

# CoIn: Counting the Invisible Reasoning Tokens in Commercial Opaque LLM APIs

Guoheng Sun<sup>1\*</sup>, Ziyao Wang<sup>1</sup>, Bowei Tian<sup>1</sup>,  
Meng Liu<sup>1</sup>, Zheyu Shen<sup>1</sup>, Shwai He<sup>1</sup>, Yexiao He<sup>1</sup>,  
Wanghao Ye<sup>1</sup>, Yiting Wang<sup>1</sup>, Ang Li<sup>1†</sup>

<sup>1</sup>University of Maryland, College Park

## Abstract

As post-training techniques evolve, large language models (LLMs) are increasingly augmented with structured multi-step reasoning abilities, often optimized through reinforcement learning. These reasoning-enhanced models outperform standard LLMs on complex tasks and now underpin many commercial LLM APIs. However, to protect proprietary behavior and reduce verbosity, providers typically conceal the reasoning traces while returning only the final answer. This opacity introduces a critical transparency gap: users are billed for invisible reasoning tokens, which often account for the majority of the cost, yet have no means to verify their authenticity. This opens the door to *token count inflation*, where providers may overreport token usage or inject synthetic, low-effort tokens to inflate charges. To address this issue, we propose CoIn, a verification framework that audits both the *quantity* and *semantic validity* of hidden tokens. CoIn constructs a verifiable hash tree from token embedding fingerprints to check token counts, and uses embedding-based relevance matching to detect fabricated reasoning content. Experiments demonstrate that CoIn, when deployed as a trusted third-party auditor, can effectively detect token count inflation with a success rate reaching up to 94.7%, showing the strong ability to restore billing transparency in opaque LLM services. The dataset and code are available at <https://github.com/CASE-Lab-UMD/LLM-Auditing-CoIn>.

## 1 Introduction

Large language models (LLMs) have achieved significant advances in recent years. Yet, as pre-training begins to saturate available data resources Zoph et al. [2020], the research community has increasingly turned to inference-time innovations Hu et al. [2023], Kumar et al. [2025]. Among these, reinforcement learning (RL)-optimized reasoning models have shown promise by generating longer, structured reasoning traces that improve performance, particularly in tasks involving mathematics and code Guo et al. [2025], Muennighoff et al. [2025]. Such models, exemplified by DeepSeek-R1 Guo et al. [2025] and ChatGPT-O1 Jaech et al. [2024], demonstrate that scaling at inference time can yield new capabilities without further pretraining.

With this shift, providers like OpenAI increasingly adopt new service models. Reasoning traces, while critical for quality, are often verbose, sometimes speculative Jin et al. [2024], Zhang et al. [2025], and may reveal internal behaviors vulnerable to distillation Gou et al. [2021], Sreenivas et al. [2024]. To protect proprietary methods and streamline outputs, commercial APIs typically suppress these intermediate steps, exposing only the final answer. However, users are still charged for all generated tokens, including those hidden from view. We refer to such services as **Commercial Opaque LLM APIs (COLA)**—proprietary, pay-per-token APIs that conceal both intermediate text and logits.

\*ghsun@umd.edu

†angliece@umd.edu

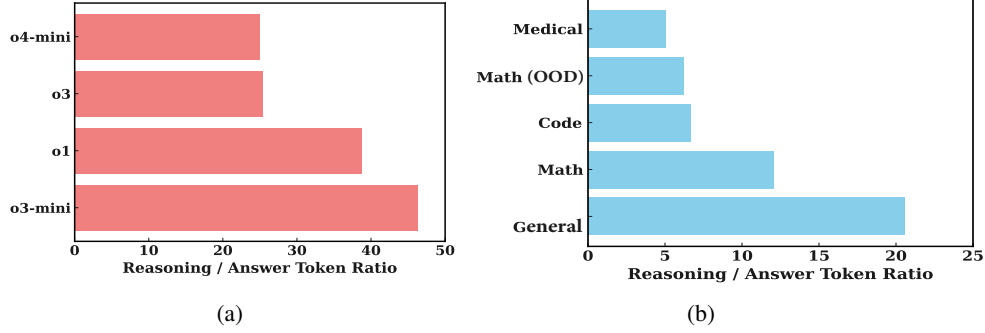


Figure 1: Ratio of reasoning tokens to answer tokens across datasets and deployed APIs. (a) Token ratios on the OpenR1-Math dataset across different OpenAI reasoning models. (b) Token ratios of the DeepSeek-R1 DeepSeek-AI [2025] across various reasoning datasets. In both cases, the number of reasoning tokens often exceeds answer tokens by an order of magnitude or more.

This design introduces a critical vulnerability: users have no means to verify token usage or detect overbilling. Because reasoning tokens often outnumber answer tokens by more than an order of magnitude (Figure 1), this invisibility allows providers to **misreport token counts** or **inject low-cost, fabricated reasoning tokens to artificially inflate token counts**. We refer to this practice as **token count inflation**. For instance, a single high-efficiency ARC-AGI run by OpenAI’s o3 model consumed 111 million tokens, costing \$66,772.<sup>3</sup> Given this scale, even small manipulations can lead to substantial financial impact. Such information asymmetry allows AI companies to significantly overcharge users, thereby undermining their interests.

To tackle this problem, we design **CoIn (Counting the Invisible)**, a verification framework that enables third-party auditing of invisible reasoning tokens in COLA services. CoIn ensures billing accountability by enabling users to validate token counts reported by the commercial provider, while preserving the confidentiality of hidden content and maintaining protection against distillation.

CoIn consists of two key components: **(1) Token Quantity Verification**, which leverages a verifiable hash tree Merkle [1987] to store fingerprint embeddings of reasoning tokens. Upon an audit request, CoIn allow users to query a small subset of the token fingerprints in the hash tree to verify the number of invisible tokens, avoiding accessing the actual reasoning tokens; and **(2) Semantic Validity Verification**, which detects fabricated, irrelevant, or low-effort token injection via a semantic relevance matching head. This matching head takes the embeddings of both the reasoning tokens and the answer tokens as input, and outputs a relevance score indicating their semantic consistency. Users can assess this score to identify token count inflation with low-effort token injection. Together, these components enable CoIn to identify misreported token counts and fabricated reasoning traces, enabling transparent billing without exposing proprietary data. In practice, CoIn can be deployed as a trusted third-party auditing service that ensures billing transparency while preserving the integrity and confidentiality requirements of COLA providers.

Our main contributions are as follows:

- We define the COLA architecture and formalize the emerging threat of *token count inflation*, categorizing both naive and adaptive inflation strategies.
- We design CoIn, a verification framework combining *token quantity verification* via verifiable hashing and *semantic validity verification* via embedding relevance, to audit invisible reasoning tokens without exposing proprietary content.
- Our experiments demonstrate that CoIn can achieve a 94.7% detection success rate against various adaptive attacks with less than 40% embedding exposure and less than 4% token visibility. Moreover, even when 10% of tokens are maliciously forged by COLA, CoIn still maintains a 40.1% probability of successful detection.

