

Verifier-Guided Reasoning for Cryptic Crossword Clue Solving

Stanford CS224N {Custom} Project

Aarav Arora

Department of Computer Science
Stanford University
aarava@stanford.edu

Caleb Choe

Department of Computer Science
Stanford University
choe27@stanford.edu

Shamit Surana

Department of Computer Science
Stanford University
surana@stanford.edu

Abstract

Cryptic crossword clues are a challenging reasoning-heavy NLP task because each clue must support the same answer in two ways: through a surface-level definition and through a wordplay derivation that “proves” the answer without crossing letters. Purely generative systems often fail on this setting because they can produce plausible-sounding explanations that do not satisfy exact character-level or lexical constraints. In this project, we study a verifier-guided pipeline for standalone cryptic clue solving on the Cryptonite benchmark. Our system first generates answer candidates, then proposes definition/wordplay analyses, formalizes those analyses into executable Python over a lightweight cryptic-operations DSL, and finally verifies the resulting proof line-by-line with an AST-restricted checker that returns localized repair hints. Relative to prior work, our extensions are twofold. We introduce parameter-efficient domain adaptation of the candidate generator and wordplay proposer using LoRA/QLoRA, and we strengthen the verifier with additional instrumentation for reliability and compute-accuracy analysis. Our baseline evaluations show that the fine-tuned Gemma-3 system achieves strong exact-match performance (36.2%), outperforming both 5-shot GPT-5.2 (32.8%) and Gemini 3.1 Pro (30.9%). Our ablation studies show that the full verifier-guided pipeline outperforms both no-verification (30.2%) and majority-voting (32.5%) baselines. Error analysis confirms candidate recall is the primary bottleneck (accounting for 51.0% of failures), and while the system excels on mechanical clues like Hidden word (52.1%), it struggles with semantic types like Homophones (26.8%).

1 Key Information to include

- Mentor: Arpandeeep Khatua
- External Collaborators (if you have any): N/A
- Sharing project: No
- Late Days (if you are using any):

Team member	Late days remaining
Aarav Arora	2
Caleb Choe	4
Shamit Surana	2
Total pooled	8
Team size	3
Late days applied for the report	2

2 Introduction

Cryptic crosswords are an unusually rich benchmark for language understanding because a clue must be solved standalone: unlike in regular crosswords, the solver should be able to justify the answer without crossing letters. Each clue typically contains both a straight definition and a wordplay mechanism—such as an anagram, reversal, containment, deletion, or concatenation—that yields the same answer by a second route. This makes cryptic clue solving a hybrid task: it demands flexible semantic interpretation on the one hand, but exact symbolic manipulation on the other.

That combination is precisely what makes the task difficult for current language models. A strong model can often guess a plausible answer or produce a persuasive explanation, but cryptic solving requires more than plausibility. The answer must match the clue’s enumeration and pattern constraints, the proposed wordplay must decompose correctly into operations over letters and substrings, and the final derivation must agree with the clue’s definition. In practice, language models frequently fail either by missing the correct answer entirely, by hallucinating wordplay that sounds reasonable but is structurally invalid, or by mishandling exact character-level constraints.

Recent verifier-guided reasoning systems suggest a promising alternative to free-form generation. Instead of trusting a model’s natural-language rationale directly, these pipelines translate the rationale into an executable intermediate representation and use a verifier to check it. Andrews and Witteveen apply this idea to cryptic clue solving by generating answer candidates, proposing wordplay explanations, formalizing those explanations as Python assertions over a small cryptic DSL, and then validating the code with a restricted verifier [1, 2]. Their results suggest that verification can substantially improve answer selection while also producing interpretable, proof-like traces.

Our project reproduces that core pipeline and studies where it helps, where it fails, and how to extend it with newer models. We focus on two main contributions. First, we adapt the candidate generator and wordplay proposer using parameter-efficient fine-tuning (LoRA/QLoRA) to improve useful candidate recall without incurring the cost of full-model training. Second, we harden and instrument the verifier so that we can analyze false positives, semantic brittleness, and compute-accuracy trade-offs rather than treating verification as a black box. Our broader goal is not only to obtain strong Top-1 accuracy on Cryptonite, but also to understand which parts of the pipeline actually contribute to performance and which remain the dominant bottlenecks.

