task-Preserving Representation Changes

We give two examples in which changing the output representation preserves the semantic answer and output length but increases resource requirements. The first isolates encoding computation; the second illustrates the memory required to coordinate input processing with output construction. Both use the semantic task of returning a bit string $u = (u_1, \ldots, u_n) \in \{0,1\}^n$, where $n \geq 2$.

A.1 Output Encoding Can Increase Required Circuit Depth

We consider a family of tasks in which each output is the parity of selected input bits. We first characterize their minimum circuit depth, then bound how much an invertible output encoding can change that depth.

Tasks and output representations. Let $\mathbb{F}_2 = \{0,1\}$ denote binary arithmetic, where addition is XOR and multiplication is AND. A task is specified by a matrix $A \in \mathbb{F}_2^{m \times n}$ and maps an input $u \in \mathbb{F}_2^n$ to the semantic answer

$$y = Au \in \mathbb{F}_2^m.$$

Each row of A specifies one output: a one in column j means that input bit u_j contributes to that output's parity.

The native representation requires outputting y directly. For an invertible matrix $L \in \mathbb{F}_2^{m \times m}$, the alternate representation requires outputting

$$\widetilde{y} = Ly = LAu.$$

The input u remains unchanged. Because $L^{-1}\widetilde{y} = y$, both representations express the same semantic answer using the same number of output bits. The encoding rule L is fully specified.

Computation model. We use deterministic Boolean circuits whose gates each take at most two inputs. All binary Boolean gates, including XOR, are allowed. Inputs and constants have depth zero; a gate has depth one plus the maximum depth of its inputs. Wires may copy a value to any number of destinations without adding depth. The circuit depth is the largest depth of any output.

For a binary matrix M, let $D(M)$ be the minimum depth of a circuit that computes Mx correctly for every input x. Let $w(M)$ be the maximum number of ones in a row of M. Thus, $w(M)$ counts the largest number of input bits contributing to any one output.

Proposition 1 (Depth from input dependence)**.** *For every binary matrix M,*

$$D(M) = \lceil \log_2 \max\{1, w(M)\} \rceil . \tag{4}$$

Proof. Both directions are standard for $\mathbb{F}_2$ linear circuits; we include the argument to fix notation and to make the budget discussion below self-contained.

For the upper bound, consider an output whose row contains k ones. If $k = 0$, the output is the constant zero; if $k = 1$, it copies an input. Both require depth zero. For $k \geq 2$, compute the parity of the selected bits using a balanced binary tree of XOR gates. Padding with zero inputs if necessary gives depth $\lceil \log_2 k \rceil$. All outputs can be computed in parallel, giving the stated upper bound.

For the lower bound, any value at depth d can depend on at most 2^d input bits. At depth zero, a value is an input or a constant. Each subsequent gate combines at most two preceding values, so the number of inputs on which it can depend at most doubles per layer.

An output corresponding to a row with k ones depends on all k selected inputs: flipping any one of them flips the output. Consequently, its depth must satisfy $2^d \geq k$. Taking a row of maximum weight gives the matching lower bound. The all-zero matrix requires only constant outputs and has depth zero. □

Bounding the effect of an output encoding. For any dimensionally compatible binary matrices M and N,

$$D(MN) \leq D(M) + D(N). \tag{5}$$

To see this, first compute Nx and then feed those outputs into a circuit computing $M(Nx)$. The depths of these two circuits add. The minimum depth for MN cannot exceed the depth of this particular construction.

Proposition 2 (Effect of an invertible output encoding)**.** *For every task matrix $A \in \mathbb{F}_2^{m\times n}$ and invertible output encoding $L \in \mathbb{F}_2^{m\times m}$,*

$$-D(L^{-1}) \le D(LA) - D(A) \le D(L). \tag{6}$$

Proof. Both bounds follow directly from the subadditivity of circuit depth under composition, and the argument is standard. We include it for completeness.

Computing the native answer and then encoding it gives

$$D(LA) \le D(A) + D(L),$$

which proves the upper bound. Conversely, since $A = L^{-1}(LA)$, computing the encoded answer and then decoding it gives

$$D(A) \le D(LA) + D(L^{-1}).$$

Rearranging proves the lower bound. □

The bound allows the alternate representation to be either harder or easier. It also permits the depth to remain unchanged. Computing the answer and then encoding it is always an available strategy, but a circuit that computes LAu directly may exploit cancellations or other simplifications.