<sup>3</sup><https://arcprize.org/blog/oai-o3-pub-breakthrough>

## 2 Related Work

**Reasoning Model.** LLMs have shown strong performance on complex reasoning tasks by generating intermediate steps, a technique known as chain-of-thought prompting Wei et al. [2022]. This paradigm has been further enhanced by methods such as self-consistency Wang et al. [2022] and program-aided reasoning Gao et al. [2023]. Recent research reveals that generating more reasoning steps at inference time can lead to higher answer accuracy, a phenomenon referred to as the test-time scaling law Snell et al. [2024], which has become a guiding principle for optimizing LLMs. Reasoning models are typically LLMs fine-tuned via RL Rafailov et al. [2023], Wu et al. [2023], Ramesh et al. [2024] to produce structured reasoning traces before generating final answers, thereby improving answer quality. These reasoning traces are often longer, more indirect, and may include failed attempts, but are nonetheless closely tied to the final answer Hao et al. [2024], Yang et al. [2025]. Since these reasoning tokens are generated in the same autoregressive manner as answer tokens, COLA charge for them based on token count. However, the indirect and verbose nature of reasoning makes it challenging to audit their legitimacy without direct access to the reasoning traces themselves.

**COLA Auditing.** Several works have emerged to address the lack of transparency in COLA. Cai et al. [2025] proposes a watermark-based method to audit whether a COLA uses the required LLM rather than a cheaper LLM. Similarly, Yuan et al. [2025] develops a user-verifiable protocol to detect nodes that run unauthorized or incorrect LLM in a multi-agent system. Another series of works Zheng et al. [2025], Marks et al. [2025] proposes auditing some bad behaviors of LLMs, e.g., cheating and offensive outputs. These techniques mainly focus on the model auditing and lack attention to the token count auditing of COLA.

## 3 Preliminary

**Participants and Problem Formulation.** The CoIn framework involves three roles: (i) COLA — a commercial LLM service provider (e.g., OpenAI) that performs multi-step reasoning and returns only the final output to the user; (ii) User — an end-user who submits a prompt and receives an answer along with a billing summary; and (iii) CoIn auditor — a trusted third party responsible for verifying the invisible reasoning tokens on behalf of the user.

In each service interaction, the user sends a prompt  $P$  to COLA. The LLM generates reasoning tokens  $R = \{r_1, r_2, \dots, r_m\}$ , followed by answer tokens  $A = \{a_1, a_2, \dots, a_n\}$ . Only the final answer  $A$  is returned to the user, while the reasoning trace  $R$  remains hidden. Billing is based on the total number of tokens  $m + n$ , including the invisible reasoning tokens. As Figure 1 shows, reasoning tokens often dominate the total count, i.e.,  $m \gg n$ , resulting in a significant transparency gap.

**Token Count Inflation.** We consider two strategies for inflating token counts:

- **Naive token count inflation.** COLA reports a falsified token count  $m_f > m$ , leading to direct overbilling without modifying the output.
- **Adaptive token count inflation.** Anticipating user-side defenses (e.g., hash matching, spot-checking), COLA may append low-effort fabricated reasoning tokens to the original reasoning trace. These fabricated tokens can be generated via random sampling, retrieval from related documents, or repetition of existing tokens, and then indistinguishably mixed with genuine reasoning tokens. The inflated sequence is then used for billing, bypassing naive verification methods and still overcharging the user.

To address these threats, CoIn employs two components: (1) **Token Quantity Verification**, which audits the reported token count using verifiable commitments and exposes embeddings; and (2) **Semantic Validity Verification**, which evaluates the relevance between reasoning and answer tokens to detect low-quality injections.

**Threat Model.** COLA has access to the user prompt  $P$ , the full reasoning trace  $R$ , and the answer  $A$ , and controls the billing report  $(m, n)$ , where  $m$  is the claimed number of reasoning tokens and  $n$  is the number of answer tokens. It can manipulate the reported count without user visibility. The CoIn auditor operates as a trusted third party. It can access  $P$ ,  $A$ , and  $(m, n)$ , but cannot observe  $R$  directly or directly query the LLM used by COLA. However, it can request COLA to return the embeddings of  $R$ , computed using an embedding model fixed by the auditor to prevent tampering.

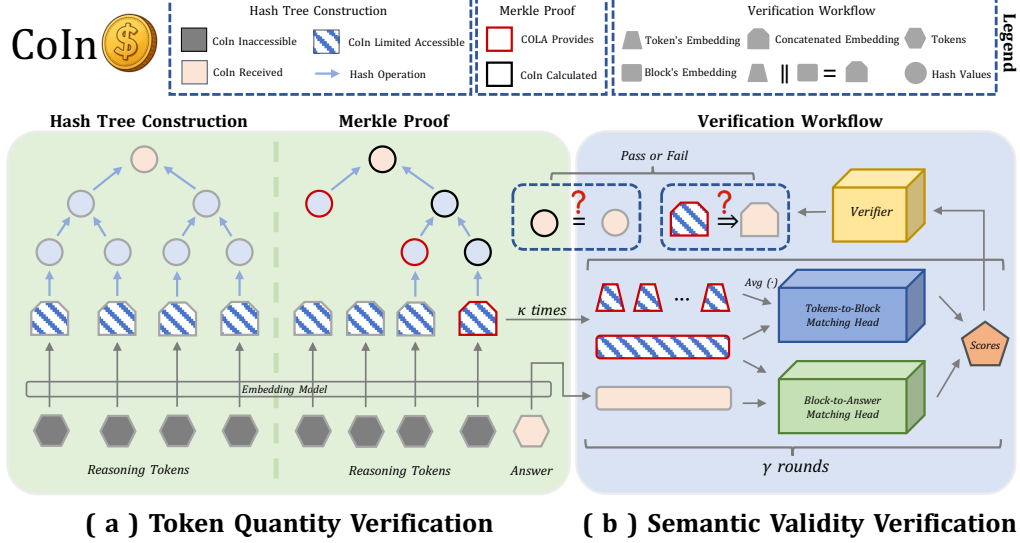


Figure 2: CoIn Framework.

## 4 CoIn: Counting the Invisible Reasoning Tokens

CoIn comprises two complimentary components: **token quantity verification** and **semantic validity verification**. The token quantity verification module treats embeddings of invisible reasoning tokens as cryptographic fingerprints and organizes them into a verifiable hash tree. By querying a small subset of these fingerprints, users can audit the claimed number of invisible tokens without accessing their contents. The semantic validity verification module trains a lightweight neural network, referred to as a *matching head*, to evaluate the relevance between embeddings. During auditing, CoIn retrieves token embeddings from the hash tree and uses the matching head to compute relevance scores both among reasoning tokens and between reasoning and answer tokens. These scores help detect token count inflation through the injection of fabricated or irrelevant reasoning tokens. An overview of the CoIn framework is illustrated in Figure 2.