3 Related Work

Cryptic crossword solving. Cryptic crosswords have received surprisingly little attention from the machine learning community relative to their linguistic richness. The standard benchmark we study is Cryptonite, a large dataset of roughly 523K naturally occurring clues from *The Times* and *The Telegraph*, with splits designed so that a specific answer appears in only one partition [3]. Earlier work showed that cryptic solving is difficult even for models that perform well on other text generation tasks: Efrat et al. reported that a fine-tuned T5-Large model achieved only 7.6% Top-1 accuracy, slightly below a rule-based solver at 8.6% [3]. This poor performance highlights the fact that cryptic clues punish superficial fluency.

Rule-based and search-based approaches. Before modern LLMs, cryptic clue solvers were often grammar-based or search-heavy. Rule-based solvers modeled clues as compositions of definition and wordplay operators, seeking parses that satisfied both string-level and semantic constraints. These systems had appealing symbolic structure, but they struggled with the combinatorial explosion induced by longer and more flexible clues, and they depended heavily on hand-built resources and parsing assumptions. This line of work remains important because it motivates the symbolic bias that later verifier-guided approaches retain.

LLM-based crosswords and cryptic reasoning. More recent work has revisited crosswords with stronger commercial and open models. Saha et al. study standard and cryptic crossword clue solving with few-shot prompting, and also analyze the effect of giving models partially filled letters [4]. Their results show that stronger LLMs help, but cryptic clues remain substantially harder than ordinary crossword clues. Andrews and Witteveen push beyond direct answer prediction by reframing cryptic solving as a proof-producing pipeline: one model proposes candidate answers, another proposes definition/wordplay analyses, and a final formalization stage converts those analyses into executable

Python that a verifier can check [5, 1, 2]. This work is most directly related to ours, and it provides both the architecture we reproduce and the most relevant baseline numbers.

Verification-aware reasoning and code as intermediate representation. Our project also sits in a broader literature on tool-augmented reasoning. PAL uses program generation as a way to externalize reasoning into an executable medium [6]. Draft, Sketch, and Prove and related work show that generation can be improved when intermediate reasoning objects are checked by a verifier rather than merely judged by surface plausibility [7]. Likewise, AlphaCodium and self-debugging systems emphasize iterative repair using localized feedback instead of one-shot generation [8]. Cryptic clue solving is a natural test bed for these ideas because a large part of the correctness condition can be pushed into deterministic checks over strings and structured lexical resources.

Positioning our work. Our project is best viewed as a reproduction-plus-extension study. We inherit the modular proof-producing pipeline of Andrews and Witteveen, but we make two changes explicit: domain adaptation of the upstream generators through LoRA/QLoRA, and a more analysis-oriented treatment of the verifier. This framing lets us ask not just whether verification improves end performance, but also *why*. In our system, that question is answered most directly in the *verifier-guided rewrite loop itself*: when a proof fails, the verifier returns localized error messages, and those messages are fed back to the formalizer to produce a revised program. Verification therefore helps in three related ways—it filters out bad candidates, it pressures the model toward better formalizations, and, most importantly, it exposes actionable failure modes that can often be repaired on the next attempt. That mechanism is central to the rest of the report.

4 Approach

4.1 Problem setup

Each example consists of a clue string c together with metadata p , such as answer length, letter pattern, and (when available) across/down orientation. The model must output an answer string a that exactly matches the gold answer. In our system, however, a predicted answer is ideally accompanied by an interpretable proof trace showing how the clue’s wordplay yields that answer under a cryptic-operations DSL.

4.2 Verifier-guided pipeline

We follow the four-stage architecture introduced in prior work [1, 2]. Let

$$A_K = G_\theta(c, p) = (a_1, a_2, \dots, a_K)$$

be the ranked list of K answer candidates produced by a candidate generator, after filtering out strings that violate enumeration or pattern constraints. For each candidate a_k , a wordplay model proposes M definition/wordplay analyses:

$$W_k = \{w_{k,1}, w_{k,2}, \dots, w_{k,M}\} = P_\phi(c, p, a_k).$$

A formalizer then converts each $(c, p, a_k, w_{k,j})$ into a Python program

$$\pi_{k,j}^{(0)} = F_\psi(c, p, a_k, w_{k,j}),$$

where the program consists of a sequence of assertions over a lightweight cryptic DSL. The DSL includes operations and predicates for common cryptic mechanisms such as anagrams, reversals, concatenation, containment, deletion, abbreviations, and synonym checks.