Recovering the original copying example. Define the encoding

$$G_k(x_1, \dots, x_k) = (x_1 \oplus \cdots \oplus x_k,\ x_2, \dots, x_k), \qquad k \ge 2.$$

Here G_k also denotes the binary matrix implementing this map. Applying it twice returns the original string: the contributions of $x_2, \dots, x_k$ cancel in the first coordinate. Thus, $G_k^{-1} = G_k$, and

$$D(G_k) = D(G_k^{-1}) = \lceil \log_2 k \rceil.$$

For the copying task $A = I_n$ and encoding $L = G_n$,

$$D(A) = 0, \qquad D(LA) = D(L) = \lceil \log_2 n \rceil.$$

The upper bound is therefore attained. Taking instead $A = G_n$ and $L = G_n$ gives $LA = I_n$, attaining the lower bound.

A family with nonzero task and encoding costs. Let $p, q \ge 2$ and divide the input into p disjoint blocks, each containing q bits. Write the input bits as $u_{i,j}$, where i identifies the block and j identifies the position within that block.

The semantic task returns the parity of every block:

$$y_i = \bigoplus_{j=1}^{q} u_{i,j}, \qquad i = 1, \dots, p.$$

Its matrix $A \in \mathbb{F}_2^{p\times pq}$ has exactly q ones in each row, so Proposition 1 gives

$$D(A) = \lceil \log_2 q \rceil.$$

Choose the output encoding $L = G_p$. The required outputs become

$$\widetilde{y}_1 = \bigoplus_{i=1}^{p} y_i = \bigoplus_{i=1}^{p} \bigoplus_{j=1}^{q} u_{i,j}, \qquad \widetilde{y}_i = y_i \quad (i = 2, \dots, p).$$

The first encoded output depends on all pq input bits. The remaining outputs each depend on q bits. Hence,

$$D(L) = \lceil \log_2 p \rceil, \qquad D(LA) = \lceil \log_2 (pq) \rceil.$$

When p and q are powers of two,

$$D(LA) = D(A) + D(L) = \log_2 q + \log_2 p. \tag{7}$$

Thus, both the native task and the encoding require nonzero depth, and no circuit computing the encoded answer can avoid their combined depth in this family. For general p and q, the sum $\lceil \log_2 q \rceil + \lceil \log_2 p \rceil$ exceeds $\lceil \log_2 (pq) \rceil$ by at most one.

For example, take eight blocks of eight bits. Computing each block's parity requires depth three. The alternate first output is the parity of all 64 bits, which requires depth six. A depth-three budget therefore suffices for the native task but cannot compute the encoded task correctly on every input.

Connection to computation budgets. Let β be an allowed circuit depth. If

$$D(A) \leq \beta < D(LA),$$

there exists a circuit within budget that solves the native task on every input, but no circuit within that budget solves the encoded task on every input. Increasing the budget to at least $D(LA)$ makes an exact solution computationally feasible.

This establishes a possible resource obstruction to representation transfer. The result concerns minimum depth for exact deterministic computation; it does not predict accuracy on a particular input distribution. Likewise, interpreting increased LLM reasoning effort as additional computational capacity is a qualitative connection. The existence of a circuit within budget does not guarantee that a particular model will discover or execute the required computation.

A.2 Coordination Requires Additional Working Memory

Problem pair and computation model. The native problem requires outputting $I(u) = u$; the alternate problem requires outputting

$$R(u) = (u_n, \ldots, u_1). \tag{8}$$

Since $R(R(u)) = u$, both outputs express the same answer using n bits.

Input arrives as a one-pass stream. A consumed bit cannot be revisited unless stored. Output is append-only and cannot be read back or used as working storage. Solvers are deterministic and must be correct on every input. Cost is the maximum number of bits of internal state retained during computation, including stored data and indices. For the separate encoding-only problem, the complete answer is supplied as a read-only array whose storage is excluded from working memory.

Let $S_{\mathrm{native}}(n)$ and $S_{\mathrm{reverse}}(n)$ denote the minimum worst-case working memory for the two stream problems, and let $S_{\mathrm{encode}}(n)$ denote it for reversing the supplied array.

Proposition. These costs satisfy

$$\begin{aligned} S_{\mathrm{native}}(n) &= O(1), \\ S_{\mathrm{encode}}(n) &= O(\log n), \\ n - 1 \;\leq\; S_{\mathrm{reverse}}(n) &\leq n + O(\log n). \end{aligned} \tag{9}$$

Proof. The native solver copies each bit as it arrives, using constant memory. For encoding alone, an index suffices to traverse the read-only answer array backward, using $O(\log n)$ bits.