### 4.1 Token Quantity Verification

**Token Fingerprint Generation.** In CoIn, COLA is required to generate embeddings of its reasoning tokens using a third-party embedding model  $\text{Embd}(\cdot)$  designated by the CoIn auditor. These embeddings serve as token fingerprints used to construct a verifiable hash tree for auditing. This hash tree enables CoIn to audit the total number of invisible tokens without accessing the tokens themselves.

Specifically, given a reasoning sequence  $R$ , COLA first partitions  $R$  into  $\alpha$  blocks. For each token  $r_i$  in block  $B_j$ , COLA computes: (i) the block embedding  $\text{Embd}(B_j)$ , which embeds all the tokens inside the block; and (ii) the token embedding  $\text{Embd}(r_i)$ , which embeds the single token itself. Each reasoning token therefore acquires both the block embedding and the token embedding. For each reasoning token  $\text{Embd}(r_i)$ , CoIn concatenated its block embedding and token embedding to form the token fingerprint:  $\text{Embd}(B_j) \parallel \text{Embd}(t_i)$ .

**Fingerprint Hash Tree Construction.** COLA applies a cryptographic hash function (e.g., SHA-256), agreed upon with CoIn, to each token fingerprint to construct the leaf nodes of a Merkle Hash Tree Merkle [1987]. The number of leaf nodes is padded to the nearest power of two, and parent nodes are built recursively by hashing concatenated sibling nodes up to the Merkle Root. This root serves as a commitment to the full set of reasoning tokens and is submitted to CoIn. After constructing the hash tree, COLA give the Merkle Root to CoIn for Merkle Proofs upon user’s auditing request.

**Merkle Proof.** Upon receiving the answer  $A$  and the token counts  $m$  and  $n$ , a user may suspect token inflation. To verify the count of invisible reasoning tokens, the user selects a block  $B_j$  and randomly chooses token indices to audit. Upon receiving the request, CoIn auditor requests the following

information from COLA: (i) the fingerprints of the selected tokens; and (ii) the corresponding Merkle Path, which is a sequence of sibling hashes needed to reconstruct the Merkle Root from the corresponding token. CoIn recomputes the Merkle root from the provided data and checks for consistency with the original commitment by COLA provider. A successful match confirms the integrity of the selected token; a mismatch indicates possible fabrication and inflated token reporting. The construction and Merkle Proof procedure is illustrated in Figure 2-(a) and detailed further in Appendix D.1 , D.2.

The Merkle proof in token quantity verification ensures both the structural integrity and the correctness of the reported token count, effectively defending against naive token count inflation. However, a dishonest COLA may still conduct adaptive token count inflation by injecting irrelevant or low-effort fabricated tokens that pass count verification. To address this limitation, we introduce semantic validity verification.

## 4.2 Semantic Validity Verification

To defend against adaptive token count inflation, we introduce the semantic validity verification component, as illustrated in Figure 2-(b). This component ensures that reasoning tokens are semantically meaningful and contribute to the final answer, preventing low-effort or fabricated token insertion. Based on this principle, CoIn verifies the semantic validity of invisible tokens from two perspectives:

- **Token-to-Block verification** checks whether each reasoning token  $r_i$  is semantically coherent within its enclosing block  $B_j$ . This defends against randomly injected or meaningless tokens.
- **Block-to-Answer verification** evaluates whether a reasoning block  $B_j$  is semantically aligned with the final answer  $A$ , thus identifying the insertion of low-cost content that is insufficiently relevant to the task.

To support both tasks, CoIn trains two lightweight neural modules called the *matching heads*, which are binary classifiers that determines whether two embeddings are semantically associated. Given two token embeddings  $a$  and  $b$ , the matching head first computes the cosine similarity:  $\cos\_sim = \frac{a \cdot b}{\|a\| \|b\|}$ , and constructs the feature vector:

$$h = [a; b; a - b; a \odot b; \cos\_sim] \in \mathbb{R}^{4d+1},$$

where  $d$  is the embedding dimension,  $[\cdot]$  denotes concatenation, and  $\odot$  denotes element-wise multiplication. The feature  $h$  is then passed through a two-layer feedforward network to produce a scalar match score  $S \in [0, 1]$ , representing the likelihood that  $a$  and  $b$  are semantically aligned. This process can be viewed as a regression function  $S = \text{MH}(a, b)$ .

In CoIn, the matching heads  $\text{MH}_{tb}(\cdot)$ ,  $\text{MH}_{ba}(\cdot)$  are trained offline for token-to-block and block-to-answer verification respectively. CoIn use open-source corpora and the same embedding model in token fingerprinting to build the datasets for matching heads training.

**Verification Protocol.** In each verification round, the user randomly selects some reasoning tokens  $r_i$  (by default, 10% of the tokens within a selected block) from the hash tree. Since the token fingerprint consists of both the token embedding  $\text{Embd}(r_i)$  and the corresponded block embedding  $\text{Embd}(B_j)$ , it can be directly used for Tokens-to-Block verification. For the Block-to-Answer verification, we use  $\text{Embd}(B_j)$  and the embedding of the whole answer to compute the score:

$$S_{tb} = \text{MH}_{tb}(\text{AVG}(\text{Embd}(r_i)), \text{Embd}(B_j)), \quad S_{ba} = \text{MH}_{ba}(\text{Embd}(B_j), \text{Embd}(A)). \quad (1)$$

Here,  $S_{tb}$  and  $S_{ba}$  represent the relevance scores for the two respective verification tasks. Each score reflects the estimated likelihood that the two input embeddings are semantically relevant.

## 4.3 Workflow of CoIn

**Enforcing Billing Integrity with CoIn.** When a user suspects token count inflation in a specific response, they can initiate an audit request to CoIn. The audit begins with the user selecting a fraction  $\gamma$  of the total reasoning blocks for verification. CoIn then performs two Semantic Validity Verifications and multiple Merkle Proofs on these selected blocks. The resulting match scores are passed to a verifier, which issues a final decision. If the verifier accepts, the audit concludes successfully. If the verifier rejects, the user continues by randomly selecting another unverified block for auditing. This process repeats until either a successful judgment is reached or all blocks are

exhausted. If no verification passes, the audit concludes with COLA being flagged for token inflation. The user may then request COLA to justify the charges by disclosing the original reasoning content. The complete procedure is outlined in Algorithm 4.