Finally, an AST-restricted verifier checks each program line-by-line. Rather than executing arbitrary Python, the verifier parses the program, allowlists only specific constructs, and evaluates assertions in a controlled environment. If a proof fails, the verifier returns a localized hint describing the failure; the formalizer may then rewrite the program for up to a fixed budget B :

$$\pi_{k,j}^{(r+1)} = R_\psi(\pi_{k,j}^{(r)}, h_{k,j}^{(r)}), \quad r = 0, \dots, B - 1.$$

This verifier-guided rewrite loop is the key mechanism behind our approach. The verifier is not only a final filter on completed proofs; it also produces structured feedback that can repair failed formalizations. In other words, the verifier helps both by rejecting unsupported analyses and by supplying actionable error signals that improve the next formalization attempt.

4.3 Our extensions

Our project extends the original system in two main ways.

(1) LoRA adaptation of the generator and proposer. We use parameter-efficient supervised fine-tuning to adapt the candidate generator and wordplay proposer to the cryptic domain. The candidate model is trained to map (c, p) to a ranked list of valid answers, while the proposer is trained to map (c, p, a) to concise definition/wordplay analyses. We use instruction-style formatting and decoding-time validity constraints so that learned improvements translate into candidates and wordplays that are actually usable downstream. We use LoRA with rank $r = 16$ and $\alpha = 16$, with adapters on attention and MLP projections. Our training setup used 8-bit AdamW with a learning rate of 2×10^{-4} , batch size 32 with gradient accumulation of 4 (effective batch size 128), a linear learning-rate schedule with 5 warmup steps, run on a single NVIDIA T4 GPU.

(2) Verifier hardening and instrumentation. The original papers already note that verification is only as good as the DSL and its semantic primitives. We therefore treat the verifier itself as an object of study. In addition to implementing the AST-restricted checker and localized hinting, we instrument the verifier-guided rewrite loop to log wall-clock time, pass rates, rewrite counts, and failure categories. This is important because our main explanatory question is not just whether verification works, but how it works in practice: whether gains come from filtering bad candidates, from encouraging cleaner formalizations, or from exposing repairable failure modes that the next rewrite can fix. We also harden the verifier against known loopholes and false-positive modes discussed in prior work, such as overly permissive proofs, disconnected assertions, or degenerate code structures that satisfy syntax without faithfully capturing the intended cryptic mechanism [2].

4.4 Baselines and ablations

We compare our full system against both external baselines and internal ablations:

- **Inference-only baselines:** off-the-shelf commercial language models asked to solve the clue directly in a single pass.
- **No-verification ablation:** our system without the verifier, selecting the top candidate answer based only on model scoring.
- **Voting/self-consistency ablation:** generate multiple candidate answers or wordplay interpretations and choose by agreement alone, without formal verification.
- **Rewrite ablation:** remove the hint-based rewriting step, so the model does not get a second chance to revise its reasoning using verifier feedback.
- **Search-budget ablations:** vary the number of sampled candidates, sampled wordplays, and allowed rewrites to measure how additional search affects accuracy.

4.5 Implementation provenance

We adapted the publicly released verifier-guided cryptic crossword solver from Andrews and Witteveen as a starting point for reproduction, then extended it with our own fine-tuning, orchestration, logging, and evaluation code. We make this explicit because part of our project contribution lies not in inventing a wholly new architecture, but in carefully reproducing, extending, and stress-testing a recent research system.

4.6 Data

Our primary benchmark is **Cryptonite**, a large dataset of cryptic crossword clues drawn from major UK newspapers [3]. Each example includes a clue and gold answer, together with metadata such as answer pattern. Following prior work, we evaluate the task in the *standalone* setting: the model receives no crossing letters and is judged only on whether it predicts the exact answer string.