For the alternate stream problem, no output can be emitted before the final input bit arrives: the first output must be u_n, which can still be either zero or one. At this point, the internal state must distinguish all 2^{n-1} possible input prefixes. Otherwise, two distinct prefixes lead to the same state; supplying the same final bit then produces identical outputs, although the required reversals differ. At least $n - 1$ state bits are therefore necessary. Storing all n bits and traversing them backward uses $n + O(\log n)$ bits, establishing the upper bound. □

Thus, $S_{\mathrm{reverse}}(n) = \Theta(n)$ even though copying the stream and reversing an available array each require little working memory. The joint problem must retain the answer that the encoding-only problem receives in stored form. This storage requirement makes coordination the dominant demand under the specified access model.

Scope. These examples establish resource separations for exact deterministic computation. The first relies on bounded gate fan-in; the second relies on one-pass input and unreadable output. They illustrate how encoding and coordination can determine resource requirements without assuming an additive cost decomposition. They do not model LLM inference or predict benchmark accuracy.

B Related-Work Evaluation Comparison

Table 6: Comparison of focal representation-robustness evaluations. ✓ denotes explicit coverage; × denotes absence. *Native-success conditioning* requires a reported metric whose denominator is the model's native-correct instances. *Real* and *Synthetic* denote existing and experimentally assigned target conventions.

Study	Domain	Native-success conditioning	Target coverage	
			Real	Synthetic
Embers: multiplication formats (McCoy et al., 2023)	Arithmetic	×	✓	×
Reasoning or Reciting? (Wu et al., 2024)	Arithmetic; code; spatial; games	×	✓	✓
ReCode (Wang et al., 2023)	Code generation	×	✓	✓
Planning Abilities (Valmeekam et al., 2023)	Blocksworld; Logistics	×	✓	✓
Talk like a Graph (Fatemi et al., 2024)	Graph reasoning	×	✓	✓
MultiPL-E (Cassano et al., 2022)	Programming	×	✓	×
FormatSpread (Sclar et al., 2024)	NLP classification; MCQ	×	✓	×
Identifier Swaps (Miceli-Barone et al., 2023)	Python code	×	×	✓
ParaRel (Elazar et al., 2021)	Factual queries	×	✓	×
Larger Models: SUL-ICL (Wei et al., 2023b)	NLP classification	×	×	✓
CheckList: invariance (Ribeiro et al., 2020)	Sentiment; QQP; QA	×	✓	×
GSM-Plus (Li et al., 2024)	Math word problems	×	✓	×
Semantic vs. Symbolic Reasoners (Tang et al., 2023)	Logical reasoning	×	×	✓
MatheMagic (O'Brien et al., 2025)	Arithmetic	×	✓	✓
Symbol Tuning (Wei et al., 2023a)	NLP classification	×	×	✓
PRR (ours)	**Chess; Boolean logic**	✓	✓	✓

C Evaluation Prompts

The following templates reproduce the prompts used in the experiments. Angle-bracketed fields are replaced with the corresponding task-instance values. Step 1 withholds the alternate-representation rules, whereas Step 3 supplies them in full. Chess experiments use the JSON output mode shown below. Not limiting the reasoning wordings would end up resulting in very long reasoning traces, and the model would end up answering blank or not answering.

C.1 Chess Prompts

Chess Step 1: Syn-Chess, rules withheld

```
You are a high-level chess engine. Find the completely forced checkmate sequence.

IMPORTANT: You must express your answer using Syn-Chess notation, NOT standard
    UCI or SAN.

THE PUZZLE
  Side to move : <SIDE>
  Mate in      : <N>
  FEN          : <FEN>

OUTPUT FORMAT
Output ONLY the attacker's moves as a comma-separated Syn-Chess list.
```

```
Do NOT include the opponent's forced replies.

Respond in valid JSON with exactly two keys:
{
  "moves": "<comma-separated move list here>",
  "reasoning": "<at most two short sentences>"
}

Example:
{
  "moves": "Ne2 B 2, Bc6 BBL 1",
  "reasoning": "The first move forces the king back. The second move delivers
      mate."
}

Put `moves` first. Keep `reasoning` under 40 words.
Output ONLY the complete JSON object. No markdown fences, no extra text.
```

