THE TOKEN GAMES: EVALUATING LANGUAGE MODEL REASONING WITH PUZZLE DUELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Evaluating the reasoning capabilities of Large Language Models is increasingly challenging as models improve. Human curation of hard questions is highly expensive, especially in recent benchmarks using PhD-level domain knowledge to challenge the most capable models. Even then, there is always a concern about whether these questions test genuine reasoning or if similar problems have been seen during training. Here, we take inspiration from 16th-century mathematical duels to design *The Token Games* (TTG): an evaluation framework where models challenge each other by creating their own puzzles. We leverage the format of Programming Puzzles — given a function that returns a boolean, find inputs that make it return True — to flexibly represent problems and enable verifying solutions. Using results from pairwise duels, we then compute Elo ratings, allowing us to compare models relative to each other. We evaluate 18 models — frontier systems, low-reasoning-effort variants, and open-weights models — on TTG, and closely match the rankings from existing benchmarks such as Humanity's Last Exam, without involving any human effort in creating puzzles and at a total cost of about $2,400 — a small fraction of the cost of sourcing expert-authored questions for comparable benchmarks. We also find that creating good puzzles is still a highly challenging task for current models. Overall, our work suggests new paradigms for evaluating reasoning that avoid saturation by design, and that allow testing models for other skills like creativity and task creation alongside problem solving.

1 INTRODUCTION

In the 16th century, mathematicians of the Italian renaissance rose to fame not by publishing breakthrough results but by winning public mathematical duels, in which two competitors posed a series of problems to each other: to win, one should solve the opponent's problems and stump them with one's own (Azzolini, 2019). In a famous duel against Antonio Maria Fior, Niccolò Tartaglia's victory revealed that he had found a general solution to cubic polynomials before any method was publicly known (Toscano, 2020). As an evaluation of mathematical prowess, such duels had remarkable properties. Since solutions were verifiable, the public could confirm a victory without understanding the underlying reasoning. Since arbitrarily difficult problems were acceptable, as long as they were well-defined and the proposer could solve them, duels could not be saturated: there was always room for a new mathematician to showcase their brilliance. Finally, duels implicitly tested creativity: posing hard but already known problems would not work against opponents that had memorized the literature.

In this paper, we take inspiration from mathematical duels to design a reasoning evaluation for Large Language Models that overcomes several limitations of current benchmarks. As LLMs improve in their capability for reasoning, it has become steadily harder for humans to design new questions that can measure progress as new models are released (Phan et al., 2025a; Rein et al., 2023). We thus propose *The Token Games* (TTG), a reasoning evaluation based on duel-style competitions between pairs of LLMs where models themselves propose challenge puzzles for each other. TTG inherits the properties outlined above: it cannot be saturated (stronger models can in principle always surpass previous ones), it evaluates models on arbitrarily difficult problems without expensive human curation, and it tests creativity: repeating hard problems seen during training is sub-optimal, as other models may have been trained on them too.

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

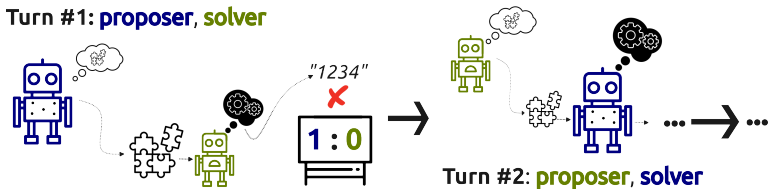

Figure 1: Reasoning duel in The Token Games. Two language models take turns between being puzzle proposers or solvers. Proposers score in a turn if they design a puzzle, give a correct solution, and have the opponent fail to solve the challenge. Puzzles are represented as Python functions returning a boolean value, with the challenge consisting of finding inputs to make it return true.

In a TTG duel, as illustrated in Figure 1, two language models take turns alternating between the roles of "proposer" and "solver". At the beginning of a turn, the proposer designs a puzzle and gives its own solution. To ensure that we can verify solutions, we adopt the format of Programming Puzzles (Schuster et al., 2021): a puzzle is the source code of a verifier function that returns a boolean value, and a solution is an input for which it returns true. Despite its name, this representation is flexible enough to encode arbitrarily hard logical and mathematical problems without revealing their solutions (see Section 3). If the proposer's own solution is incorrect, the solver scores. If it is correct but the solver also succeeds, the turn ends in a draw; if the solver fails, the proposer scores. Whoever scores more points across a fixed number of turns wins the match.

We use the outcomes of a set of duels to compute Elo ratings for each language model, allowing us to rank models while always leaving space for new, more capable models to outperform. We empirically evaluate whether the TTG ranking agrees with other public benchmarks, such as GPQA-Diamond, HLE, and ARC-AGI. We find significant positive correlations between our Elo ratings and HLE ($\rho = 0.69$), ARC-AGI ($\rho = 0.84$), GPQA Diamond ($\rho = 0.77$), SWE-Bench Pro ($\rho = 0.72$), and TextQuests (a text-adventure benchmark, $\rho = 0.77$), all of which survive multiple-testing correction and a leave-one-out sensitivity analysis. Moreover, we also separately evaluate models' abilities in terms of the fraction of turns they won as a solver and as a proposer. We find that win rates as a proposer correlate most strongly with general reasoning benchmarks (up to $\rho = 0.90$ with ARC-AGI), and that win rates as a solver correlate most strongly with SWE-Bench Pro ($\rho = 0.80$). This indicates that, at a total cost of roughly $2,400 for a full 18-model tournament (see Appendix B) — versus, e.g., the $500k prize pool used to source the questions of HLE — TTG is a strong proxy both for general reasoning (like HLE, ARC-AGI, GPQA) and code understanding (like SWE-Bench Pro). We show several analyses of the capabilities and failure modes that TTG reveals, including many cases of overconfidence of frontier models when designing hard puzzles (and failing to solve their own problems), as well as unique preferences for topics exhibited by different models across the 3,607 distinct puzzle topics arising in our tournament. Overall, our work contributes new ideas for designing scalable and inexpensive evaluations of LLM reasoning to stand the challenges posed by rapidly improving models.

2 Related Work

Language Model Reasoning. Early experiments with GPT-3 and PaLM (Wei et al., 2022) showed that prompting models to provide a *chain-of-thought*, or fine-tuning them on scratchpads (Nye et al., 2021), substantially improves performance on multi-step reasoning problems. Recent models are increasingly trained with reinforcement learning targeted at reasoning in language (Zelikman et al., 2022), yielding a line of "reasoning models" starting with OpenAI o1 (Jaech et al., 2024) and DeepSeek r1 (Guo et al., 2025). The resulting rapid, sometimes discontinuous jumps in benchmark performance have made measuring the capabilities of the most recent models a challenge of its own.

LLM Reasoning Benchmarks. Early benchmarks of mathematical problems used to evaluate LLMs included GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021), extending to broader tasks such as BIG-Bench (BIG-bench authors, 2023) and ARC-Challenge (Clark et al., 2018). With rapid improvement, and sometimes near saturation, of performance in these benchmarks, recently benchmark developers turned to curating problems from experts, with benchmarks like

GPQA (Rein et al., 2023), Humanity's Last Exam (HLE, Phan et al. (2025a)), ARC-AGI (Chollet et al., 2026) and Frontier Math (Glazer et al., 2024) recruiting participant experts explicitly to design questions requiring highly specialized (e.g., PhD-level) knowledge in various fields. Relying on domain experts to design questions that are difficult for frontier models comes at a substantial cost: constructing HLE, for instance, involved sourcing experts with a prize pool of 500k USD for the top high-quality submissions. Even then, these benchmarks still risk being saturated by rapidly improving models: even one year after its release, for instance, the most performant models already achieved close to 90% accuracy on GPQA. Furthermore, without having access to model's training data, it is impossible to disentangle factors such as potential data contamination (e.g., GPQA is available online) or similarity to previously existing problems (even if human authors were unaware of them). TextQuests (Phan et al., 2025b) and SWE-Bench Pro (Jimenez et al., 2024), which we also compare our rankings to, are both based on real-world data: old works of interactive fiction and source-code repositories from code hosting platforms, however, their difficulty is unlikely to scale well with newer models.

Problem Generation with LLMs. Several recent works use language models themselves to generate problems, but due to the concern that LLM-generated problems might be unreliable (e.g., have incorrect answers, missing hypotheses, or be ambiguous), these problems generally serve as training data rather than as a target for evaluation. Notable examples include MetaMath (Yu et al., 2024), which uses LLMs to generate synthetic mathematical problems for training in language, AbsoluteZero (Zhao et al., 2025) which self-trained a Qwen model using model-generated programming puzzles, and models that generate and prove formal mathematical conjectures (Poesia et al., 2024; Dong & Ma, 2025). Our work uses LLMs to propose *Programming Puzzles* (Schuster et al., 2021) for one another, in contrast with previous work where puzzles were extracted from human sources. Moreover, our main goal here is evaluation, whereas in all the aforementioned self-play setups models propose and train on their own puzzles.

LLM arenas and game-based evaluations Benchmarks such as Chatbot Arena (Chiang et al., 2024) and Copilot Arena (Chi et al., 2025) also compare pairs of LLMs in competition, ranking their responses to user prompts. In existing "arenas", however, the judges are human participants, and the prompts are also created by users, unlike in our work where both generation and verification are automated. Other LLM evaluations place models in sequential decision-making setups, including in playing games such as in PuzzlePlex (Long et al., 2025) and SmartPlay (Wu et al., 2023); here, however, puzzles are designed by benchmark authors, testing fixed capabilities, and might not scale in difficulty with increasingly capable models.

3 The Token Games

We now describe *The Token Games*, our proposed evaluation framework for LLM reasoning. TTG is based on the concept of a *programming puzzle* introduced in Schuster et al. (2021) as a unified representation for reasoning problems. A programming puzzle (or "puzzle", for short) is specified by a function $f(x)$ that receives arbitrary input x and returns a boolean value (true or false). Then, y is a solution to the puzzle if $f(y) = \texttt{true}$.