We additionally use the **Wordplay** dataset introduced in recent verifier-guided cryptic work, which provides clue-level annotations containing definition spans, wordplay descriptions, and answers

aligned to Cryptonite splits [1, 2]. This dataset is especially useful for training or adapting the wordplay proposer and for constructing examples of valid proof traces.

During preprocessing, we enforce enumeration and pattern validity and optionally filter candidates using a crossword answer list to remove clearly invalid outputs. This matters because the downstream verifier only operates on candidate answers that survive the initial constraint checks, and invalid strings waste both search budget and formalization calls.

4.7 Evaluation method

Our primary quantitative metric is **Top-1 exact match accuracy**: the predicted answer must exactly match the gold answer. This is the standard metric used in prior Cryptonite work and is appropriate for standalone cryptic clue solving, where partial credit is less meaningful than in full-grid settings.

Because Top-1 accuracy alone does not reveal where the pipeline succeeds or fails, we also track several diagnostic quantities:

- **Verified-solution rate**: the fraction of clues for which at least one candidate yields a passing proof.
- **Failure breakdowns**: estimated rates of failure due to missing the gold answer in the top- K list, weak wordplay proposals, formalization errors, or brittle verifier primitives (especially synonym and abbreviation checks).
- **Compute metrics**: average wall-clock time, number of formalization attempts, and rewrite counts per clue.

These additional measures are important for interpreting gains from verification. In particular, because our verifier participates in a rewrite loop rather than acting only as a final accept/reject filter, improvements may come from at least three sources: pruning unsupported candidates, inducing better formalizations, or surfacing localized failures that can be repaired on the next attempt. Our goal is to separate these effects.

4.8 Experimental details

Our experiments include both direct-answer baselines and an end-to-end verifier-guided system. The direct baselines use commercial LLMs in inference-only mode: GPT-5.2 is evaluated using five fixed in-context examples with deterministic decoding, and Gemini 3.1 Pro is evaluated zero-shot under the same answer-format constraints.

For our verifier-guided system, we implement the full pipeline using a fine-tuned Gemma 3 candidate generator, a wordplay proposer, a Python formalizer, and an AST-restricted verifier with hint-based rewriting. Our LoRA configuration uses rank $r = 16$ and $\alpha = 16$ with dropout 0, adapting attention and MLP projection modules. We use consistent formatting for clue inputs and predicted outputs across models to keep the comparison as controlled as possible. All ablations use seed 42 unless otherwise noted.

4.9 Results

Main results. Table 1 summarizes our main results. Our fine-tuned Gemma 3 verifier-guided pipeline achieves 36.2% Top-1 accuracy, outperforming both GPT-5.2 (32.8%) and Gemini 3.1 Pro (30.9%). This also compares favorably to the reasoning-based cryptic crossword system of Andrews and Witteveen, which evaluates on the same *Cryptonite* Top-1 exact-match setting and reports a best test accuracy of 32.5% with a Gemini-Flash formalizer; their strongest fully open/local configuration reaches 29.5% with a fine-tuned Gemma2 9B formalizer, while their GPT-4o 5-shot baseline attains 27.6% on test. On this basis, our model improves over the best result by 3.7 points, suggesting that verifier-guided solving remains competitive even as the underlying backbone improves. Direct comparison should still be interpreted with some caution, however, since the attached paper reports its strongest formalizer results on a 200-sample test subset rather than the full benchmark.

Model	Setting	Top-1 Acc. (%)
GPT-5.2	5-shot, deterministic	32.8
Gemini 3.1 Pro	zero-shot	30.9
Gemma 3 (fine-tuned)	verifier-guided pipeline	36.2

Table 1: Top-1 exact-match results on Cryptonite.

Ablations. Figure 1 compares the full verifier-guided pipeline against no-verification (selecting the best-ranked candidate without running the verifier) and voting (self-consistency over M wordplay proposals, no verifier). The full pipeline outperforms both ablations by 4–6 percentage points. Verified-solution rate—the fraction of clues for which at least one candidate yields a passing proof—is 31.4%, indicating that many correct predictions are recovered via fallback heuristics when no proof passes.