Chess Step 2: Native representation

```
You are a high-level chess engine. Find the completely forced checkmate sequence.

THE PUZZLE
  Side to move : <SIDE>
  Mate in      : <N>
  FEN          : <FEN>

OUTPUT FORMAT
Output ONLY the attacker's moves. Do NOT include the opponent's forced replies.

Respond in valid JSON with exactly two keys:
{
  "moves": "<comma-separated UCI move list, attacker moves only>",
  "reasoning": "<at most two short sentences>"
}

Example:
{
  "moves": "e1e8, d5e6",
  "reasoning": "The rook check forces the king's reply. The queen then delivers
      mate."
}

Put `moves` first. Keep `reasoning` under 40 words.
Output ONLY the complete JSON object. No markdown fences, no extra text.
```

Chess Step 3: Syn-Chess with rules

```
In this variant called "syn-chess", standard chess rules remain unchanged, but
     the way moves are expressed is transformed using two deterministic systems: a
      piece-symbol cipher and a directional inversion.

First, every moving piece is renamed using a fixed shift mapping:
- Pawns (P)     Knights (N)
- Knights (N)     Bishops (B)
- Bishops (B)     Rooks (R)
- Rooks (R)     Queens (Q)
- Queens (Q)     Kings (K)
- Kings (K)     Pawns (P)

Second, movement directions are defined relative to the moving side (the attacker
     ), then inverted:
- Forward (F) = toward the opponents back rank
- Backward (B) = opposite of forward
- Right (R) = attackers right
- Left (L) = attackers left

Each direction is then flipped 180  :
- F     B
```

```
- B     F
- R     L
- L     R
- FL     BR, FR     BL, BL     FR, BR     FL

Knight movement is expressed as a combination of two directions (2 squares + 1
    square), with the longer component written first, and both components
    inverted.

Moves must be written in relative movement notation (RMN) as:
[Piece][StartSquare] [Direction] [Steps]

Where the piece symbol is AFTER applying the cipher, the direction is already
    inverted, and steps indicate distance (knights always use 1).

Only output the attacking s i d e s moves, separated by commas. Do not include
    opponent moves and do not use standard chess notation.

Base, untransformed RMN examples:
- White pawn e2e4     Pe2 F 2
- White knight g1f3     Ng1 FFL 1
- Black queen d8h4     Qd8 FL 4



The Puzzle to solve:
  Side to move : <SIDE>
  Mate in      : <N>
  FEN          : <FEN>

Find the forced checkmate sequence and express every attacker move in RMN.
Output ONLY the attacker's moves as a comma-separated RMN list.
Do NOT include the opponent's forced replies.

Respond in valid JSON with exactly two keys:
{
  "moves": "<comma-separated move list here>",
  "reasoning": "<at most two short sentences>"
}

Example:
{
  "moves": "Ne2 B 2, Bc6 BBL 1",
  "reasoning": "The first move forces the king back. The second move delivers
      mate."
}

Put `moves` first. Keep `reasoning` under 40 words.
Output ONLY the complete JSON object. No markdown fences, no extra text.
```

C.2 Boolean-Logic Prompts

Logic Step 1: Redstone, rules withheld

```
You have the below components available:
-> redstone dust (R)
-> lever (L)
-> block of quartz (Q)
-> redstone torch (T can be TU, TD, TL, TR or just T. The direction of the letter
      is the direction of the tip of the torch and the flow of the power)
-> Redstone Lamp (G)
-> Skip space / Air (0)

The way you can make use of these components is by placing them inside of square
    matrices as follows:

Placing a lever followed by redstone on a 9x9 matrix.
[
```

```
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,],
[0,0,0,0,'R',0,0,0,0,],
[0,0,0,0,'R',0,0,0,0,],
[0,0,0,0,'L',0,0,0,0,],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0]]

Placing a torch pointing away from a block of quartz on a 9x9 Matrix. Thus we use
     U meaning up. Keep in mind the directional torches require a block to be
    placed beforehand.
[
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0,],
[0,0,0,0,'TU',0,0,0,0,],
[0,0,0,0,'Q',0,0,0,0,],
[0,0,0,0,0,0,0,0,0,],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0]]

YOUR task is to design a <GATE_DESCRIPTION> on a 9x9 matrix.

IMPORTANT: THE MATRIX MUST ALWAYS BE INSIDE OF TRIPLE BACKTICKS ```
```

Logic Step 2: Native Python

```
You are a Python programmer with expertise in digital logic. Your task is to
    write a Python function that implements <GATE_DESCRIPTION> which has inputs
    as <INPUTS> and expects outputs as <OUTPUTS>.

Please provide a complete Python function that:
1. Takes the appropriate number of boolean inputs
2. Implements the logic of the specified gate
3. Returns the boolean output

Use only basic Python operations and avoid using any external libraries.
the output of your code must be a string which shows the truth table in any form
    of representation.


IMPORTANT: Your response should only contain the Python code, without any
    additional explanation, IT WILL DIRECTLY BE USED TO RUN.
```