**Verifier Design.** Each audit round produces a variable-length sequence of match scores, as the number of verified blocks depends on verifier decisions. To handle this, we implement two types of verifiers: (i) **Rule-based:** Averages the scores from two semantic verifications. The audit passes if both averages exceed a threshold  $\tau$ . (ii) **Learning-based:** Uses a lightweight DeepSets model Zaheer et al. [2017] to process the unordered set of match scores and audit will succeed if the confidence exceeds  $\tau$ .

Auditing outcomes enable users to assess the trustworthiness of a COLA provider. Frequent failures in CoIn audits may erode user trust and damage provider reputation. By introducing verifiable accountability, the CoIn framework serves as a deterrent against token count inflation in commercial LLM services.

**Hyperparameter and Verification Cost.** CoIn is governed by a few hyperparameters that control auditing granularity and cost. Specifically,  $\alpha$  is the number of blocks,  $\beta$  the block size,  $\gamma$  the initial sampling ratio (default: 0.3), and  $k$  the number of tokens sampled per block (default:  $0.1 * \beta$ ). A smaller  $\beta$  reduce exposure but increase overhead. The protocol begins with  $\gamma \cdot \alpha$  rounds and may proceed up to  $\alpha$  rounds under early stopping, so the number of verification rounds satisfies  $\ell \in [\gamma \cdot \alpha, \alpha]$ . As a result, the total number of Merkle Proofs is  $k \cdot \ell$ , and the number of Semantic Judgments is  $2 \cdot \ell$ .

## 5 Experiments

We systematically evaluate the robustness and reliability of CoIn and its submodules under various adaptive inflation attacks across multiple datasets. We further analyze the construction cost of the Hash Tree, as well as whether the partially exposed block embeddings and tokens can be exploited to recover the reasoning tokens of COLA. Finally, we assess the difficulty of the dataset we constructed.

### 5.1 Experiment Setup

**Token Inflation Implementations.** We study both *naive* and *adaptive* token count inflation strategies. To enable fine-grained evaluation and systematic dataset construction, we design four variants of adaptive inflation. All inflation types used in our experiments are summarized in Table 1. These strategies are applied to generate inflated samples for both training and evaluation.

Table 1: Token inflation types used in our experiments.

Type	Description
<b>Naive Inflation</b>	Randomly select tokens from the vocabulary for injection.
<b>Ada. Inflation 1</b>	Inject tokens with embeddings similar to $P$ , $R$ , or $A$ .
<b>Ada. Inflation 2</b>	Inject tokens directly sampled from $P$ , $R$ , or $A$ .
<b>Ada. Inflation 3</b>	Inject reasoning sequences extracted from other inputs.
<b>Ada. Inflation 4</b>	Inject retrieved sequences semantically similar to $P$ , $R$ , or $A$ .

**Datasets and Training Setup.** We conduct experiments on five datasets derived from DeepSeek-R1 DeepSeek-AI [2025], covering diverse reasoning domains: medical Chen et al. [2024], code Team [2025], Face [2025], mathematics Face [2025], general reasoning<sup>4</sup>, and out-of-domain (OOD) mathematics Team [2025], Face [2025]. For training, we randomly sample 20,000 examples from each dataset and combine them into a joint dataset. Another 1,000 samples per dataset are held out to form the evaluation set for the CoIn framework. We use the tokenizer of DeepSeek-R1 in our experiments.

For the matching head, we use all-MiniLM-L6-v2 Reimers and Gurevych [2019] as model structure. In the **token-to-block** verification task, we treat original samples as normal instances and apply Naive Inflation as well as Adaptive Inflation 1 and 2 to construct inflated samples. Normal and inflated samples are labeled 0 and 1 respectively and mixed at a 1:1 ratio to form the training set.

<sup>4</sup><https://huggingface.co/datasets/glaiveai/reasoning-v1-20m>

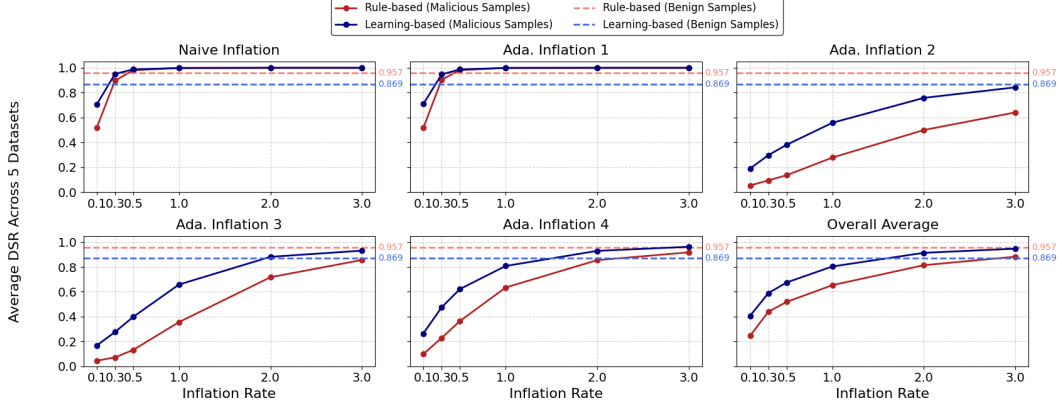


Figure 3: Performance of CoIn across different inflation methods and verifiers. The red lines and the blue lines represent the DSR of rule-based verifier and learning-based verifier, respectively.  $\gamma$

For the **block-to-answer** verification task, we adopt a similar setup, but use Ada. Inflation 1–4 to construct inflated samples. This setting enables a thorough evaluation of the model’s ability to detect both shallow and semantically sophisticated inflation attacks. The details of training and dataset are explained in Appendix A, B.

**Metrics.** We define the *Detection Success Rate (DSR)* as the classification accuracy of our module, computed separately for malicious samples and benign samples. *Inflation Rate (IR)* indicates the percentage of fabricated tokens injected by COLA relative to the number of original reasoning tokens. For benign samples, the *Average Exposure Rate (AER)* refers to the proportion of blocks exposed during the CoIn multi-step verification process out of the total number of blocks.

## 5.2 Detection Performance of CoIn

We evaluate CoIn’s ability to detect various token count inflation attacks. Figure 3 shows the relationship between IR and DSR across five datasets. DSR increases with IR, indicating that more aggressive inflation is easier to detect. For example, in the Naive Inflation setting, DSR approaches 100% when IR exceeds 0.5, suggesting near-perfect detection for heavily inflated queries. Figure 4 illustrates the impact of  $\tau_h$ . As  $\tau_h$  increases, DSR for malicious samples increases, while for benign samples decreases. This highlights a tunable trade-off between user utility and COLA’s protection.