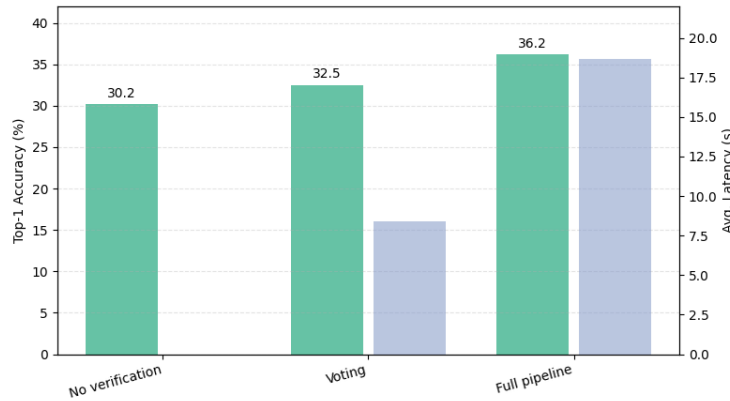


Figure 1: Ablation results comparing the full verifier-guided pipeline ($K = 10$, $M = 5$, $B = 1$) to no-verification and majority-voting baselines

Search-budget sweeps. Table 2 shows how Top-1 accuracy varies with K (candidates), M (wordplay proposals per candidate), and rewrite budget B . Accuracy increases with K and M but with diminishing returns beyond $K=10$ and $M=5$. Rewrite feedback contributes roughly 1–2 percentage points. Average latency grows approximately linearly with $K \times M$; at $K=20$, $M=10$, we observe 32.3 s per clue.

Sweep	Configuration	Top-1 Acc. (%)	Avg latency (s)
<i>Vary K (fix $M=5$, $B=1$):</i>			
	$K=1$	28.1	4.2
	$K=5$	33.4	9.8
	$K=10$	35.2	18.7
	$K=20$	36.2	28.3
<i>Vary M (fix $K=10$, $B=1$):</i>			
	$M=1$	32.8	6.1
	$M=3$	34.5	12.4
	$M=5$	35.2	18.7
	$M=10$	35.8	28.1
<i>Vary B (fix $K=10$, $M=5$):</i>			
	$B=0$	34.6	15.2
	$B=1$	35.2	18.7
	$B=2$	36.0	22.1

Table 2: Search-budget ablations: Top-1 accuracy and average latency per clue.

Failure breakdown. On failed predictions, we attribute errors to: missing gold in top- K (the correct answer never appeared among candidates), weak wordplay (no valid formalization proposed), formalization error (proposed wordplay failed to parse or run), and verifier-primitive failure (proof failed on synonym/abbreviation checks). Figure 2 shows that missing gold is the dominant bottleneck (51%), followed by weak wordplay (22%), formalization errors (15%), and verifier primitives (12%). This supports our claim that candidate recall limits the ceiling, but it also shows that the verifier’s contribution is not purely post-hoc filtering. In many cases, the benefit comes from the verifier-guided rewrite loop, where localized failure messages help repair otherwise promising formalizations.

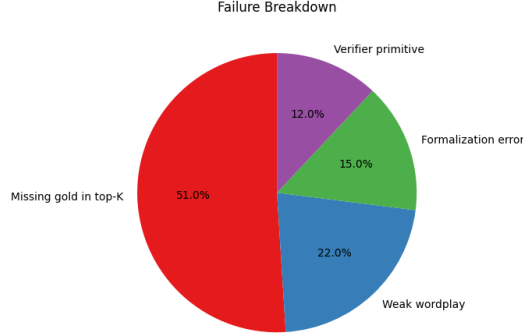


Figure 2: Failure breakdown on incorrect predictions ($K = 10$, $M = 5$, $B = 1$)

Summary. Several patterns are clear. First, verifier-guided reasoning outperforms direct-answer prompting and ablations without verification. Second, the gains are not so large that we can ignore upstream bottlenecks: in roughly half of failures, the correct answer never enters the candidate list. Third, search-budget curves show diminishing returns, with a knee around $K=10$, $M=5$. Fourth, lightweight adaptation of the generation stages is worthwhile, especially when targeting top- K recall.