Logic Step 3: Redstone with rules

```
You have the below components available, along with the letter they will be
    represented as, a general idea of their working and their definitions from
    the official wiki:

1. redstone dust (R) [You can think of this as a wire to direct power, one
    property is if two wires merge into a third wire, the third wire directs
    power when either of those wires direct power]
-> Wiki Definition: Redstone dust (often just called redstone) is a mineral used
    in crafting, brewing, and trimming, as well as being used as a placeable wire
    . Redstone wire is a non-solid block that can transmit a redstone power
    signal. Redstone wire can power blocks it is placed on or pointing at.

2. lever (L) [Can produce continuous power, when switched on]
-> Wiki Definition: The lever is a non-solid block that can be switched to
    provide redstone power.

3. Block of quartz (Q) [A type of block in minecraft]
-> Wiki Definition: A block of quartz, internally known as a quartz block, is a
    mineral block used for decoration.
```

```
4. Redstone Torch (T, can be TU, TD, TL, TR or just T) [The direction of the
     letter is the direction of the tip of the torch and the flow of the power,
     can also be used to invert power when combined with a Block]
-> Wiki Definition: A redstone torch is a non-solid block that produces a full-
     strength redstone signal on all sides adjacent to it, except for its attached
      block, and can power the block directly above it. It deactivates while the
     block it is attached to is powered.

5. Redstone Lamp (G) [consumes power to glow]
-> Wiki Definition: A redstone lamp is a solid block that produces light while it
      is receiving redstone power.

6. Skip space / Air (0) [used to indicate no blocks being placed in that space]


Placing a lever followed by redstone dust on a 9x9 matrix, indicating that the
     lever is powering the redstone dust.
[
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,'G',0,0,0,0],
[0,0,0,0,'R',0,0,0,0],
[0,0,0,0,'R',0,0,0,0],
[0,0,0,0,'L',0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0]]

In this case the Redstone Lamp glows. However, the direction of the current is
     also important, if the Redstone lamp is placed adjacent to the line of
     redstone dust, instead of the redstone line of dust flowing into it, it will
     not glow.

[
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,'R',0,0,0,0],
[0,0,0,0,'R',0,0,0,0,],
[0,0,0,'G','R',0,0,0,0],
[0,0,0,0,'R',0,0,0,0],
[0,0,0,0,'L',0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0]]

In this case the Redstone lamp will not glow.

Placing a torch pointing away from a block of quartz means its adjacent to that
     block on a 9x9 Matrix, indicating if there exists current flowing into the
     block of quartz, it would get inverted while leaving the torch.
[
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,'R',0,0,0,0],
[0,0,0,0,'TU',0,0,0,0],
[0,0,0,0,'Q',0,0,0,0],
[0,0,0,0,'R',0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0]]

One cannot directly power a block of Quartz using a lever, it needs to be given
     power through the redstone dust.

One thing to note is in the case that redstone overlaps, if needed can be
     designed in the below way:
[
[0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 'R', 0, 0, 0, 0],
```

```
[0, 0, 0, 0, 'R', 0, 0, 0, 0],
[0, 0, 'R0', 'R1', 'R2', 'R1', 'R0', 'R', 0],
[0, 0, 0, 0, 'R', 0, 0, 0, 0],
[0, 0, 0, 0, 'R', 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0]
]

where R0 shows the start of the overlap, R1 shows redstone being placed one level
     higher and R2 shows redstone being placed 2 levels higher. This gradual
    increase and decrease in y-levels must be followed whenever redstone overlap
    is needed to be used. this notation is to be used only if needed.

YOUR task is to design a <GATE_DESCRIPTION> on a square n x n matrix, where 2 <=
    n <= 30, and each component should be in quotes.

IMPORTANT: THE MATRIX MUST ALWAYS BE INSIDE OF TRIPLE BACKTICKS ```
```

D Asset Availability and Licensing

The benchmark construction procedure is described in Section 4.1, and the complete evaluation prompts are provided in Appendix C. The code and generated evaluation assets are not publicly available during review. We commit to releasing the code and generated evaluation assets publicly after the review decision under the GNU Affero General Public License v3.0 (AGPL-3.0).