**Learning-based Verifier Excels at Detecting Malicious Samples.** For a fair comparison, we set the threshold  $\tau = 0.5$  for learning-based verifier, and  $\tau = 0.6$  for rule-based verifier. The initial sampling ratio  $\gamma$  is set to 0.3. Across all settings, the learning-based verifier consistently outperforms the rule-based variant in detecting inflated samples, achieving up to 94.7% average DSR at an inflation ratio (IR) of 3.0. Among the adaptive methods, Ada. Inflation 2 and Ada. Inflation 3 present greater detection challenges. Nevertheless, CoIn still achieves an average DSR of approximately 84.3% and 93.1% at IR = 3.0 for these cases. These results demonstrate that CoIn remains robust even under strong adaptive inflation, and that learning-based semantic verification offers significant advantages in practical detection scenarios.

### Rule-based Verifier Excels at Handling Benign Samples.

Although it is less effective at detecting inflated samples, the rule-based verifier performs better on benign samples when properly tuned, due to its lower false positive rate. Given our verification mechanism, a higher DSR on benign samples results in a lower Average Exposure Rate (AER), which makes COLA more favorable toward such mechanisms, potentially even negotiating certain parameters with users. Based on the rule-based verifier ( $\tau = 0.6$ ), Table 2 reports the average number of blocks  $\alpha$ , average verification rounds  $\ell$  and AER across different block sizes  $\beta$ . With increasing

Table 2: Influence of Block Size.

Metric	Block Size $\beta$		
	256	512	1024
Avg. $\alpha$	16.8	8.6	4.5
Avg. $\ell$	6.3	3.7	2.2
AER $\downarrow$	0.38	0.43	0.49

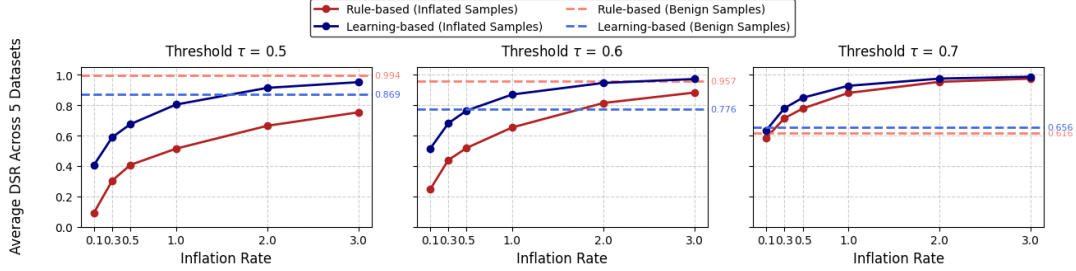


Figure 4: Impact of threshold  $\tau$  on DSR.

$\beta$ ,  $\ell$  decreases, indicating lower cost. However, AER increases, and the results in Appendix E show that increasing the block size reduces the DSR for adaptive inflation.

### 5.3 Performance of the Semantic Validity Verification

**Block-to-Answer Verification Task.** We separately evaluate the performance of the two types of matching heads introduced in Section 4.2. Table 3 shows the DSR of the matching heads on the *Block-to-Answer Verification* task. The model achieves an average DSR of 94.8% across attack types. Even for the Math (OOD) dataset, which was excluded from training, the model performs strongly, indicating good generalization. The DSR drops slightly on clean (non-inflated) samples due to the presence of reasoning blocks not directly contributing to the final answer (see Section 6). Additionally, Content Reuse 2 attacks introduce hard negatives that resemble real data, making separation more difficult.

Table 3: Block-to-Answer Verification Performance Across Attack Types and Domains.

Attack Type	Medical	Code	Math	General	Math (OOD)	Avg.
Naive Inflation	99.4	100.0	100.0	99.3	100.0	99.7
Ada. Inflation 1	95.3	98.7	98.6	96.8	98.2	97.5
Ada. Inflation 2	94.4	92.3	92.8	94.2	92.7	93.3
Ada. Inflation 3	89.2	81.5	84.3	92.9	84.6	86.5
Ada. Inflation 4	94.2	97.9	99.0	96.1	97.8	97.0
Avg. With Inflation	94.5	94.1	94.9	95.8	94.7	94.8
No Inflation	87.9	90.3	87.1	86.5	87.9	87.9

**Tokens-to-Block Verification Task.** Table 4 shows the results for token-to-block verification. The model performs well overall but struggles with Adaptive Inflation 2, where tokens reused from the same sample lead to significant lexical and semantic overlap. This overlap can blur the distinction between original and fabricated content, especially when reused tokens legitimately contribute to the block.

Table 4: Tokens-to-Block Verification Performance Across Attack Types and Domains.

Attack Type	Medical	Code	Math	General	Math (OOD)	Avg.
Naive Inflation	90.8	90.5	95.3	84.5	94.6	91.2
Ada. Inflation 1	95.1	96.1	95.8	95.5	95.8	95.6
Ada. Inflation 2	76.0	75.2	73.9	73.6	74.8	74.7
Avg. With Inflation	87.3	87.2	88.4	84.5	88.4	87.2
No Inflation	82.0	80.4	87.2	79.0	86.0	82.9

**Cost of Building Hash Trees.** We evaluate the computational overhead of constructing the Merkle hash tree, with respect to input size and hidden dimension. Experiments were conducted on a dual-socket AMD EPYC 7763 system (128 cores, 256 threads). All constructions ran as single-threaded



processes on one logical core. As shown in Figure 5, the construction time grows approximately linearly with the input length for a fixed hidden dimension, and increases more steeply with higher dimensions. Given that most LLM inference servers have underutilized CPUs, and the Merkle Tree construction process scales effectively with multi-core parallelism, the practical cost of building the hash tree is nearly negligible.

## 6 Discussion

### Can the original text be recovered from the tokens and embeddings exposed by COLA?

During the verification process in CoIn, COLA leaks a certain number of block embeddings and tokens within the blocks to CoIn. To quantify the impact of such leakage, we assume a malicious CoIn leverages an RAG system to retrieve documents highly similar to the exposed embeddings and tokens, then feeds all retrieved information into an LLM to reconstruct the original content. The design and further details are provided in Appendix F. We randomly selected 100 samples from a mathematical dataset. We evaluated the similarity between the reconstructed blocks and the original ones using embedding similarity, BLEU score Papineni et al. [2002], ROUGE-L Lin [2004], and BERTScore Zhang et al. [2019]. As shown in Table 5, we observe that a high BERTScore/EmbedSim combined with low BLEU/ROUGE indicates that the LLM successfully preserves the core semantics, while significantly differing from the real block in terms of surface expression and syntactic structure.

Table 5: Similarity Between Blocks Reconstructed by CoIn and Real Blocks.