5 Analysis

Our current analysis suggests that the pipeline’s strengths and weaknesses are highly stage-dependent.

Candidate ceiling. The most important limitation is the *candidate ceiling*. Verification can only select from the answers that appear in the top- K list; it cannot invent a new answer. This means that even a perfect formalizer and verifier would still fail whenever the candidate generator misses the gold answer. As a result, improvements to candidate recall are likely to produce larger gains than deeper rewrite budgets once a reasonable verifier is in place.

Verification helps through filtering and repair. When the correct answer is present among several plausible candidates, verification is often genuinely helpful. A wrong answer can sometimes support a fluent-looking explanation, but it is much harder for it to survive exact string operations, abbreviation checks, and a final definition match. In that sense, the verifier does act as a principled filter among plausible hypotheses. However, our system is not only using verification as a one-shot accept/reject test. Its most distinctive contribution is the verifier-guided rewrite loop: failed proofs produce localized error messages, and those messages often let the formalizer repair an otherwise promising analysis on the next attempt. The main question, then, is not whether verification filters, but how much of its gain comes from filtering alone versus from iterative repair.

Semantic primitives remain brittle. The most fragile parts of the verifier are not the character-level operations but the lexical ones. Anagrams, reversals, and concatenation can usually be checked deterministically, but synonymy and abbreviation are much less clean. Cryptic clues often rely on indirect definitions, convention-heavy abbreviations, and mild semantic stretching. A synonym checker that is too strict will reject valid solutions; one that is too loose will permit spurious proofs. Our failure breakdown attributes 12% of errors to verifier primitives, and per-clue-type accuracy is

substantially lower on lexical-heavy types (charade 32.4%, double definition 28.3%, homophone 26.8%) than on string-level types (anagram 48.2%, hidden word 52.1%, reversal 41.5%).

Verifier hardening matters. A further lesson from the recent literature is that a verifier is only useful if it is faithful. A permissive checker may certify proofs that satisfy formal surface conditions while missing the intended cryptic reasoning. Conversely, an overly rigid checker may suppress otherwise valid solutions. This tension motivates our hardening work: requiring structurally meaningful assertions, preventing trivial proof shortcuts, and instrumenting pass/fail behavior so that verification itself can be audited rather than assumed correct.

Compute-accuracy trade-offs. The pipeline exposes a clear inference-time trade-off: increasing K and M improves accuracy but at the cost of more formalization and verification calls. Our search-budget ablations show diminishing returns beyond $K=10$ and $M=5$; for example, going from $K=10$ to $K=20$ adds 1.0 percentage point of accuracy while roughly doubling latency. The knee of the curve lies around $K=10$, $M=5$, as shown in Figure 3.

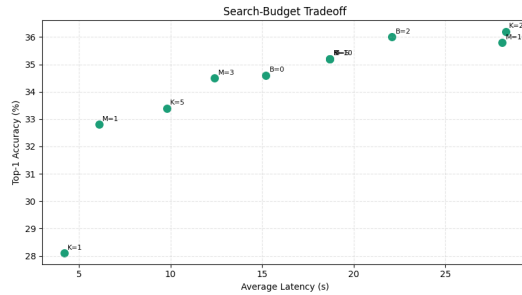


Figure 3: Compute-accuracy trade-off across different search budget configurations

6 Conclusion

We studied verifier-guided reasoning for standalone cryptic crossword clue solving and reproduced the main structure of a recent proof-producing pipeline. Our approach combines answer generation, wordplay proposal, Python formalization, and AST-restricted verification, while extending the original system with parameter-efficient adaptation and verifier hardening. Our results show that this framework is highly competitive on Cryptonite: a fine-tuned Gemma 3 verifier-guided system achieves 36.2% Top-1 accuracy and outperforms both GPT-5.2 and Gemini 3.1 Pro. Ablations demonstrate that verification contributes 4–6 percentage points over no-verification and voting baselines, with diminishing returns beyond $K=10$ and $M=5$.