Specifying a verifier f is a natural way to encode a wide variety of arbitrarily difficult reasoning challenges, including NP-complete problems such as boolean satisfiability and a range of mathematical problems; despite their name, programming puzzles can be extremely difficult while only requiring a very basic ability to read code (we quantify this in Section 4.2). Even several open problems in mathematics, such as finding a cycle in the Collatz $3n + 1$ sequence, can be represented as a puzzle that simply *verifies* the given solution, without f revealing anything about how to solve the problem. Instead of collecting human-written puzzles to evaluate LLMs, our key idea is to design TTG around "reasoning duels" where two LLMs propose puzzles to challenge each other.

3.1 Reasoning duel

In TTG, pairs of LLMs in a pool compete in a series of *duels*, which are conducted programatically as described in Algorithm 1. In a duel, the two models play a fixed number of rounds where they alternate between the roles of *proposer* and *solver*. Each round can end with one of the two models

Table 1: Model performance comparison on The Token Games and, in column order, Humanity's Last Exam, ARC-AGI, SWE-Bench Pro, TextQuests, and GPQA Diamond. For TTG, we show match Elo with 95% bootstrap confidence intervals as super/subscripts, the clean solver success rate on valid puzzles (Solv%), the duel-scoring solver win rate (SolvW%), and the proposer win rate (Prop%), as defined in Section 3.2. For the other benchmarks, we report public-leaderboard accuracies (sources in Appendix C). Each metric is followed by the model's rank; the top and bottom ranks are highlighted. The top-2 models are preserved (in order) in 97% of bootstrap resamples, the same top-3 set in 61%, and the same bottom-3 set in 73%; bootstrap details in Appendix A.

	TTG								HLE		ARC-AGI		SWE-BP		TQ		GPQA-D	
Model	**Elo**	**R**	**Solv%**	**R**	**SolvW%**	**R**	**Prop%**	**R**	**Acc**	**R**	**Acc**	**R**	**Acc**	**R**	**Acc**	**R**	**Acc**	**R**
gpt-5.4	1746^{+213}_{-118}	1	98.9	1	99.3	1	60.4	1	40.3	5	65.0	4	49.7	6	42.2	2	92.0	4
gpt-5.5	1445^{+120}_{-85}	2	97.6	2	98.3	2	38.2	2	43.6	2	77.5	1	53.4	4	42.0	3	93.5	2
claude-opus-4-8	1337^{+102}_{-85}	3	88.2	7	92.3	6	28.7	4	42.2	3	67.5	3	65.4	1	40.1	4	92.0	4
gemini-3.1-pro-pre...	1311^{+103}_{-87}	4	85.7	8	90.1	8	33.8	3	45.9	1	73.3	2	46.7	7	45.8	1	94.1	1
claude-opus-4-7	1280^{+102}_{-89}	5	92.0	3	94.4	3	16.9	6	39.0	6	50.8	6	60.9	2	37.0	5	91.4	6
claude-sonnet-4-6	1238^{+97}_{-84}	6	88.5	6	91.6	7	13.3	8	21.1	11	24.2	8	53.8	3	31.5	8	87.5	11
qwen/qwen3.7-max	1203^{+86}_{-81}	7	91.5	4	93.8	4	12.3	9	–	–	–	–	–	–	–	–	92.3	3
gpt-5.4-mini	1096^{+101}_{-97}	8	84.5	10	89.9	9	18.6	5	23.5	9	5.8	9	37.9	10	29.6	9	87.5	11
claude-opus-4-7:low	1045^{+79}_{-86}	9	85.5	9	89.6	10	12.3	9	–	–	–	–	–	–	–	–	–	–
grok-4.20-0309-rea...	1045^{+90}_{-89}	9	79.8	12	85.9	12	13.4	7	30.2	8	55.0	5	26.3	12	18.5	12	88.5	10
claude-haiku-4-5	1000^{+0}_{-0}	11	69.5	15	77.0	15	1.8	16	9.7	13	4.0	11	41.0	8	15.1	13	67.2	15
z-ai/glm-5.2	820^{+82}_{-113}	12	80.8	11	86.0	11	8.1	13	41.4	4	–	–	51.8	5	33.3	7	89.5	9
grok-4-fast-reason...	746^{+86}_{-132}	13	62.0	16	71.0	16	6.0	15	17.8	12	3.3	12	12.0	13	20.1	11	84.7	13
gpt-5.5:low	739^{+79}_{-118}	14	89.3	5	92.6	5	9.5	12	–	–	–	–	–	–	–	–	91.0	7
gemini-3-flash-pre...	725^{+97}_{-128}	15	72.9	13	80.5	13	11.6	11	36.6	7	30.8	7	38.6	9	36.4	6	89.8	8
deepseek-v3.2-thin...	664^{+91}_{-147}	16	71.9	14	79.7	14	6.9	14	21.8	10	5.0	10	33.1	11	21.2	10	84.0	14
qwen/qwen3-coder	210^{+31}_{-240}	17	53.4	17	63.0	17	0.6	17	–	–	–	–	–	–	–	–	61.8	16
gpt-4o	135^{+3}_{-251}	18	20.8	18	32.2	18	0.1	18	2.7	14	0.0	13	4.9	14	13.1	14	54.3	17

scoring a point, or in a draw. Round i starts with the proposer first designing a puzzle, f_i, and providing a solution, s_i, to its own puzzle. The proposer is given a private scratchpad for reasoning about their puzzle design before committing to a final puzzle and its answer; this allows models to engage in chain-of-thought reasoning when constructing puzzles. The puzzle is accepted only if the proposer's solution is correct: that is, $f_i(s_i) = \texttt{true}$. If this is not the case, the proposer immediately loses this round, and models change sides. Otherwise, if the proposer's solution is correct, the solver receives f_i (but not the proposer's private reasoning) and is challenged to find some solution, x_i (which need not be equal to s_i). We then test whether $f_i(x_i) = \texttt{true}$: if not, the proposer wins this round, and otherwise the round ends in a draw. At the end of all rounds, the model with most points is the winner. The match draws if both models end with the same number of points.

Solutions are verified by executing the puzzle code in a sandboxed Python environment. If evaluation of $f_i(s_i)$ or $f_i(x_i)$ results in a runtime error (e.g., due to malformed code, type errors, or timeout), the solution is treated as incorrect. A puzzle input can be any Python value that can be constructed in-place, including strings, numbers and many data structures.

As the duel progresses, both models receive a history of all previous rounds. This allows models to adapt their strategies dynamically, such as adjusting to their opponent's difficulty level. Unlike for their own puzzles, each model only has access to partial information about their opponent's rounds: they see the puzzle code f_i, their own solution, and the round's outcome, but not the opponent's private reasoning or explanation of their puzzle's intended solution.

Algorithm 1 Reasoning Duel

Require: Models M_1, M_2; number of rounds R
$S_1 \leftarrow 0, S_2 \leftarrow 0, H \leftarrow \emptyset$
for $i \leftarrow 1$ to $2R$ **do**
  $m_{\text{prop}} \leftarrow M_{1+(i \bmod 2)}$, $m_{\text{solv}} \leftarrow M_{1+((i+1) \bmod 2)}$
  $(f_i, s_i) \leftarrow \textsc{ProposePuzzle}(m_{\text{prop}}, H)$
  if $f_i(s_i) \neq \texttt{true}$: $\textsc{ScorePoint}(m_{\text{solv}})$; **continue**
  $x_i \leftarrow \textsc{Solve}(m_{\text{solv}}, f_i)$
  if $f_i(x_i) = \texttt{false}$: $\textsc{ScorePoint}(m_{\text{prop}})$
  $H \leftarrow H \cup \{(f_i, \text{OUTCOME}_i)\}$
end for
return M_1 wins **if** $S_1 > S_2$; M_2 wins **if** $S_2 > S_1$; **else** draw

Reasoning duels have interesting properties from the perspective of LLM evaluations. Duels in principle incentivize models to creatively come up with *novel questions*: an opponent might have already seen the solution of any puzzle taken straight from training data (indeed, we observe solvers defeating puzzles precisely by recognizing memorized artifacts; see Appendix D.7). Second, the fact that models design the puzzles themselves allows them to show capabilities not explicitly asked for during benchmark design. Finally, a duel requires a model to have a good self-evaluation of its own capabilities: it should not propose puzzles that it itself cannot solve. Our analysis in Section 4 shows that even the most capable reasoning models are sometimes overconfident in this way.

3.2 Rating models

To obtain a single scalar measure of reasoning ability across multiple models, we run duels between model pairs and fit ratings by applying the Bradley-Terry (BT) model on the set of outcomes, as in prior work on pairwise preference evaluation of LLMs (Chi et al., 2025; Chiang et al., 2024). We parameterize the BT equation using the same exponential scoring function used in the Elo rating system, popular in chess and other competitive sports. Under the standard Elo model, the probability that model A defeats B in a match is $P(A \succ B) = 1/\big(1 + 10^{(E_B - E_A)/\sigma}\big)$, where E_A and E_B denote the Elo ratings of models A and B, respectively, and σ is a scale parameter (we use $\sigma = 400$, the standard choice in chess). We fit Elo ratings simultaneously on the final outcomes from all pairwise duels. Formally, let w_{AB}, ℓ_{AB}, and d_{AB} denote the number of wins, losses, and draws for model A against model B across all duels. Following the standard convention in Elo-based systems, we treat each draw as half a win for each player, defining effective wins $\tilde{w}_{AB} = w_{AB} + d_{AB}/2$ and effective losses $\tilde{\ell}_{AB} = \ell_{AB} + d_{AB}/2$. The negative log-likelihood of the observed outcomes is then $\mathcal{L}(\{E_m\}) = -\sum_{(A,B)} \Big[\tilde{w}_{AB} \log P(A \succ B) + \tilde{\ell}_{AB} \log P(B \succ A)\Big]$.

Since Elo ratings are only meaningful in relative terms, we fix the rating of an arbitrarily chosen model to $E_1 = 1000$ (using the model whose name comes first in lexicographical order) and optimize the remaining ratings $\{E_2, \cdots, E_n\}$ to minimize $\mathcal{L}$ via gradient descent. A difference of σ points corresponds to an expected win probability of approximately 91% for the higher-rated model; each 100-point difference represents a $\approx 64\%$ expected win rate.