Metric	Block Size $\beta$		
	256	512	1024
EmbedSim	0.65	0.66	0.75
BLEU	0.04	0.05	0.03
ROUGE-L	0.23	0.25	0.24
BERTScore	0.83	0.83	0.84

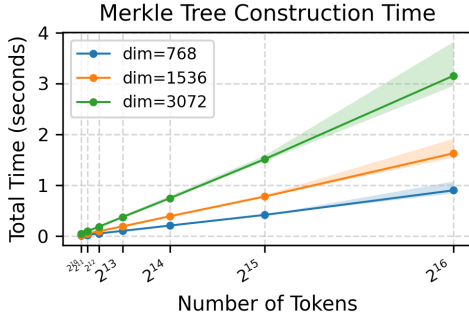


Figure 5: Merkle Tree Construction Time with Fluctuation Range.

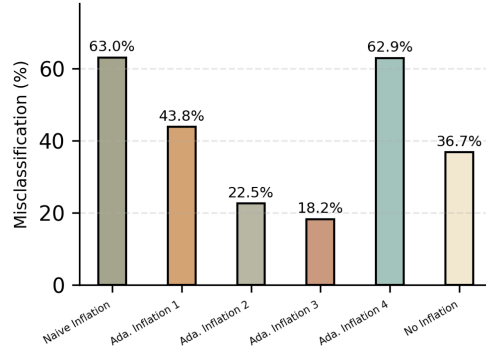


Figure 6: Misclassification Rates of LLMs on Constructed Datasets.

**How difficult is the dataset we constructed?** To investigate the dataset difficulty, we submitted the failed samples from the *Block-to-Answer Verification* task, along with their Answer, to a LLM. Based on the idea of LLM-as-a-Judge Zheng et al. [2023], Li et al. [2024], we use a prompt to instruct the LLM to perform binary classification. The prompt used is provided in Appendix F. The relatively high misclassification rate suggests that the LLM, after reading the original text, tends to align with the matching head’s judgment. The LLM shows high error rates on Naive Inflation, Ada Inflation 1 and 4, indicating strong performance of the matching head in these cases. However, it still struggles with the remaining two adaptive inflations. Notably, 36.7% of real blocks were misclassified by the LLM, suggesting that some parts of the true reasoning steps may be unrelated to answer derivation.

## 7 Limitations

We acknowledge that CoIn, despite its merits, possesses certain limitations that warrant discussion.

- Firstly, CoIn exhibits suboptimal performance in the detection of malicious samples when the Inflation Rate is low. However, it is pertinent to note that under such circumstances, the incentive for COLA to engage in data falsification is also correspondingly diminished.
- Secondly, CoIn is inherently probabilistic, and as such, it is susceptible to a non-zero misclassification rate. Consequently, when benign samples are erroneously identified as malicious, the protocol of CoIn necessitates that COLA discloses the original text to the user for verification.
- Thirdly, the auditing process facilitated by CoIn requires the active cooperation of COLA. Ideally, COLA itself could deploy CoIn to attest to its own integrity. This would allow COLA to continue concealing its reasoning tokens, thereby mitigating the risk of its proprietary model being subjected to distillation attacks.
- Finally, CoIn comprises multiple small-scale neural network components and is not architected as an end-to-end system. Nevertheless, this modular design confers a distinct advantage: it permits the independent training of each module, which significantly enhances both the convergence speed and the overall efficacy of the training process.

## 8 Conclusion

This paper presents CoIn, a novel auditing framework designed to verify the token counts and semantic validity of hidden reasoning traces in COLA. We identify and formalize the problem of *token count inflation*, in which service providers can overcharge users by injecting redundant or fabricated reasoning tokens that are not visible to the user. To address this, CoIn integrates two complementary components: a hash tree-based token quantity verifier and a semantic relevance-based validity checker. Our extensive experiments demonstrate that CoIn can detect both naive and adaptive inflation strategies with high accuracy, even under limited exposure settings. By enabling transparent and auditable billing without revealing proprietary content, CoIn introduces a practical mechanism for accountability in commercial LLM services. We hope this work lays the foundation for future research on LLM API auditing, transparent reasoning, and verifiable inference services.

## References

- Will Cai, Tianneng Shi, Xuandong Zhao, and Dawn Song. Are you getting what you pay for? auditing model substitution in llm apis. *arXiv preprint arXiv:2504.04715*, 2025.
- Junying Chen, Zhenyang Cai, Ke Ji, Xidong Wang, Wanlong Liu, Rongsheng Wang, Jianye Hou, and Benyou Wang. Huatuoqpt-o1, towards medical complex reasoning with llms, 2024. URL <https://arxiv.org/abs/2412.18925>.
- DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL <https://arxiv.org/abs/2501.12948>.
- Hugging Face. Open r1: A fully open reproduction of deepseek-r1, January 2025. URL <https://github.com/huggingface/open-r1>.
- Xingcheng Gao, Swaroop Mishra, et al. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2023.
- Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6):1789–1819, 2021.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
- Shibo Hao, Yi Gu, Haotian Luo, Tianyang Liu, Xiyan Shao, Xinyuan Wang, Shuhua Xie, Haodi Ma, Adithya Samavedhi, Qiyue Gao, et al. Llm reasoners: New evaluation, library, and analysis of step-by-step reasoning with large language models. *arXiv preprint arXiv:2404.05221*, 2024.
- Zhiqiang Hu, Lei Wang, Yihuai Lan, Wanyu Xu, Ee-Peng Lim, Lidong Bing, Xing Xu, Soujanya Poria, and Roy Ka-Wei Lee. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models. *arXiv preprint arXiv:2304.01933*, 2023.

- 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.
- Mingyu Jin, Qinkai Yu, Dong Shu, Haiyan Zhao, Wenyue Hua, Yanda Meng, Yongfeng Zhang, and Mengnan Du. The impact of reasoning step length on large language models. *arXiv preprint arXiv:2401.04925*, 2024.
- Komal Kumar, Tajamul Ashraf, Omkar Thawakar, Rao Muhammad Anwer, Hisham Cholakkal, Mubarak Shah, Ming-Hsuan Yang, Phillip HS Torr, Fahad Shahbaz Khan, and Salman Khan. Llm post-training: A deep dive into reasoning large language models. *arXiv preprint arXiv:2502.21321*, 2025.
- Haitao Li, Qian Dong, Junjie Chen, Huixue Su, Yujia Zhou, Qingyao Ai, Ziyi Ye, and Yiqun Liu. Llm-as-judges: a comprehensive survey on llm-based evaluation methods. *arXiv preprint arXiv:2412.05579*, 2024.
- Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.
- Samuel Marks, Johannes Treutlein, Trenton Bricken, Jack Lindsey, Jonathan Marcus, Siddharth Mishra-Sharma, Daniel Ziegler, Emmanuel Ameisen, Joshua Batson, Tim Belonax, et al. Auditing language models for hidden objectives. *arXiv preprint arXiv:2503.10965*, 2025.
- Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.
- Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*, 2025.
- Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.
- Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36:53728–53741, 2023.
- Shyam Sundhar Ramesh, Yifan Hu, Iason Chaimalas, Viraj Mehta, Pier Giuseppe Sessa, Haitham Bou Ammar, and Ilija Bogunovic. Group robust preference optimization in reward-free rlhf. *Advances in Neural Information Processing Systems*, 37:37100–37137, 2024.
- Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. URL <https://arxiv.org/abs/1908.10084>.
- Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.
- Sharath Turuvekere Sreenivas, Saurav Muralidharan, Raviraj Joshi, Marcin Chochowski, Ameya Sunil Mahabaleshwarkar, Gerald Shen, Jiaqi Zeng, Zijia Chen, Yoshi Suhara, Shizhe Diao, et al. Llm pruning and distillation in practice: The minitron approach. *arXiv preprint arXiv:2408.11796*, 2024.
- Open Thoughts Team. Open Thoughts, January 2025.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, et al. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, et al. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