The chess tasks are derived from the Lichess open puzzle database (`https://database.lichess.org/#puzzles`), whose database exports are released under the Creative Commons CC0 1.0 Universal dedication. The Qwen3.6-27B and Qwen3.8-27B checkpoints are distributed under the Apache License 2.0 and obtained through the official Qwen repositories (`https://huggingface.co/Qwen/Qwen3.6-27B` and `https://huggingface.co/Qwen/Qwen3.8-27B`). The Gemma-4-31B checkpoint is also distributed under the Apache License 2.0 and obtained through its official provider repository. GPT-5.6-Sol and GPT-5.6-Luna are accessed through the official OpenAI API under the provider's applicable terms.

E Human Evaluation Details

We conduct a small evaluation study to test whether chess players can successfully apply the Syn-Chess rules. We ask two volunteers with chess ratings of at least 1000 to participate in the study. Each participant receives a tutorial specifying the piece cipher, directional transformation, and output format, and then solves the first 50 low-rated puzzles in ascending difficulty order from the benchmark.

We score first-move and full-puzzle accuracy against the deterministic RMN ground truth. Participant 1 achieves 100% $(50/50)$ first-move accuracy and 94% $(47/50)$ full-puzzle accuracy. Participant 2 achieves 98% $(49/50)$ first-move accuracy and 92% $(46/50)$ full-puzzle accuracy. These results show that participants can apply the supplied Syn-Chess rules while solving the underlying chess task.

F QUALITATIVE FAILURE EXAMPLES

We visualize six chess and four Minecraft Redstone instances for which a model solves the native task but fails in the alternate representation despite receiving its rules. Each example is drawn from a pair of saved result JSONs; the boards, move strings, Python excerpts, and circuit matrices reflect those records. Reference encodings and highlighted defects are obtained by deterministic checks. These curated examples illustrate observable failure modes, not their prevalence or the models' internal causes of failure.

F.1 CHESS: FROM UCI TO SYN-CHESS

Figures 4–6 show six GPT-5.6-Sol Mate-in-2 puzzles. Every native attacker line replays legally to checkmate when interleaved with the defender replies saved from a compatible solution-DAG path; the defenders replies are not model output. The first alternate move fails the recorded evaluation in every case. Boards show the initial position and first-move paths, with White at the bottom even when Black moves. Later alternate moves are quoted as not reached, without inventing a continuation.

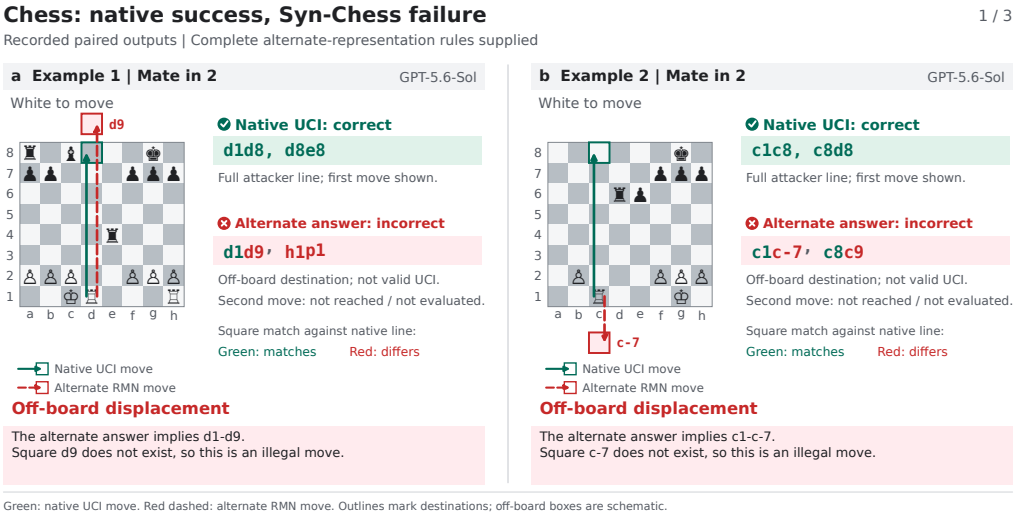

Figure 4: **Off-board Syn-Chess displacements.** GPT-5.6-Sol solves both native puzzles but produces no legal decoded first move in Syn-Chess. (a) On `09T65`, `Qd1 B 8` overshoots the board; the native move encodes as `Qd1 B 7`. (b) On `00JET`, `Qc1 F 8` differs in both direction and distance from `Qc1 B 7`, again leaving the board.