Besides ranking models on their overall performance, in principle the ability of LLMs to solve hard puzzles need not be identical to their ability to propose good challenges. Thus, we also separately estimate each model's ability as a solver and as a proposer. To this end, we measure the fraction of *rounds* won as a proposer (i.e., when playing as a proposer, how often did the model find a valid puzzle that its opponent could not solve). For the solver role we report two measures: the clean solver success rate (Solv%), i.e., the fraction of *valid* opposing puzzles the model solved, and the duel-scoring Solver Win Rate (SolvW%), which counts all solver rounds and also credits the solver when the proposer forfeits the round with an invalid puzzle. We call the proposer-side measure the model's *Proposer Win Rate* (Prop%). Note that both solver measures consider the solver successful when it is able to solve the opponent's puzzle, whereas for computing the duel-level outcome we consider such situation a draw.

4 Results

We evaluated 18 models using TTG by running duels with all $18 \times 17 = 306$ ordered pairs of distinct models, repeating each ordered pairing three times to assess run-to-run stability (918 duels in total; see Table 7). Each duel ran for 14 rounds, where models alternated roles (thus, each model played each role 7 times per duel and proposed 714 puzzles overall). We compare TTG performance to performance on a number of benchmarks widely used to compare the strongest models: GPQA Diamond (Rein et al. (2023)), TextQuests (Phan et al. (2025b)), ARC-AGI v2 (Chollet et al. (2026)), SWE-Bench Pro (Jimenez et al. (2024)) and Humanity's Last Exam (HLE; Phan et al. (2025a)). We run models with the highest mode of "thinking" or "reasoning" enabled, except for the two marked low-effort variants (claude-opus-4-7:low and gpt-5.5:low) and gpt-4o as a non-reasoning baseline. We investigate the following research questions:

RQ1: Does performance on TTG correlate with other standard reasoning benchmarks?

RQ2: Do proposers show different preferences for the topics of their puzzles?

RQ3: Are strong problem solvers also good problem proposers?

RQ4: Do models use the duel's history to make puzzles more challenging in later rounds?

RQ5: Do models fail to win as proposers for being too conservative (proposing easy puzzles) or too ambitious (failing on their own puzzle)?

RQ6: How reproducible are TTG rankings across repeated tournaments?

4.1 RQ1: Correlation between TTG and expert-authored benchmarks

We first assess whether performance on TTG, a benchmarking framework with no human-designed problems, is indicative of strong performance on widely used reasoning benchmarks where questions were authored by diverse domain experts. Table 1 shows three performance metrics based on TTG duels, as well as publicly reported accuracies on other benchmarks. Prop%, Solv%, and SolvW% indicate the turn-level Proposer Win Rate, the clean solver success rate, and the duel-scoring Solver Win Rate respectively, as described in Section 3.2.

Despite the complete absence of human-authored problems in TTG, we observe that model rankings are generally consistent across benchmarks. The models leading TTG (GPT-5.4, GPT-5.5, Gemini 3.1 Pro Preview and Claude Opus 4-8) are also among the leaders on the expert-authored benchmarks, although TTG discriminates among them more sharply: GPT-5.4 and GPT-5.5 are nearly indistinguishable on HLE and GPQA-Diamond, yet separated by about 300 Elo points on TTG, with the top-2 ordering preserved in 97% of bootstrap resamples (Table 1). More generally, the five TTG leaders span 29% of the benchmark's full score range, versus 16% on HLE and only 7% on GPQA-Diamond: TTG retains discriminative power at the top of the range, where static benchmarks compress. At the other end of the spectrum, we find a similar effect: most models that struggle on TTG also tend to perform poorly on the expert-authored benchmarks. For instance, gpt-4o, our non-reasoning baseline, ranks in the bottom three on all rankings.

Table 2 presents Spearman rank correlations between TTG metrics and the reference benchmarks. TTG Elo correlates positively and significantly with every benchmark ($\rho = 0.69$ for HLE, $\rho = 0.84$ for ARC-AGI, $\rho = 0.72$ for SWE-Bench Pro, $\rho = 0.77$ for TextQuests, and $\rho = 0.77$ for GPQA-Diamond; all $p \leq 0.007$), with higher values when restricting to models that have scores on every benchmark. All 15 correlations remain significant after Bonferroni correction and under leave-one-out deletion of any single model (Appendix Table 8); match Elo also ranks models near-identically to proposer Elo (Kendall $\tau = 0.98$, differing only by two adjacent swaps). Proposer Win Rate correlates most strongly with the general reasoning benchmarks (up to $\rho = 0.90$ for ARC-AGI) and most weakly with SWE-Bench Pro ($\rho = 0.57$, $p = 0.034$), whereas Solver Win Rate has its strongest relationship with SWE-Bench Pro ($\rho = 0.80$). We find no same-lab advantage: a solver is no more likely to crack a puzzle proposed by a model from its own lab[1]

[1]Odds ratio 1.03, $+0.5$ percentage points at the base rate; lab-relabeling permutation $p = 0.87$), suggesting that duel outcomes are not driven by shared training pipelines or stylistic familiarity. We estimated this using a binomial logistic regression of round outcomes on proposer and solver fixed effects plus a same-lab indicator.

Table 2: Spearman rank correlation (ρ) between model performance on TTG and reference benchmarks. Rows marked "w/o incomplete rows" recompute the correlations after excluding models that are missing scores on one or more of the benchmarks. Bolded results are for estimates with $p < 0.05$. Table 8 in the appendix reports bootstrap confidence intervals, multiple-testing corrections, and a leave-one-out sensitivity analysis for these correlations.

	vs HLE		**vs ARC-AGI**		**vs SWE-BP**		**vs TQ**		**vs GPQA-D**	
TTG Metric	ρ	p	ρ	p	ρ	p	ρ	p	ρ	p
Elo	**+0.69**	0.007	**+0.84**	0.000	**+0.72**	0.004	**+0.77**	0.001	**+0.77**	0.000
(w/o incomplete rows)	**+0.78**	0.002	**+0.84**	0.000	**+0.78**	0.002	**+0.82**	0.001	**+0.83**	0.000
Solver Win Rate	**+0.70**	0.005	**+0.80**	0.001	**+0.80**	0.001	**+0.82**	0.000	**+0.83**	0.000
(w/o incomplete rows)	**+0.75**	0.003	**+0.80**	0.001	**+0.81**	0.001	**+0.82**	0.001	**+0.79**	0.001
Proposer Win Rate	**+0.81**	0.000	**+0.90**	0.000	**+0.57**	0.034	**+0.85**	0.000	**+0.82**	0.000
(w/o incomplete rows)	**+0.90**	0.000	**+0.90**	0.000	**+0.62**	0.024	**+0.88**	0.000	**+0.90**	0.000

In summary, we find a positive answer to our first research question: we find a *statistically significant correlation between problem-solving performance in TTG, as measured by both solver and proposer win rates, and strong performance on expert-authored reasoning benchmarks*. This is striking due to the minimal cost in designing TTG, with no human-authored problems. Furthermore, we find that problem proposal and problem solving relate differently to existing evaluations, highlighting TTG's ability to separately assess these reasoning skills.

4.2 RQ2: What are the puzzles about for different proposers?

To understand what TTG puzzles are about and whether different models exhibit diverse preferences of topics for their puzzles, we used `gpt-5.4-nano` with high reasoning to tag each puzzle in our annotated corpus with short topic tags (e.g., `modular-arithmetic`, `bit-manipulation`, `diophantine-equations`), given a list of 39 example tags, and reviewed the outputs for accuracy. Tags represent both broad (e.g., `number-theory`) and skill-specific categories (e.g., `chinese-remainder-theorem`), allowing us to understand topics at different levels of granularity, much like competitive programming problems are tagged in online platforms like Codeforces. This allowed us to obtain both a global understanding of puzzle topics, as well as a topic profile per proposer, which is summarized in Table 3. On average, each puzzle had 5.58 tags associated with it — puzzles frequently combine multiple topics by implementing a verifier that tests multiple, sometimes widely different, constraints.

Top topics differ widely across models. Although a few topics are globally common across proposers (the three most frequent tags globally are `modular-arithmetic` (49.0% of puzzles), `strings` (44.6%), and `bit-manipulation` (27.4%)), the most frequent tag for each individual model still varies markedly: Gemini 3 Flash heavily uses `bit-manipulation` (80.7% of its puzzles), GPT-5.5's most common tag is `cryptography` (50.7%), GPT-5.4 stands out for `hashing` (48.3%), and Claude Haiku and Claude Sonnet favor `number-theory` (54.6% and 61.1%).

Each model has its own signature topic. In Table 3, the "Distinctive preference" column reports the tag from each proposer that had the highest "lift" (ratio between frequency in that proposer's puzzles and global frequency). Remarkably, these tags have *no overlap* across the proposers, with lifts between $7.9\times$ and $19.0\times$ over the global rate. We see many striking examples of strong preferences not shared by other models: only Gemini 3.1 Pro writes `lambda-calculus` puzzles (11 in our corpus; no other proposer writes any; see Appendix D.15 for an example), and GPT-5.5 is responsible for *all* 28 `elliptic-curve-cryptography`-tagged puzzles, an unusually large exclusive count consistent with its top tag (`cryptography`, 51%). Other distinctive preferences include division-by-zero puzzles for Grok-4.20 (which proposed all six in the corpus), triangular-number puzzles for Claude Sonnet, RSA puzzles for DeepSeek, circular bit-rotations for Gemini 3 Flash, and systems of equations for GLM-5.2.

Puzzle topics have a long tail. Aside from the globally common topics, we also find a long tail of topics that different proposers contribute to TTG. Globally, most topics are rare: we find 2,918 tags (81% of all tags; including date-time, involution, lexicographic-ordering) that only appear in 5 or fewer puzzles. The most frequent topic for each model that is only used by that proposer is reported on the last column of Table 3. We find several niches that are filled by exactly one model, such as `slicing-permutation` from GPT-5.4, `utf-8-decoding` from DeepSeek, `two-complement` arithmetic from Qwen3-Coder, and `math-isqrt` from Claude Sonnet. Overall, we both observe (1) a wide range of topics arising from the simple structure of TTG, and (2) that proposers strongly show distinct preferences for particular topics besides common ones. Tags also track difficulty: against our three uniform solvers, `hmac` (36%), `byte-manipulation` (38%) and `cryptography` (45%) puzzles have the lowest solve rates among common tags, while `digit-manipulation` (94%) and `perfect-squares` (93%) puzzles are the easiest — consistent with hash- and cipher-style puzzles being the weapon of choice of the strongest proposers. We list diverse interesting examples in Appendix D, and release our puzzle labels together with the data.