- Tianhao Wu, Banghua Zhu, Ruoyu Zhang, Zhaojin Wen, Kannan Ramchandran, and Jiantao Jiao. Pairwise proximal policy optimization: Harnessing relative feedback for llm alignment. *arXiv preprint arXiv:2310.00212*, 2023.
- Shu Yang, Junchao Wu, Xin Chen, Yunze Xiao, Xinyi Yang, Derek F Wong, and Di Wang. Understanding aha moments: from external observations to internal mechanisms. *arXiv preprint arXiv:2504.02956*, 2025.
- Michael J Yuan, Carlos Campoy, Sydney Lai, James Snewin, and Ju Long. Trust, but verify. *arXiv preprint arXiv:2504.13443*, 2025.
- Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. *Advances in neural information processing systems*, 30, 2017.
- Jintian Zhang, Yuqi Zhu, Mengshu Sun, Yujie Luo, Shuofei Qiao, Lun Du, Da Zheng, Huajun Chen, and Ningyu Zhang. Lightthinker: Thinking step-by-step compression. *arXiv preprint arXiv:2502.15589*, 2025.
- Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*, 2019.
- Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, 2023.
- Xiang Zheng, Longxiang Wang, Yi Liu, Xingjun Ma, Chao Shen, and Cong Wang. Calm: Curiosity-driven auditing for large language models. *arXiv preprint arXiv:2501.02997*, 2025.
- Barret Zoph, Golnaz Ghiasi, Tsung-Yi Lin, Yin Cui, Hanxiao Liu, Ekin Dogus Cubuk, and Quoc Le. Rethinking pre-training and self-training. *Advances in neural information processing systems*, 33: 3833–3845, 2020.

## A Dataset Construction and Experimental Details

---

### Algorithm 1 Streamlined Generation of Inflated Reasoning Sequences

---

**Require:** Original dataset  $D_{orig}$ , inflation ratios  $\mathcal{K}$ , strategies  $S_{list}$  with weights  $W_S$ , tokenizer  $\mathcal{T}$ , embedder  $\mathcal{E}$ , anchor source  $Src_{anchor}$ , segment length range  $[L_{min}, L_{max}]$ , insertion mode  $M_{ins}$ , and optional block range  $[B_{min}, B_{max}]$  if using block mode.

**Ensure:** Inflated dataset  $D_{inflated}$

```

1: Initialize  $D_{inflated} \leftarrow \emptyset$ 
2: Build FAISS indexes for RAG-based strategies
3: for each data point  $item_i = (P_i, R_i, A_i)$  in  $D_{orig}$  do
4:    $T_{orig} \leftarrow \mathcal{T}(R_i)$ ;
5:   if  $T_{orig}$  is empty then continue
6:   end if
7:    $T_{anchor} \leftarrow \text{SelectAnchor}(item_i, Src_{anchor})$ 
8:    $N_{max} \leftarrow \lfloor |T_{orig}| \cdot \max(\mathcal{K}) \rfloor$ 
9:    $T_{pool} \leftarrow \text{CollectTokens}(N_{max}, T_{anchor}, S_{list}, W_S)$ 
10:  for each  $k \in \mathcal{K}$  do
11:     $N_k \leftarrow \lfloor |T_{orig}| \cdot k \rfloor$ 
12:     $T_k \leftarrow \text{Subsample}(T_{pool}, N_k)$ 
13:     $T_{final} \leftarrow \text{Insert}(T_{orig}, T_k, M_{ins}, [B_{min}, B_{max}])$ 
14:    Add  $\mathcal{T}^{-1}(T_{final})$  to  $D_{inflated}$  with metadata
15:  end for
16: end for
17: return  $D_{inflated}$ 

```

---

### A.1 Dataset Construction Details

We construct two verification datasets for Block-to-Answer and Token-to-Block verification, each dataset includes two types of inflated samples. The simple version consists entirely of artificially generated (inflated) tokens, while the hard version contains a mixture of real and inflated tokens. For Token-to-Block verification, we randomly sample between 3.125% and 12.5% of tokens from each block to create both training and test instances.

For both verification tasks, we generate 1,200,000 positive and negative samples respectively. The training set is uniformly distributed across four datasets. Since the difficulty levels of the samples vary, we adjust the composition using an adaptive inflation strategy (applied in Block-to-Answer) to ensure balanced learning.

For training the DeepSets model, we additionally sample 1,000 examples. To preserve generalization capability, the data used for training this model does not overlap with any samples seen by the matching heads.

### A.2 Experimental Details

All evaluation results, unless stated otherwise, are reported on 1,000 examples. This applies to Block-to-Answer, Token-to-Block, and the test sets used within the CoIn framework. Each numeric result is computed over a minimum of 1,000 samples to ensure statistical significance. Please refer to the Algorithm 1 for our CoIn workflow test set construction process.

### A.3 Source of Dataset

To evaluate CoIn’s performance across different domains, we constructed training and test sets based on five datasets distilled from DeepSeek-R1 DeepSeek-AI [2025], including Medical Chen et al. [2024]<sup>5</sup>, Code Team [2025], Face [2025]<sup>6</sup>, Math Face [2025]<sup>7</sup>, General<sup>8</sup>, and Out-of-Domain data Math (OOD) Team [2025], Face [2025]<sup>9</sup>. Our final training set is a mixture of these five datasets.

## B Training and Model Details

For the matching heads used in Token-to-Block verification and Block-to-Answer verification, we set the learning rate to  $2 \times 10^{-5}$ , the batch size to 128, and train for 3 epochs. We employ the Adam optimizer and use the Focal Loss function. The hidden dimension of the model follows that of the embedding model, set to 384.

For the DeepSets model in the verifier, we use a batch size of 128, a hidden dimension of 256, and train for 5 epochs. We adopt the Adam optimizer with a learning rate of  $1 \times 10^{-3}$  and use the binary cross-entropy (BCE) loss. All experiments are conducted with a fixed random seed of 42.