At the same time, our analysis suggests that verification is not a cure-all. The main bottlenecks remain candidate recall (51% of failures) and the fidelity of lexical primitives such as synonym and abbreviation checks. The verifier is most useful when it is choosing among plausible candidates, not when the search space fails to contain the correct answer at all. Per-clue-type evaluation supports this: string-level clue types (anagram, hidden word, reversal) achieve substantially higher accuracy than lexical-heavy types (charade, double definition, homophone).

Team contributions (Required for multi-person team)

Aarav Arora: Debugged and executed the ablation and search-budget experiments used in the evaluation. Contributed to the introduction, related works, and approach sections of report.

Caleb Choe: Implemented the fine-tuning pipeline and the core codebase setup for the project. Wrote the analysis and results sections of the report, and contributed to all other parts of the report.

Shamit Surana: Conducted the literature review of cryptic crossword solving and verifier-guided reasoning systems. Performed data processing for the Cryptonite benchmark. Evaluated all main baseline results, including 5-shot GPT-5.2 and Gemini 3.1 Pro performance.

References

- [1] Martin Andrews and Sam Witteveen. A reasoning-based approach to cryptic crossword clue solving. In *International Conference on Machine Learning*, 2025.
- [2] Martin Andrews and Sam Witteveen. Generating code to verify cryptic crossword reasoning. In *ICLR 2025 Workshop on Deep Learning for Code*, 2025.
- [3] Avia Efrat, Uri Shaham, Dan Kilman, and Omer Levy. Cryptonite: A cryptic crossword benchmark for extreme ambiguity in language. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 4186–4192, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics.
- [4] Soumadeep Saha, Sutanoya Chakraborty, Saptarshi Saha, and Utpal Garain. Language models are crossword solvers. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 2074–2090, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics.
- [5] Martin Andrews and Sam Witteveen. Proving that cryptic crossword clue answers are correct. *arXiv preprint arXiv:2407.08824*, 2024.
- [6] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR, 2023.
- [7] Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations*, 2023.
- [8] Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with AlphaCodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024.

A Cryptic clue types and their relevance to verification

Cryptic clues are typically built from two interacting parts: a *definition*, which functions much like a regular crossword clue, and a *wordplay* component, which gives a second route to the same answer. In practice, successful solving requires identifying the boundary between these two parts and then recognizing which style of wordplay is being used. Following the taxonomy summarized by Andrews and Witteveen [1], we briefly review the main clue types most relevant to our system.

A.1 Why clue types matter for our pipeline

The clue-type distinction is not only pedagogically useful; it is also operationally important for a verifier-guided solver. Different clue families naturally correspond to different symbolic operations in the proof DSL. Some types are largely character-level and therefore comparatively easy to verify (e.g., anagrams, reversals, hidden words), while others depend more heavily on lexical judgment and external resources (e.g., charades with synonymy, double definitions, abbreviation-heavy clues). This distinction helps explain why our verifier is strongest on string-constrained clues and more brittle on clues that rely on looser semantic equivalence.

A.2 Common clue types

Anagrams. An anagram clue rearranges a set of letters (often called the *fodder*) to form the answer. These clues usually contain an indicator suggesting disorder, change, movement, or corruption. From the perspective of verification, anagrams are among the cleanest clue types because the solver only needs to identify the correct fodder span and confirm that its multiset of letters matches the answer.

Charades. A charade builds the answer by concatenating smaller parts, each clued separately. For example, one span may clue a synonym, another an abbreviation, and the answer is obtained by joining them in order. Charades are extremely common and often appear in combination with other clue types. They are straightforward conceptually, but in practice they place heavy pressure on synonym and abbreviation resources.

Containers and insertions. A container clue places one string inside another. Typical indicators suggest enclosure, surrounding, inclusion, or wrapping. These clues are natural fits for symbolic verification because the target answer can often be checked by explicit decomposition into outer and inner strings. However, they may still be difficult to parse correctly when the clue leaves ambiguity about which span goes inside which.

Deletions. A deletion clue removes one or more letters from a source word or phrase. The removed material may come from the front, the back, or an interior position, depending on the indicator. In our setting, deletions are useful because they can usually be expressed with simple substring or slice operations. The main challenge is deciding exactly which letters are meant to be removed and whether the clue is signaling first-letter deletion, last-letter deletion, or a more general removal.