Programming or reasoning? A natural concern is that TTG, built on programming puzzles, measures programming skill while claiming to measure reasoning. However, no software-engineering topic appears among the most frequent tags. Instead, 85% of puzzles carry at least one mathematics, cryptography, or self-reference tag. Puzzles encode diverse topics from Pell equations (Appendix D.3) to Church numerals (Appendix D.15) to integer factorization (Appendix D.1). Second, the correlations

of Table 2 show that proposing correlates most strongly with general-reasoning benchmarks (ARC-AGI, $\rho = 0.90$) and most weakly with the coding benchmark SWE-Bench Pro ($\rho = 0.57$).

4.3 RQ3: Are strong solvers also good proposers?

Unlike existing evaluations, TTG allows us to measure both the aptitude for solving puzzles as well as for creating challenging ones. Here, we investigate how these abilities are related: do models that excel at solving puzzles also excel at proposing challenging problems?

We find a strong positive correlation between solver and proposer win rates (Spearman $\rho = 0.81$, $p < 10^{-4}$). As shown in Table 1, models that rank highly as solvers tend to also rank highly as proposers: for instance, GPT-5.4 leads in both roles. This suggests that the reasoning capabilities required to solve difficult puzzles overlap with those needed to construct them. However, we also observe a notable asymmetry when examining absolute performance levels. Across all rounds, proposers either fail to generate a valid puzzle, or to stump their opponents, 83.7% of the time. Even the second-strongest proposer, GPT-5.5, wins only 38.2% of rounds as a proposer, compared to a near-perfect 97.6% as solver; GPT-5.4, the strongest proposer, is the sole outlier at 60.4%. At the other extreme, Claude Haiku 4.5 wins only 1.8% of its rounds as proposer, despite itself solving 69.5% of the valid puzzles proposed by its opponents (see Table 5). The stumping puzzles are nonetheless genuinely solvable: 95.0% of all valid puzzles were solved by *someone* — either by the opponent during the duel, or by the proposer itself when re-solving its puzzle cold, without access to its private notes ("Either S" in Table 5) — and the remaining 5% concentrate among the strongest proposers' hardest puzzles, consistent with genuine difficulty rather than unsolvability.

This disparity reveals that **creating good puzzles is substantially harder than solving them**. Models frequently either propose puzzles that their opponents can solve, or fail to solve their own problems. The proposer role thus also represents a highly discriminating test of reasoning capability, with a wider spread in performance across models.

4.4 RQ4: Can proposers increase difficulty using the duel's history?

In TTG duels, proposers can see the whole history of interactions. This design decision was based on the hypothesis that proposers should adapt the difficulty of their problems as they observe either their own failures in proposing valid puzzles with correct solutions (likely indicating they should be less ambitious) or seeing the opponent's successes as solvers (pressuring them to attempt to propose harder puzzles). To assess whether such an adaptation is taking place, we use the performance of three uniform solvers — Qwen3.8 27B (run locally), Grok 4 Fast Reasoning, and GPT-5.4 Nano, spanning overall solve rates from 57% to 81% — on all puzzles extracted from all duels as an assessment of puzzle difficulty beyond the two competing models. When solving a puzzle, as also happens during duels, models are only given general instructions on format and the puzzle itself, but no other previous context. We evaluate each solver's success rate for puzzles proposed across turns from 1 to 14 in all duels. Figure 2 shows the results. Solve rates decline across turns for all three solvers, and the sensitivity is monotone in solver strength: the weakest solver (Grok 4 Fast) loses 0.81 percentage points of solve rate per turn — over 10 points across a duel — while the strongest (GPT-5.4 Nano) loses only 0.33 points per turn. The duels themselves show the same trend: the number of puzzles solved by the duel opponent falls from 594 in the first turn to 490 in the last (Appendix Table 6). Overall, we find that proposers measurably increase difficulty across turns, and this is felt most strongly by weaker solvers. The uniform solvers also validate the duel outcomes themselves: their per-proposer failure rates agree closely with each other and with the rate at which each proposer stumped its opponents in the arena (Spearman ρ between 0.87 and 0.98 per solver, and $\rho = 0.99$ for their average; $n = 18$ proposers), indicating that duel outcomes measure puzzle difficulty rather than opponent-specific quirks.

4.5 RQ5: Are failing proposers too conservative or too ambitious?

When playing as a proposer, a model can fail to win the turn in one of two ways: (1) its own solution is wrong (giving a point to its opponent), or (2) the opponent's solution is correct (ending the turn in a draw). In the former case, the proposer was arguably overly confident in its ability to solve the puzzle it proposed. In the latter case, the puzzle was too easy for the opponent. How do models

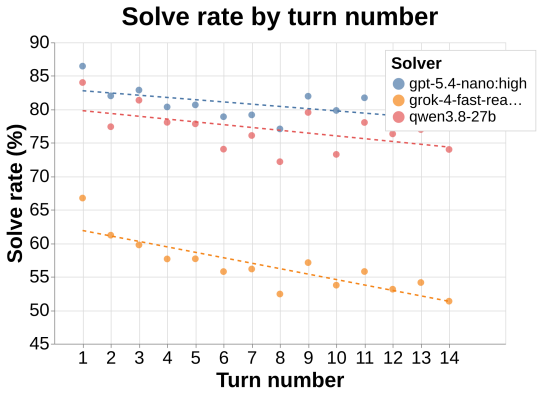

Figure 2: Solve rates of three uniform solvers (Qwen3.8 27B, Grok 4 Fast Reasoning, and GPT-5.4 Nano) on puzzles from all duels, by the turn in which the puzzle was proposed. Regression fits show puzzles becoming slightly harder each turn.

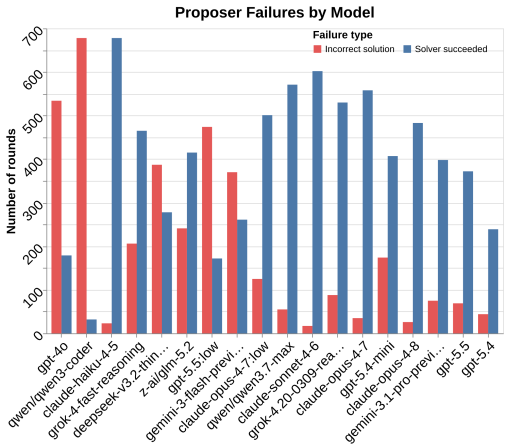

Figure 3: Failure modes for proposers. Did the model's own sample solution fail (red) or did the solver succeed in solving the puzzle (blue)? Each model had 714 puzzle proposals.

behave with respect to these two failure modes as proposers? Figure 3 shows the number of failures as proposers of each of these two categories and Table 5 shows overall performance as a proposer. Across all models, we see that failing at providing hard-enough puzzles is more common (55.6% of all puzzle proposals are solved by the opponent) than providing a too-hard puzzle with an incorrect sample solution (28.2%). Data, however, differs a lot by model. Gemini 3 Flash Preview exhibits high overconfidence (51.2% penalties for an incorrect solution to its own puzzle). Sonnet 4.6, on the other hand, usually writes puzzles that are too easy – its competitors can solve Sonnet's puzzles 82.0% of the time. We find recurring failure patterns: arithmetic slips in the proposer's own construction (Appendix D.2, D.4), mismatches between the proposer's mathematical model and Python's actual semantics, and subtle representation issues such as string re-escaping (Appendix D.11).

4.6 RQ6: How reproducible are TTG rankings?

Repeating every ordered pairing three times lets us measure reproducibility directly. At the match level, 61% of pairings produce the identical outcome in all three repetitions and only 17.6% ever reverse; the mean within-pairing variance of the match score is 0.049, against 0.25 for a coin flip (Appendix Table 7). At the ranking level, the bootstrap intervals of Table 1 preserve the top-2 ordering in 97% of resamples, with uncertainty concentrated mid-table. At the tournament level, the three repetitions form independent tournaments whose Elo rankings agree at Spearman $\rho = 0.95$ pairwise, implying a Spearman–Brown reliability of 0.98 for the full tournament. TTG's own noise therefore attenuates the correlations of Table 2 by at most 0.01: the gap to static benchmarks reflects differences in what is measured, not ranking instability — and a single repetition already suffices to recover the ranking, at a third of our reported cost.

5 Conclusion

We propose a competition-style evaluation of LLM reasoning with no human-authored problems. For 18 models, performance on TTG correlates significantly with other standard reasoning benchmarks that were highly expensive to design. With 18 models, we were able to run duels between all pairs of models (three repetitions of each ordered pairing). In the future, BT ratings can be inferred from sparse subsets of all duels. This will become important as we add new models to TTG and dueling against all previous models becomes infeasible.

AI Disclosure Statement

We used generative AI tools for assisting with code implementation, data analysis/visualization, proof-reading and drafting of the paper. LLMs are also the object of our analysis. Generative AI tools were not used for brainstorming, suggesting experiments or hypotheses, or other higher-level research tasks. We have reviewed all AI-assisted work. We take full responsibility for the final content of this work, including text and artifacts produced with the aid of generative AI.

References

Monica Azzolini. There were no medals to be won: Scientific duels in the italian renaissance. *Nuncius*, 34(2):258–283, 2019.

BIG-bench authors. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=uyTL5Bvosj.

Wayne Chi, Valerie Chen, Anastasios Nikolas Angelopoulos, Wei-Lin Chiang, Aditya Mittal, Naman Jain, Tianjun Zhang, Ion Stoica, Chris Donahue, and Ameet Talwalkar. Copilot arena: A platform for code llm evaluation in the wild. *arXiv preprint arXiv:2502.09328*, 2025.

Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Banghua Zhu, Hao Zhang, Michael Jordan, Joseph E Gonzalez, et al. Chatbot arena: An open platform for evaluating llms by human preference. In *Forty-first International Conference on Machine Learning*, 2024.

Francois Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. Arc-agi-2: A new challenge for frontier ai reasoning systems, 2026. URL https://arxiv.org/abs/2505.11831.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try ARC, the AI2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Kefan Dong and Tengyu Ma. Stp: Self-play llm theorem provers with iterative conjecturing and proving. *arXiv preprint arXiv:2502.00212*, 2025.

Elliot Glazer, Ege Erdil, Tamay Besiroglu, Diego Chicharro, Evan Chen, Alex Gunning, Caroline Falkman Olsson, Jean-Stanislas Denain, Anson Ho, Emily de Oliveira Santos, et al. Frontiermath: A benchmark for evaluating advanced mathematical reasoning in ai. *arXiv preprint arXiv:2411.04872*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2021.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Yitao Long, Yuru Jiang, Hongjun Liu, Yilun Zhao, Jingchen Sun, Yiqiu Shen, Chen Zhao, Arman Cohan, and Dennis Shasha. Puzzleplex: Benchmarking foundation models on reasoning and planning with puzzles. *arXiv preprint arXiv:2510.06475*, 2025.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Scale AI, and Center for AI Safety. Humanity's last exam. *arXiv preprint arXiv:2501.14249*, 2025a.

Long Phan, Mantas Mazeika, Andy Zou, and Dan Hendrycks. Textquests: How good are llms at text-based video games?, 2025b. URL `https://arxiv.org/abs/2507.23701`.

Gabriel Poesia, David Broman, Nick Haber, and Noah Goodman. Learning formal mathematics from intrinsic motivation. *Advances in Neural Information Processing Systems*, 37:43032–43057, 2024.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Tal Schuster, Ashwin Kalyan, Oleksandr Polozov, and Adam Tauman Kalai. Programming puzzles. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2021.

Fabio Toscano. *The Secret Formula: How a Mathematical Duel Inflamed Renaissance Italy and Uncovered the Cubic Equation*. Princeton University Press, 2020.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Yue Wu, Xuan Tang, Tom M Mitchell, and Yuanzhi Li. Smartplay: A benchmark for llms as intelligent agents. *arXiv preprint arXiv:2310.01557*, 2023.

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T. Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. MetaMath: Bootstrap your own mathematical questions for large language models. In *International Conference on Learning Representations*, 2024.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. STaR: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. *arXiv preprint arXiv:2505.03335*, 2025.

A Additional Results

The Elo confidence intervals in Table 1 are computed from a 1000-sample bootstrap over match outcomes; intervals are relative to the anchor model (claude-haiku-4-5, fixed at 1000), whose own interval is degenerate by construction.

Table 3 shows the per-proposer puzzle tag profile discussed in Section 4.2.

Table 3: Per-proposer puzzle tag profile across all duels. *Distinctive preference*: among tags appearing ≥ 5 times for the proposer, the one with the largest "lift", i.e., the ratio of the tag's per-proposer frequency to its global frequency across all proposers; an $N\times$ multiplier means the proposer uses the tag N times more often than the average proposer. *Exclusive tag*: the most-used tag whose appearances *all* come from that one proposer (puzzle count in parentheses).

Proposer	Distinctive preference	Exclusive tag
claude-haiku-4-5	`factorials` (1.4%, 12.9×)	`arithmetic-exponentiation` (3)
claude-opus-4-7	`string-to-digits` (2.1%, 5.9×)	`involution` (2)
claude-opus-4-7:low	`range-checking` (0.7%, 9.0×)	`arithmetic-algebra` (2)
claude-opus-4-8	`modular-square-roots` (0.7%, 12.9×)	`code-points` (2)
claude-sonnet-4-6	`prime-testing` (1.3%, 8.1×)	`math-isqrt` (3)
deepseek-v3.2-thinking	`rsa` (1.3%, 7.7×)	`base64-encoding` (2)
gemini-3-flash-preview	`circular-rotation` (2.8%, 16.4×)	`unsigned-integer` (3)
gemini-3.1-pro-preview	`lambda-calculus` (1.5%, 18.0×)	`lambda-calculus` (11)
gpt-4o	`case-insensitive-comparison` (0.7%, 12.9×)	`vowel-detection` (2)
gpt-5.4	`slicing-permutation` (0.7%, 18.0×)	`slicing-permutation` (5)
gpt-5.4-mini	`cipher` (1.5%, 11.6×)	`list-indexing` (4)
gpt-5.5	`elliptic-curve-cryptography` (3.8%, 18.0×)	`elliptic-curve-cryptography` (27)
gpt-5.5:low	`hash-like-mixing` (0.8%, 10.8×)	`obfuscated-validation` (3)
grok-4-fast-reasoning	`ord-chr` (1.4%, 8.2×)	`polynomial-iteration` (3)
grok-4.20-0309-reasoning	`division-by-zero` (0.7%, 18.0×)	`division-by-zero` (5)
qwen/qwen3-coder	`binary-parsing` (1.8%, 13.0×)	`two-complement` (3)
qwen/qwen3.7-max	`ascii-ord-mapping` (1.0%, 14.0×)	`leap-year-calculation` (2)
z-ai/glm-5.2	`systems-of-equations` (1.1%, 7.6×)	`equation-system` (2)

Table 4 shows puzzle difficulty by each model as a proposer, as evaluated by the ability of three uniform solvers (Qwen3.8 27B, Grok 4 Fast Reasoning, and GPT-5.4 Nano) to solve them.

Table 4: Uniform solvers by proposer model. After completing all duels, we mined puzzles from their logs and had three models attempt them outside the duel context: Qwen3.8 27B, Grok 4 Fast Reasoning, and GPT-5.4 Nano. We only used valid puzzles, i.e. those that had correct sample solutions, and we show each solver's solve rate for puzzles by their original proposer.

Proposer Model	gpt-5.4-nano:high	grok-4-fast-reasoning	qwen3.8-27b	Total
claude-haiku-4-5	680/691 (98%)	633/689 (92%)	673/689 (98%)	1986/2069 (96%)
gpt-4o	177/180 (98%)	161/180 (89%)	178/180 (99%)	516/540 (96%)
grok-4-fast-reasoning	477/506 (94%)	464/503 (92%)	453/503 (90%)	1394/1512 (92%)
qwen/qwen3-coder	31/36 (86%)	31/35 (89%)	34/35 (97%)	96/106 (91%)
z-ai/glm-5.2	444/473 (94%)	351/473 (74%)	412/473 (87%)	1207/1419 (85%)
qwen/qwen3.7-max	615/659 (93%)	487/659 (74%)	563/659 (85%)	1665/1977 (84%)
claude-sonnet-4-6	619/697 (89%)	511/697 (73%)	612/697 (88%)	1742/2091 (83%)
claude-opus-4-7:low	532/587 (91%)	402/587 (68%)	511/587 (87%)	1445/1761 (82%)
deepseek-v3.2-thinking	290/327 (89%)	226/327 (69%)	283/327 (87%)	799/981 (81%)
claude-opus-4-7	602/679 (89%)	450/679 (66%)	591/679 (87%)	1643/2037 (81%)
grok-4.20-0309-reasoning	562/624 (90%)	413/622 (66%)	518/622 (83%)	1493/1868 (80%)
gemini-3-flash-preview	311/344 (90%)	176/344 (51%)	259/344 (75%)	746/1032 (72%)
gpt-5.4-mini	450/540 (83%)	238/540 (44%)	409/540 (76%)	1097/1620 (68%)
claude-opus-4-8	514/688 (75%)	337/688 (49%)	526/688 (76%)	1377/2064 (67%)
gpt-5.5:low	209/240 (87%)	58/240 (24%)	153/240 (64%)	420/720 (58%)
gemini-3.1-pro-preview	425/639 (67%)	140/639 (22%)	461/639 (72%)	1026/1917 (54%)
gpt-5.5	284/644 (44%)	135/644 (21%)	242/644 (38%)	661/1932 (34%)
gpt-5.4	218/670 (33%)	13/670 (2%)	232/670 (35%)	463/2010 (23%)
Total	7440/9224 (81%)	5226/9216 (57%)	7110/9216 (77%)	19776/27656 (72%)

Table 5: Puzzle outcomes by model and role. The left columns show performance in the solver role (number of puzzles solved and failed to solve). The middle columns show performance in the proposer role (number of invented puzzles solved, unsolved (% of all correct (non-penalized) puzzles), puzzles with incorrect sample solutions which the proposer was penalized for). In a second run, we had every model try to solve its own puzzle without the prior context (i.e. without seeing its notes). "Self S" shows the proportion of the model's proposed puzzles that it could itself solve in the separate context, and "Either S" shows the proportion of puzzles that could be solved by either the model itself in a separate context or the model's opponent in the duel. This is to ensure that puzzles are solvable.