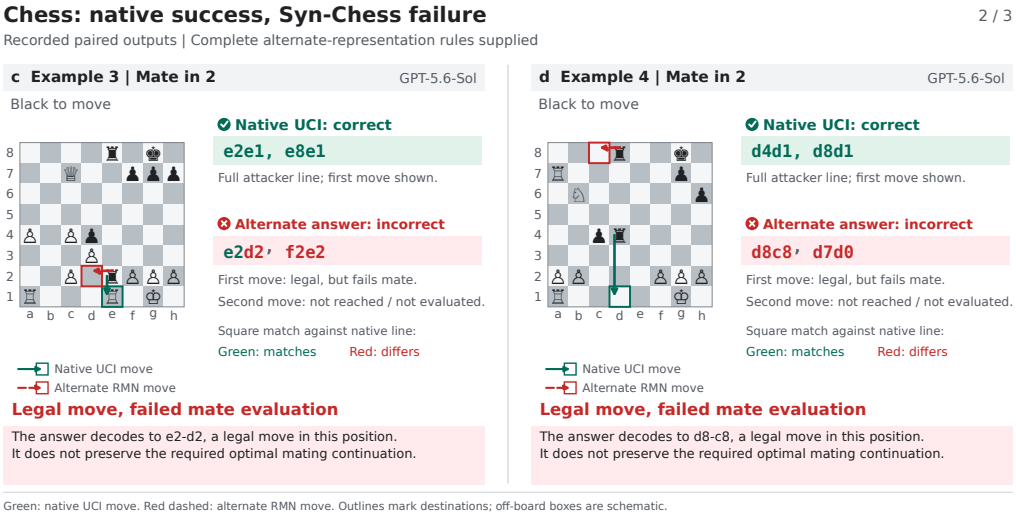

Figure 5: **Legal decoded moves that fail the mate evaluation.** GPT-5.6-Sol's alternate outputs decode to legal chess moves, but their recorded first-move verdicts are incorrect. (c) On `0BtyY`, `Qe2 L 1` decodes to `e2d2`. (d) On `1HEWG`, `Qd8 L 1` decodes to `d8c8`. The green references encode the accepted native first moves.

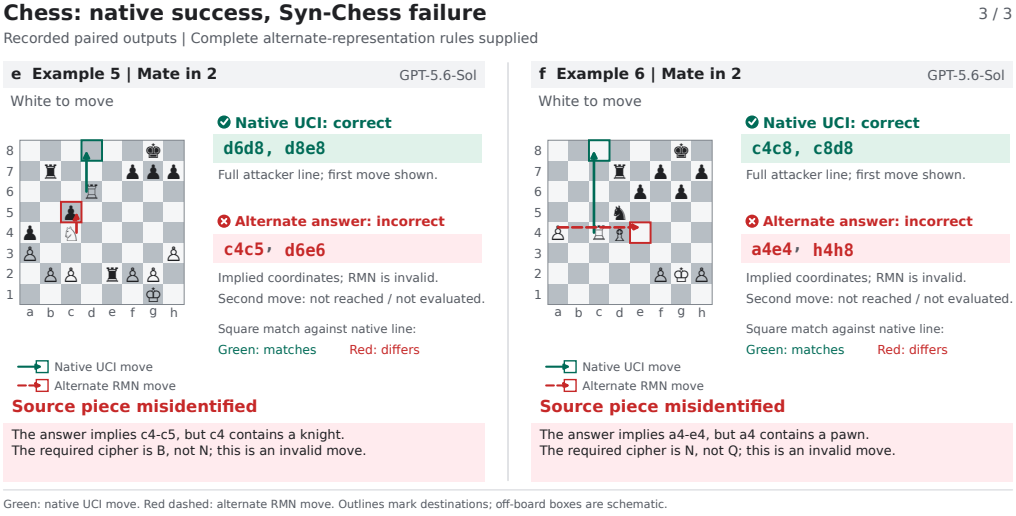

Figure 6: **Piece symbols inconsistent with the source position.** GPT-5.6-Sol's generated symbols do not match the cipher of the pieces on the named source squares. (e) On `0uzta`, `Nc4 B 1` names a knight on `c4`, whose cipher is `B`, not `N`. (f) On `0xZPn`, `Qa4 L 4` names a pawn on `a4`, whose cipher is `N`, not `Q`. Neither output yields a legal move under the repository decoder.

F.2 Boolean Logic: From Python to Minecraft Redstone

Figures 7 and 8 show three GPT-5.6-Sol outputs and one GPT-5.6-Luna output. Each native Python function's truth table was rechecked against the saved task specification. All four alternate outputs fail the repository's structural validator. The visualizations preserve the saved matrices, including missing cells, and are layout diagrams rather than Minecraft screenshots or power simulations. Component shading does not indicate a simulated power state; coordinates are zero-based. Defects may coexist within an output, and no Minecraft execution was rerun for these figures.