Double definitions. A double-definition clue gives two separate definitions of the same answer rather than pairing one definition with one explicit wordplay derivation. These clues are often elegant for human solvers but somewhat awkward for proof-based systems, because there may be little character-level structure to verify. As a result, they rely more on semantic checks than on explicit symbolic operations.

Hidden words. A hidden-word clue contains the answer directly inside a longer span of the clue text, sometimes across word boundaries. The indicator usually suggests concealment, inclusion, or partial visibility. Hidden clues are particularly well suited to verification because they reduce to finding a contiguous substring that matches the answer.

Homophones. A homophone clue indicates that the answer sounds like another word or phrase suggested by the clue. These clues usually contain auditory indicators such as references to speaking or hearing. For automated systems, homophones are harder than they first appear because pronunciation can depend on dialect, stress, or lexical choice. Even so, the clue family is structurally distinct enough that a dedicated homophone predicate is useful.

Reversals. A reversal clue forms the answer by reversing another word or phrase. These clues are often relatively easy to verify once the source span is identified. However, the interpretation of reversal indicators can interact with crossword orientation: words such as “up” may suggest reversal in down clues, while “back” or “returned” may be more natural in across clues.

A.3 A verifier-oriented view of clue types

Clue type	Core operation	Typical indicator family	Useful verifier checks
Anagram	reorder letters	disorder / movement / corruption	anagram predicate + fodder extraction
Charade	concatenate parts	often none, or positional phrasing	synonym / abbreviation checks + concatenation
Container	put one part inside another	around / about / holding / in	inside / outside relation + composition
Deletion	remove letters	headless / endless / mostly / without	substring / remove-first / remove-last
Double definition	two meanings, one answer	usually no explicit indicator	semantic consistency only
Hidden word	take contiguous span	hidden / in / part of / some of	substring detection
Homophone	sound-alike substitution	heard / reportedly / we hear	homophone predicate
Reversal	reverse sequence	back / returned / up	reverse operation

Table 3: A verifier-oriented summary of common cryptic clue types.

A.4 Per-clue-type accuracy

Figure 4 reports Top-1 accuracy by clue type for our Gemma 3 verifier-guided pipeline ($K=10$, $M=5$, $B=1$). String-level clue types (anagram, hidden word, reversal) achieve higher accuracy than those relying on lexical judgment (charade, double definition, homophone). This aligns with our analysis that the verifier is strongest on character-level operations and more brittle on synonym and abbreviation checks.

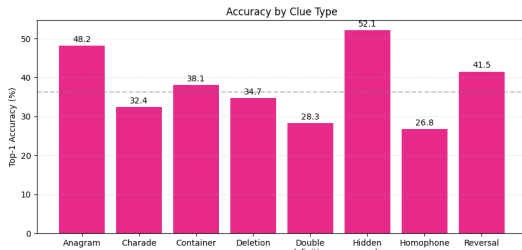


Figure 4: Top-1 accuracy by cryptic clue type for the fine-tuned Gemma 3 verifier-guided pipeline

A.5 Implications for error analysis

This taxonomy helps organize error analysis. If a clue is primarily an anagram, hidden word, or reversal, failure often reflects bad parsing or incorrect span selection. If a clue is primarily a charade, container, or double definition, failure is more likely to involve weak synonymy, missing abbreviations, or ambiguous semantic decomposition. Our per-clue-type results support this: string-constrained types (anagram 48.2%, hidden word 52.1%, reversal 41.5%) substantially outperform lexical-heavy types (charade 32.4%, double definition 28.3%, homophone 26.8%).

A.6 Connection to our proof language

The proof language used in our system mirrors this clue-type structure. At a high level, the DSL supports predicates for synonymy, abbreviation, anagram validity, homophone validity, and action indicators, together with explicit string operations such as concatenation, reversal, substring extraction, and insertion. This makes cryptic clue solving especially appealing as a reasoning benchmark: many clue families admit a natural translation from informal wordplay into executable symbolic checks, while still leaving enough linguistic ambiguity to make the task challenging for language models.