Model	Solv S (%)	Solv F	Prop S	Prop U (%)	Prop P (%)	Self S (%)	Either S (%)
gpt-5.4	450 (98.9%)	5	239	431 (64.3%)	44 (6.2%)	555 (82.8%)	571 (85.2%)
gpt-5.5	482 (97.6%)	12	372	273 (42.3%)	69 (9.7%)	618 (96.0%)	626 (97.2%)
claude-opus-4-7	458 (92.0%)	40	558	121 (17.8%)	35 (4.9%)	621 (91.5%)	651 (95.9%)
qwen/qwen3.7-ma...	475 (91.5%)	44	571	88 (13.4%)	55 (7.7%)	614 (93.2%)	645 (97.9%)
gpt-5.5:low	442 (89.3%)	53	172	68 (28.3%)	474 (66.4%)	231 (96.2%)	238 (99.2%)
claude-sonnet-4...	461 (88.5%)	60	602	95 (13.6%)	17 (2.4%)	638 (91.5%)	663 (95.1%)
claude-opus-4-8	412 (88.2%)	55	483	205 (29.8%)	26 (3.6%)	582 (84.6%)	614 (89.2%)
gemini-3.1-pro-...	424 (85.7%)	71	398	241 (37.7%)	75 (10.5%)	565 (88.4%)	593 (92.8%)
claude-opus-4-7...	437 (85.5%)	74	501	88 (14.9%)	125 (17.5%)	545 (92.8%)	568 (96.8%)
gpt-5.4-mini	392 (84.5%)	72	407	133 (24.6%)	174 (24.4%)	432 (80.0%)	485 (89.8%)
z-ai/glm-5.2	422 (80.8%)	100	415	58 (12.3%)	241 (33.8%)	443 (93.7%)	469 (99.2%)
grok-4.20-0309-...	400 (79.8%)	101	530	96 (15.3%)	88 (12.3%)	554 (89.1%)	599 (96.3%)
gemini-3-flash-...	374 (72.9%)	139	261	83 (24.1%)	370 (51.8%)	297 (86.3%)	327 (95.1%)
deepseek-v3.2-t...	371 (71.9%)	145	278	49 (15.0%)	387 (54.2%)	253 (77.4%)	303 (92.7%)
claude-haiku-4-...	374 (69.5%)	164	678	13 (1.9%)	23 (3.2%)	680 (98.7%)	688 (99.9%)
grok-4-fast-rea...	338 (62.0%)	207	465	43 (8.5%)	206 (28.9%)	474 (94.2%)	497 (98.8%)
qwen/qwen3-code...	302 (53.4%)	264	32	4 (11.1%)	678 (95.0%)	32 (91.4%)	35 (100.0%)
gpt-4o	127 (20.8%)	484	179	1 (0.6%)	534 (74.8%)	162 (90.0%)	180 (100.0%)
Total	7141 (77.4%)	2090	7141	2090 (22.6%)	3621 (28.2%)	8296 (90.0%)	8752 (95.0%)

Table 6: Puzzle Outcomes By Turn Number

Turn Number	Solved	Unsolved	Penalty (%)
1	594	117	207 (22.5%)
2	540	133	245 (26.7%)
3	533	139	246 (26.8%)
4	517	135	266 (29.0%)
5	507	146	265 (28.9%)
6	480	160	278 (30.3%)
7	510	148	260 (28.3%)
8	492	167	259 (28.2%)
9	511	150	257 (28.0%)
10	478	162	278 (30.3%)
11	504	153	261 (28.4%)
12	491	160	267 (29.1%)
13	494	146	278 (30.3%)
14	490	174	254 (27.7%)

B Cost

Table 9 shows the cost of our tournament (all 918 duels).

C Sources for public results on existing benchmarks

For HLE, TextQuests, SWE-Bench Pro, and ARC-AGI, we used numbers from the Center for AI Safety's AI leaderboard (`https://dashboard.safe.ai/`). For GPQA Diamond, we used the leaderboard provided by Artificial Analysis (`https://artificialanalysis.ai/evaluations/gpqa-diamond`).

Table 7: Run-to-run stability of match outcomes. Every ordered pairing (A proposes first, against B) is repeated three times in our tournament. Per model, over the pairings it took part in: *Identical* is the share of pairings with the same outcome in all three repetitions; *Reversed* is the share where the model both won and lost the same pairing across repetitions; *Score var.* is the mean within-pairing variance of the match score (win $= 1$, draw $= 0.5$, loss $= 0$) — a fair coin between win and loss would give 0.25. Overall, 61% of pairings are perfectly repeatable and only 18% ever reverse; most residual variability is between a win and a draw.

Model	**Pairings**	**Identical (%)**	**Reversed (%)**	**Score var.**
gpt-4o	34	97.1	0.0	0.002
qwen/qwen3-coder	34	91.2	0.0	0.005
gpt-5.4	34	88.2	5.9	0.016
deepseek-v3.2-thinking	34	67.6	17.6	0.044
gpt-5.5	34	67.6	17.6	0.044
gemini-3.1-pro-preview	34	64.7	26.5	0.059
gpt-5.5:low	34	61.8	14.7	0.046
gemini-3-flash-preview	34	58.8	23.5	0.059
grok-4-fast-reasoning	34	55.9	20.6	0.054
claude-opus-4-8	34	52.9	23.5	0.062
gpt-5.4-mini	34	52.9	32.4	0.080
grok-4.20-0309-reasoning	34	52.9	32.4	0.077
qwen/qwen3.7-max	34	52.9	8.8	0.041
claude-haiku-4-5	34	50.0	23.5	0.062
z-ai/glm-5.2	34	50.0	29.4	0.072
claude-opus-4-7:low	34	47.1	14.7	0.054
claude-opus-4-7	34	44.1	11.8	0.049
claude-sonnet-4-6	34	44.1	14.7	0.051
Total	306	61.1	17.6	0.049

Table 8: Robustness of the Spearman correlations between TTG Elo ratings (match, proposer, and solver Elo) and external benchmarks. For each pair we report the correlation ρ, a bootstrap 95% confidence interval (2000 resamples over models), the raw p-value alongside Bonferroni- and Benjamini-Hochberg-adjusted values across all 15 tests, and a leave-one-out (LOO) sensitivity analysis: the range of ρ when any single model is removed, the model whose removal moves ρ the most (with the resulting ρ), and whether the correlation remains significant ($p < 0.05$) in every LOO subset.

Elo	**Bench.**	n	ρ	**95% CI**	p	p_{Bonf}	p_{BH}	**LOO ρ range**	**Most infl. model (ρ w/o)**	**sig. all LOO**
Match	HLE	14	$+0.688$	$[+0.26, +0.91]$	0.0065	0.0981	0.0070	$[+0.61, +0.78]$	z-ai/glm-5.2 ($+0.78$)	yes
	ARC-AGI	13	$+0.835$	$[+0.50, +0.97]$	0.0004	0.0057	0.0007	$[+0.79, +0.89]$	gemini-3-flash... ($+0.89$)	yes
	SWE-BP	14	$+0.719$	$[+0.36, +0.90]$	0.0038	0.0567	0.0047	$[+0.65, +0.78]$	gpt-4o ($+0.65$)	yes
	TQ	14	$+0.767$	$[+0.31, +0.96]$	0.0014	0.0205	0.0021	$[+0.71, +0.84]$	gemini-3-flash... ($+0.84$)	yes
	GPQA-D	17	$+0.771$	$[+0.42, +0.93]$	0.0003	0.0044	0.0007	$[+0.72, +0.81]$	gpt-4o ($+0.72$)	yes
Proposer	HLE	14	$+0.688$	$[+0.24, +0.91]$	0.0065	0.0981	0.0070	$[+0.61, +0.78]$	z-ai/glm-5.2 ($+0.78$)	yes
	ARC-AGI	13	$+0.835$	$[+0.52, +0.97]$	0.0004	0.0057	0.0007	$[+0.79, +0.89]$	gemini-3-flash... ($+0.89$)	yes
	SWE-BP	14	$+0.719$	$[+0.31, +0.91]$	0.0038	0.0567	0.0047	$[+0.65, +0.78]$	gpt-4o ($+0.65$)	yes
	TQ	14	$+0.767$	$[+0.33, +0.95]$	0.0014	0.0205	0.0021	$[+0.71, +0.84]$	gemini-3-flash... ($+0.84$)	yes
	GPQA-D	17	$+0.768$	$[+0.40, +0.93]$	0.0003	0.0048	0.0007	$[+0.72, +0.81]$	gpt-4o ($+0.72$)	yes
Solver	HLE	14	$+0.815$	$[+0.43, +0.96]$	0.0004	0.0057	0.0007	$[+0.77, +0.89]$	z-ai/glm-5.2 ($+0.89$)	yes
	ARC-AGI	13	$+0.901$	$[+0.64, +1.00]$	0.0000	0.0004	0.0002	$[+0.87, +0.93]$	grok-4.20-0309... ($+0.93$)	yes
	SWE-BP	14	$+0.684$	$[+0.26, +0.90]$	0.0070	0.1056	0.0070	$[+0.60, +0.73]$	gpt-4o ($+0.60$)	yes
	TQ	14	$+0.873$	$[+0.56, +0.98]$	0.0000	0.0007	0.0002	$[+0.84, +0.92]$	grok-4.20-0309... ($+0.92$)	yes
	GPQA-D	17	$+0.870$	$[+0.58, +0.97]$	0.0000	0.0001	0.0001	$[+0.84, +0.91]$	gpt-5.4-mini ($+0.91$)	yes

Table 9: Cost of running our 918 duels.

Model	In $/M	Out $/M	In tokens	Out tokens	Calls	Cost (USD)
claude-haiku-4-5	1.00	5.00	2 011 860	17 406 364	1252	$89.04
claude-opus-4-7	5.00	25.00	2 343 607	8 134 940	1212	$215.09
claude-opus-4-7:low	5.00	25.00	2 176 891	3 328 092	1225	$94.09
claude-opus-4-8	5.00	25.00	2 779 193	8 774 103	1181	$233.25
claude-sonnet-4-6	3.00	15.00	2 323 996	20 482 467	1235	$314.21
deepseek-v3.2-thinking	0.27	0.40	1 674 783	17 535 380	1230	$7.46
gemini-3-flash-preview	0.50	3.00	2 432 448	19 504 634	1227	$59.73
gemini-3.1-pro-preview	2.00	12.00	2 671 267	26 454 751	1209	$322.80
gpt-4o	2.50	10.00	1 691 330	463 887	1325	$8.87
gpt-5.4	2.50	15.00	3 527 768	26 120 515	1169	$400.63
gpt-5.4-mini	0.75	4.50	2 307 606	14 188 898	1178	$65.58
gpt-5.5	5.00	30.00	3 332 889	10 766 055	1208	$339.65
gpt-5.5:low	5.00	30.00	1 790 731	1 396 065	1209	$50.84
grok-4-fast-reasoning	0.20	0.50	1 476 870	2 749 291	1259	$1.67
grok-4.20-0309-reasoning	1.25	2.50	1 692 488	11 451 543	1215	$30.74
qwen/qwen3-coder	0.30	1.00	2 044 030	3 083 135	1280	$3.70
qwen/qwen3.7-max	1.48	4.42	2 212 006	17 707 516	1233	$81.62
z-ai/glm-5.2	0.65	2.04	1 805 295	13 636 136	1236	$29.01
Total			40 295 058	223 183 772		**$2347.97**

D Puzzle Examples

Here we show several interesting examples of puzzles designed and solved by various models.