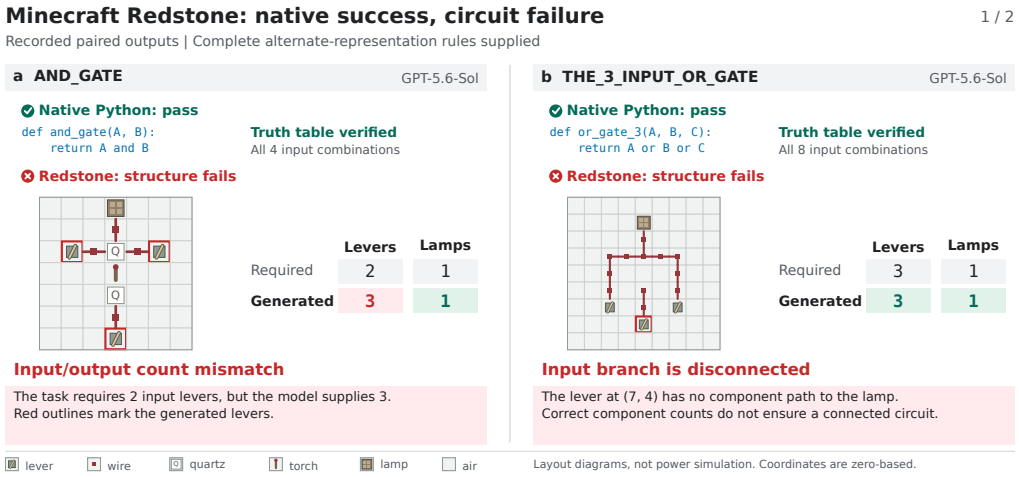

Figure 7: **Input-count and connectivity failures in Redstone.** Both examples are from GPT-5.6-Sol. (a) For `AND_GATE`, the model supplies three levers for a two-input task. (b) For `THE_3_INPUT_OR_GATE`, the lever and lamp counts match the specification, but the lever at row 7, column 4 has no component path to the lamp. The inset shows the disconnected branch without repairing the layout.

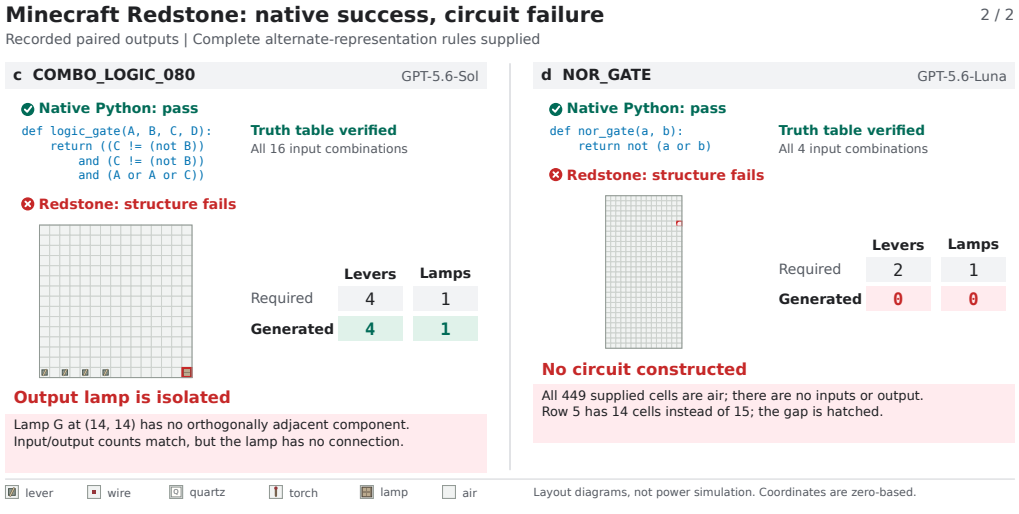

Figure 8: **An isolated output and an empty circuit.** (c) GPT-5.6-Sol supplies the required four levers and one lamp for `COMBO_LOGIC_080`, but the lamp at row 14, column 14 has no orthogonally adjacent component. This output also has isolated levers; the figure highlights the independently verified lamp defect. (d) GPT-5.6-Luna's `NOR_GATE` output contains 449 air cells and no levers or lamp. Row 5 has only 14 cells while the other rows have 15; hatching marks the absent cell rather than padding it with air.