The examples in this appendix:

- D.1 Mersenne Trap
- D.2 Fermat's Pell Equation
- D.3 Square-Triangular Numbers
- D.4 Mini-RSA Self-Fail
- D.5 RSA Fixed Point
- D.6 One Hallucinated Hex Digit
- D.7 Bitcoin Genesis Nonce
- D.8 Password in the Exponents
- D.9 The Pure-Alphabetic Quine
- D.10 A Novel Quine
- D.11 The 49-Character Quine
- D.12 Solved but Lost
- D.13 Brainfuck Compression
- D.14 Writing Machine Code
- D.15 Church Numeral 65536
- D.16 The Unsatisfiable Puzzle
- D.17 A Hash Oracle

D.1 Mersenne Trap

Six innocuous lines conceal an integer factorization problem; the intended divisor is the Mersenne prime $2^{127} - 1$.

```python
def mystery(x):
    if type(x) is not int or x <= 0 or x.bit_length() != 127:
        return False
    q, r = divmod(
        170481465828581158479375834498455752437849622286529887237097945306493374451,
        x,
    )
    return r == 0 and q.bit_length() == 120
```

Proposed by: gpt-5.4 **Solved by:** grok-4.20-0309-reasoning
Outcome: Solver Failed **Attempt:** `204489180111330397004928700368315835865292159555201088512213827622210096 9` **Solution:** `170141183460469231731687303715884105727`

D.2 Fermat's Pell Equation

The proposer correctly recalls the theory behind Fermat's 1657 challenge, then loses the round to a big-integer multiplication error.

```python
def mystery(x):
    if type(x) is not str:
        return False

    try:
        u, v = map(int, x.split(','))
    except ValueError:
        return False

    if u <= 0 or v <= 10**17:
        return False

    return u**2 - 61 * v**2 == 1
```

Proposed by: gemini-3.1-pro-preview **Solved by:** gpt-5.5
Outcome: Sample Solution Incorrect **Attempt:** -- **Solution:** `"6239781660251608001,798920404068305040"`

D.3 Square-Triangular Numbers

Neither the checker nor the prompt mentions Pell equations, yet both proposer and solver independently reach for them.

```python
def mystery(x):
    if type(x) is not int or isinstance(x, bool):
        return False

    # x must live in this large window
    if not (10**18 < x < 10**60):
        return False

    def _isqrt(n):
        if n < 0:
            return -1
        if n == 0:
            return 0
        r = 1 << ((n.bit_length() + 1) // 2)
        while True:
            nr = (r + n // r) // 2
            if nr >= r:
                break
            r = nr
        return r

    # first structural gate
    a = _isqrt(x)
    if a * a != x:
        return False

    # second structural gate
    t = 8 * x + 1
    b = _isqrt(t)
    if b * b != t:
        return False

    return True
```

Proposed by: claude-opus-4-8 **Solved by:** gpt-5.4
Outcome: Solved **Attempt:** `2507180834294496361` **Solution:** `__import__('functools').reduce(lambda t,_:(t[1],34*t[1]-t[0]+2),range(20),(0,1))[0]`

D.4 MINI-RSA SELF-FAIL

A five-line RSA puzzle is invalidated because the proposer cannot do its own modular exponentiation in its head.

```python
def mystery(x):
    if isinstance(x, str):
        try:
            x = int(x)
        except ValueError:
            return False
    if not isinstance(x, int):
        return False
    n = 1022117
    e = 5
    c = 754241
    return pow(x, e, n) == c
```

Proposed by: deepseek-v3.2-thinking **Solved by:** claude-haiku-4-5
Outcome: Sample Solution Incorrect **Attempt:** -- **Solution:** `123456`

D.5 RSA FIXED POINT

The intended solution requires CRT reasoning about fixed points; the solver instead notices that $x = 1$ satisfies the spec.

```python
def mystery(x):
    # Convert to integer if possible
    if isinstance(x, str):
        try:
            x = int(x)
        except ValueError:
            return False
    elif not isinstance(x, int):
        return False
    if x <= 0:
        return False
    n = 1000036000099
    e = 65537
    # Check if x^e mod n equals x
    return pow(x, e, n) == x
```

Proposed by: deepseek-v3.2-thinking **Solved by:** gpt-5.5
Outcome: Solved **Attempt:** `1` **Solution:** `233341700023`

D.6 ONE HALLUCINATED HEX DIGIT

The construction is mathematically sound, but the proposer reproduces a 64-digit curve constant from memory with a single wrong digit.

```python
def mystery(x):
    if type(x) is not int:
        return False
    if not (0 < x < 2**256):
        return False

    m = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F

    P = (
        0xC6047F9441ED7D6D3045406E95C07CD85AEEF0F32823C82850257530949F2818,
        0x1AE168FEA63DC339A3C58419466CEAEEF7F632653266D0E1236431A950CFE52A,
    )
    Q = (
        0xF9308A019258C31049344F85F89D5229B531C845836F99B08601F113BCE036F9,
        0x388F7B0F632DE8140FE337E62A37F3566500A99934C2231B6CB9FD7584B8E672,
    )

    def add(A, B):
        if A is None:
            return B
        if B is None:
            return A
```

```
        ax, ay = A
        bx, by = B

        if ax == bx and (ay + by) % m == 0:
            return None

        if A == B:
            s = (3 * ax * ax * pow(2 * ay, -1, m)) % m
        else:
            s = ((by - ay) * pow(bx - ax, -1, m)) % m

        cx = (s * s - ax - bx) % m
        cy = (s * (ax - cx) - ay) % m
        return (cx, cy)

    def mul(k, A):
        R = None
        while k:
            if k & 1:
                R = add(R, A)
            A = add(A, A)
            k >>= 1
        return R

    return mul(x, P) == Q
```

Proposed by: gpt-5.5 **Solved by:** claude-haiku-4-5

Outcome: Sample Solution Incorrect **Attempt:** -- **Solution:** `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A2`

D.7 Bitcoin Genesis Nonce

The puzzle is the genesis block header in disguise; the solver recalls Satoshi's nonce rather than brute-forcing 2^{32} hashes. Memorization-based puzzles do occur, but this round shows why they are usually a poor strategy: whatever constants one model has memorized, another model may have memorized too.

```
import hashlib

def mystery(x):
    if not isinstance(x, int) or isinstance(x, bool):
        return False
    if not (0 <= x < 2**32):
        return False

    prefix = bytes.fromhex(
        "01000000"
        "0000000000000000000000000000000000000000000000000000000000000000"
        "3ba3edfd7a7b12b27ac72c3e67768f617fc81bc3888a51323a9fb8aa4b1e5e4a"
        "29ab5f49"
        "ffff001d"
    )

    h = hashlib.sha256(hashlib.sha256(prefix + x.to_bytes(4, "little")).digest()).digest()
    return h[::-1].hex() == (
        "000000000019d668"
        "9c085ae165831e934"
        "ff763ae46a2a6c17"
        "2b3f1b60a8ce26f"
    )
```

Proposed by: gpt-5.5 **Solved by:** claude-opus-4-7

Outcome: Solved **Attempt:** `2083236893` **Solution:** `2083236893`

D.8 Password in the Exponents

The solver peels three layers—factorization encoding, ROT13, and a known RFC test vector—to reconstruct a 200-digit integer.

```
def mystery(x):
    import codecs
```

```
    import hashlib
    import hmac

    if type(x) is not int:
        return False

    if x <= 1:
        return False

    primes = (
        2, 3, 5, 7, 11, 13, 17, 19,
        23, 29, 31, 37, 41, 43, 47, 53,
        59, 61, 67, 71, 73, 79, 83, 89,
    )

    raw = bytearray()
    for p in primes:
        e = 0
        while x % p == 0:
            x //= p
            e += 1
            if e > 122:
                return False

        if not (65 <= e <= 90 or 97 <= e <= 122):
            return False
        raw.append(e)

    if x != 1:
        return False

    if (sum(raw), raw[0] ^ raw[-1], raw.count(ord("f"))) != (2237, 0x12, 4):
        return False

    s = raw.decode("ascii")
    y = codecs.decode(s, "rot_13").encode("ascii")

    if y[:8] != y[16:] or y[:8].upper() != y[8:16]:
        return False

    want = bytes.fromhex(
        "3d2eec4fe41c849b80c8d83662c0e44a"
        "8b291a964cf2f07038"
    )

    got = hashlib.pbkdf2_hmac(
        "sha1",
        y,
        b"saltSALTsaltSALTsaltSALTsaltSALTsalt",
        4096,
        len(want),
    )

    return hmac.compare_digest(got, want)
```

Proposed by: gpt-5.5 **Solved by:** claude-sonnet-4-6

Outcome: Solved **Attempt:** `2**99 * 3**110 * 5**102 * 7**102 * 11**106 * 13**98 * 17**101 * 19**113 * 23**67 * 29**78 * 31**70 * 37**70 * 41**74 * 43**66 * 47**69 * 53**81 * 59**99 * 61**110 * 67**102 * 71**102 * 73**106 * 79**98 * 83**101 * 89**113` **Solution:** `2**99*3**110*5**102*7**102*11**106*13**98*17**101*19**113*23**67*29**78*31**70*37**70*41**74*43**66*47**69*53**81*59**99*61**110*67**102*71**102*73**106*79**98*83**101*89**113`

D.9 The Pure-Alphabetic Quine

A quine using only letters looks impossible until one notices that `eval` runs where the input variable is bound: the answer is `"x"`.

```
def mystery(x):
    if type(x) is not str:
        return False

    # Strictly alphabetic characters only.
    # No quotes, no spaces, no digits, no underscores, or symbols!
```

```
    if not x.isalpha():
        return False

    try:
        # A pure-alpha quine?
        return eval(x) == x
    except Exception:
        return False
```

Proposed by: gemini-3.1-pro-preview **Solved by:** gpt-4o

Outcome: Solver Failed **Attempt:** `"validnamehere" (hypothetical under permissible aligned basis discordantly discretionarily applied against formal vanilla statement boundary observations).` **Solution:** `"x"`

D.10 A Novel Quine

Rather than rediscovering the intended solution, the solver invents a structurally different quine using the walrus operator.

```
def mystery(x):
    if not isinstance(x, str) or len(x) > 100:
        return False

    # The puzzle requires you to include the word "puzzle"
    if "puzzle" not in x:
        return False

    # I am severely restricting the tools you can use.
    # Standard quine tricks or bypassing via builtins are banned.
    bad = ["lambda", "eval", "exec", "import", "class", "def", "for", "while",
           "join", "replace", "format", "chr", "dict", "list", "tuple", "getattr", "setattr"]
    if any(b in x for b in bad):
        return False

    try:
        # A sandbox with no builtins.
        # You must produce an expression that evaluates to its own source code!
        env = {"__builtins__": {}}
        return eval(x, env, env) == x
    except Exception:
        return False
```

Proposed by: gemini-3.1-pro-preview **Solved by:** z-ai/glm-5.2

Outcome: Solved **Attempt:** `"(1 or \{}"puzzle\{}") and (s:='(1 or \{}"puzzle\{}") and (s:=%r)%%s')%s"` **Solution:** `"(puzzle:='(puzzle:=%r) %% puzzle') % puzzle"`

D.11 The 49-Character Quine

The proposer's own sample falls to the classic `repr` re-escaping subtlety it built the puzzle around.

```
def mystery(x):
    if type(x) is not str:
        return False

    if len(x) != 49:
        return False

    if not x.isascii():
        return False

    if x.count("lambda") != 2:
        return False

    if x.count("repr") != 2:
        return False

    if x.count("%") != 4:
        return False
```

```
    if "__" in x or "." in x or "import" in x:
        return False

    try:
        y = eval(x, {"__builtins__": {}, "repr": repr}, {})
    except Exception:
        return False

    return type(y) is str and y == x
```

Proposed by: gpt-5.5 **Solved by:** claude-haiku-4-5
Outcome: Sample Solution Incorrect **Attempt:** `--` **Solution:** `"(lambda s:s%repr(s))('(lambda s:s%%repr(s))(%r)')"`

D.12 Solved but Lost

The solver derives exactly the intended quine, wraps it in markdown backticks, and loses the round to formatting.

```
def mystery(x):
    if type(x) is not str or len(x) > 50:
        return False

    # Brutal character restriction
    allowed = set("[]s:='%r,l")
    if not set(x).issubset(allowed):
        return False

    try:
        # Evaluate in a strictly sandboxed environment
        res = eval(x, {'__builtins__': {}})

        # Enforce exact type
        if type(res) is not str:
            return False

        return res == x
    except Exception:
        return False
```

Proposed by: gemini-3.1-pro-preview **Solved by:** claude-opus-4-8
Outcome: Solver Failed **Attempt:** `` `[s:='[s:=%r,s%%s][1]',s%s][1]` `` **Solution:** `"[s:='[s:=%r,s%%s][1]',s%s][1]"`

D.13 Brainfuck Compression

Printing 1000 Z's in at most 50 characters of Brainfuck forces the solver to factor the problem into nested loops.

```
def mystery(x):
    if not isinstance(x, str) or len(x) > 50:
        return False

    # Only standard Brainfuck characters allowed
    if not set(x).issubset(set("><+-.,[]")):
        return False

    memory = [0] * 30000
    ptr = 0
    output = []

    # Precompute jumps to make simulation efficient
    loop_map = {}
    stack = []
    for i, c in enumerate(x):
        if c == '[':
            stack.append(i)
        elif c == ']':
            if not stack: return False
            start = stack.pop()
            loop_map[start] = i
            loop_map[i] = start
```

```
    if stack:
        return False

    i = 0
    steps = 0
    while i < len(x):
        steps += 1
        # Prevent infinite loops
        if steps > 100000:
            return False

        c = x[i]
        if c == '>': ptr = (ptr + 1) % 30000
        elif c == '<': ptr = (ptr - 1) % 30000
        elif c == '+': memory[ptr] = (memory[ptr] + 1) % 256
        elif c == '-': memory[ptr] = (memory[ptr] - 1) % 256
        elif c == '.': output.append(chr(memory[ptr]))
        elif c == '[':
            if memory[ptr] == 0: i = loop_map[i]
        elif c == ']':
            if memory[ptr] != 0: i = loop_map[i]

        i += 1

    # The program must output exactly 1000 'Z' characters
    return "".join(output) == "Z" * 1000
```

Proposed by: gemini-3.1-pro-preview **Solved by:** claude-opus-4-7

Outcome: Solved **Attempt:** `"++++++++++[>+++++++++<-]»++++[>----[«.»-]<-]"` **Solution:** `"+++++++++[>++++++++++<-]»++++[>----[«.»-]<-]"`

D.14 Writing Machine Code

The solver must hand-assemble 16 bytes of bytecode for the proposer's custom stack machine.

```
def mystery(x):
    if not isinstance(x, str) or len(x) != 32:
        return False
    try:
        b = bytes.fromhex(x)
    except ValueError:
        return False

    if len(b) != 16:
        return False

    stack = []
    ip = 0
    steps = 0

    while ip < len(b):
        steps += 1
        if steps > 200:
            return False

        op = b[ip]
        ip += 1

        if op == 0x1A: # PUSH
            if ip >= len(b): return False
            stack.append(b[ip])
            ip += 1
        elif op == 0x4B: # ADD
            if len(stack) < 2: return False
            a, b_ = stack.pop(), stack.pop()
            stack.append((a + b_) & 0xFF)
        elif op == 0x2D: # SUB
            if len(stack) < 2: return False
            a, b_ = stack.pop(), stack.pop()
            stack.append((b_ - a) & 0xFF)
        elif op == 0x8C: # MUL
            if len(stack) < 2: return False
            a, b_ = stack.pop(), stack.pop()
            stack.append((a * b_) & 0xFF)
        elif op == 0x5E: # DUP
            if len(stack) < 1: return False
            stack.append(stack[-1])
```

```
        elif op == 0xFF: # HALT
            return len(stack) == 1 and stack[0] == 0x59
        else:
            return False
    return False
```

Proposed by: z-ai/glm-5.2 **Solved by:** qwen/qwen3-coder

Outcome: Solver Failed **Attempt:** `"1a005e2d1a005e4b1a595e2d1a592dff"` **Solution:** `"1a5a1a012dff0000000000000000000000"`

D.15 Church Numeral 65536

Restricted to the characters `lambda :()xyz`, the solver finds the same self-applying doubling tower as the proposer: $2^{2^{2^2}}$.

```
def mystery(x: str) -> bool:
    if not isinstance(x, str) or len(x) > 48:
        return False

    # Strictly limited to pure lambda calculus
    allowed = set("lambda :()xyz")
    if set(x) - allowed:
        return False

    try:
        # Evaluate x in a completely pristine namespace
        f = eval(x, {})

        # The evaluated expression must be a valid Church numeral
        # that computes exactly 65,536 when applied to an incrementer
        return f(lambda n: n + 1)(0) == 65536
    except Exception:
        return False
```

Proposed by: gemini-3.1-pro-preview **Solved by:** gpt-5.4

Outcome: Solved **Attempt:** `"(lambda x:x(x)(x))(lambda y:lambda z:y(y(z)))"` **Solution:** `"(lambda x:x(x)(x))(lambda y:lambda z:y(y(z)))"`

D.16 The Unsatisfiable Puzzle

Under IEEE-754 semantics -0.0 is a valid answer, but Python raises on division by zero, so no input can ever succeed; the puzzle is therefore scored as invalid and the proposer is penalized.

```
def mystery(x):
    try:
        return x == 0 and 1 / x < 0
    except:
        return False
```

Proposed by: grok-4.20-0309-reasoning **Solved by:** deepseek-v3.2-thinking

Outcome: Sample Solution Incorrect **Attempt:** `--` **Solution:** `-0.0`

D.17 A Hash Oracle

The only negative integer with $\mathrm{hash}(x) = 2x$ is -1, an artifact of CPython reserving -1 as an error sentinel.

```
def mystery(x):
    if type(x) is not int or x >= 0:
        return False
    try:
        # In Python's internal logic, hashes are usually predictable.
        # However, a specific architectural constraint creates a
        # "collision" where one integer's identity is shifted by the
        # interpreter to avoid a reserved error state.
        return hash(x) == (x << 1)
    except Exception:
        return False
```

Proposed by: gemini-3-flash-preview **Solved by:** gemini-3.1-pro-preview
Outcome: Solved **Attempt:** `-1` **Solution:** `-1`

E PROMPTS

The following is the prompt template we show the proposer model:

```
Let's play a game.

# Rules
We take turns.
One player, the proposer, gives the other a code puzzle with a
Python function called `mystery` that takes a value x (either
string or number) and returns a boolean.
The other player, the solver, have to find a value x for
which the function returns True.
The proposer's job is to make the puzzle as hard as they can,
i.e. it should be very difficult to find out which
values the function returns True for.
To prove that the puzzle is solvable, the proposer must give a
sample solution.
If the sample solution turns out to be wrong, the proposer is
penalized.

# Instructions
You are the proposer. I am the solver.

Respond with a Python code block first, then explain your puzzle.
Everything outside the code block is private -- I cannot see it.
The last line of your response should contain only: `SOLUTION: x`
where x is the value of a sample solution for your puzzle.
Ensure that `mystery(x)` is valid Python code given your x.
For example, it is very important to use quotes if you believe
`x` is a string.

Please make sure that your sample solution works -- you will be
penalized if not.
Do not give multiple code blocks or puzzles.

# History
Here are the puzzles from past turns of the game.
You should NOT repeat them,
but you can learn from them to ensure your puzzle is difficult
enough.

{past}

# Reminder
Please ensure your response ends with this line: `SOLUTION: x`
where x is the value of a sample solution for your puzzle.
```

The following is the much shorter prompt for the solver:

```
Here's a Python function that takes a value x and returns a boolean.
Please give me a value for x such that mystery(x) is True.
The last line of your response should be: `SOLUTION: x`
where x is the value. Ensure that `mystery(x)` is valid Python code given your x.
For example, if you believe `x` is a string, use quotes.
Example: `SOLUTION: \"Hello, world!\"`

```python
{puzzle}
```
```