## C Computational Resources

All experiments were conducted on a high-performance workstation running Ubuntu 20.04.6 LTS. The system is equipped with a dual-socket AMD EPYC 7763 processor, providing a total of 128 physical cores and 256 threads. For GPU acceleration, we utilized an NVIDIA RTX A6000 Ada graphics card.

---

<sup>5</sup><https://huggingface.co/datasets/FreedomIntelligence/Medical-R1-Distill-Data>

<sup>6</sup>[https://huggingface.co/datasets/open-r1/OpenThoughts-114k-Code\\_decontaminated](https://huggingface.co/datasets/open-r1/OpenThoughts-114k-Code_decontaminated)

<sup>7</sup><https://huggingface.co/datasets/open-r1/OpenR1-Math-220k>

<sup>8</sup><https://huggingface.co/datasets/glaiveai/reasoning-v1-20m>

<sup>9</sup><https://huggingface.co/datasets/open-r1/OpenThoughts-114k-math>

## D Details of CoIn

### D.1 Merkle Tree Construction

Algorithm 3 details the process COLA uses to construct the Merkle Hash Tree from a reasoning sequence  $R$ . This corresponds to the "Token Fingerprint Generation" and "Fingerprint Hash Tree Construction" paragraphs.

### D.2 Merkle Proof Verification

Algorithm 2 describes how the CoIn auditor verifies the integrity of a token using its fingerprint and the Merkle path provided by COLA. This corresponds to the "Merkle Proof" paragraph.

---

**Algorithm 2** Merkle Proof Verification

---

**Require:** Committed Merkle Root  $MR_{committed}$  (from COLA).  
**Require:** Token fingerprint  $fp_{token}$  of the audited token (from COLA).  
**Require:** Merkle Path  $P = [(h_1, pos_1), (h_2, pos_2), \dots, (h_d, pos_d)]$  (from COLA), where  $h_k$  is a sibling hash and  $pos_k \in \{\text{'left'}, \text{'right'}\}$  indicates  $h_k$ 's position relative to the path node.  
**Require:** Cryptographic hash function  $H(\cdot)$ .  
**Ensure:** Boolean: **true** if verification succeeds, **false** otherwise.

```
1:  $current\_computed\_hash \leftarrow H(fp_{token})$   $\triangleright$  Hash the provided token fingerprint
2: for each pair  $(sibling\_hash, position) \in P$  do
3:   if  $position = \text{'left'}$  then
4:      $current\_computed\_hash \leftarrow H(sibling\_hash \parallel current\_computed\_hash)$ 
5:   else if  $position = \text{'right'}$  then
6:      $current\_computed\_hash \leftarrow H(current\_computed\_hash \parallel sibling\_hash)$ 
7:   else
8:     return false  $\triangleright$  Error: Invalid position in Merkle Path
9:   end if
10: end for
11:  $MR_{recomputed} \leftarrow current\_computed\_hash$ 
12: if  $MR_{recomputed} = MR_{committed}$  then
13:   return true  $\triangleright$  Verification successful: token integrity confirmed
14: else
15:   return false  $\triangleright$  Verification failed: mismatch indicates potential issue
16: end if
```

---

#### Notes on Algorithms:

- **Padding (Algorithm 3):** The text states, "The number of leaf nodes is padded to the nearest power of two." Algorithm 3 implements this by duplicating the hash of the last actual leaf node if leaves exist. If the initial set of tokens (and thus fingerprints) is empty ( $N = 0$ ), it assumes padding to  $N_{pow2} = 1$  using a hash of a predefined value (e.g., an empty string). The exact nature of this padding for an empty set should be consistently defined between COLA and the auditor.
- **Merkle Path Representation (Algorithm 2):** The Merkle Path  $P$  is assumed to be a list of (hash, position) tuples. The 'position' indicates if the sibling hash is to the 'left' or 'right' of the node on the direct path from the audited leaf to the root.
- **Concatenation for Hashing:** The order of concatenation (e.g.,  $H(leftChild \parallel rightChild)$  vs.  $H(rightChild \parallel leftChild)$ ) must be consistent throughout construction and verification. The algorithms assume a fixed order (left child first).

---

**Algorithm 3** Merkle Tree Construction by COLA

---

**Require:** Reasoning tokens  $R$ ; number of blocks  $\alpha$ ; embedding function  $\text{Embd}(\cdot)$ ; cryptographic hash function  $H(\cdot)$ .

**Ensure:** Merkle Root  $MR$ .

*// Phase 1: Token Fingerprint Generation and Leaf Node Creation*

- 1:  $Blocks \leftarrow \text{Partition}(R, \alpha)$   $\triangleright$  Partition  $R$  into  $B_1, \dots, B_\alpha$
- 2:  $Fingerprints \leftarrow \emptyset$   $\triangleright$  Initialize as an empty list
- 3: **for** each block  $B_j \in Blocks$  **do**
- 4:    $e_{block_j} \leftarrow \text{Embd}(B_j)$   $\triangleright$  Compute block embedding
- 5:   **for** each token  $r_i \in B_j$  **do**
- 6:      $e_{token_i} \leftarrow \text{Embd}(r_i)$   $\triangleright$  Compute token embedding
- 7:      $fp_i \leftarrow e_{block_j} \parallel e_{token_i}$   $\triangleright$  Form token fingerprint
- 8:     Add  $fp_i$  to  $Fingerprints$
- 9:   **end for**
- 10: **end for**
- 11:  $LeafNodes \leftarrow \emptyset$   $\triangleright$  Initialize as an empty list
- 12: **for** each fingerprint  $fp \in Fingerprints$  **do**
- 13:    $leaf \leftarrow H(fp)$   $\triangleright$  Hash fingerprint to create leaf node
- 14:   Add  $leaf$  to  $LeafNodes$
- 15: **end for**
- // Phase 2: Padding Leaf Nodes*
- 16:  $N \leftarrow \text{length}(LeafNodes)$
- 17: Let  $N_{pow2}$  be the smallest power of two such that  $N_{pow2} \geq N$ .
- 18: **if**  $N < N_{pow2}$  **then**
- 19:   **if**  $N = 0$  **and**  $N_{pow2} > 0$  **then**  $\triangleright$  e.g.,  $N = 0 \implies N_{pow2} = 1$
- 20:      $padding\_hash \leftarrow H("")$   $\triangleright$  Hash of empty string or other predefined padding value
- 21:     **for**  $k \leftarrow 1$  **to**  $N_{pow2}$  **do**
- 22:       Add  $padding\_hash$  to  $LeafNodes$
- 23:     **end for**
- 24:   **else if**  $N > 0$  **then**
- 25:      $last\_leaf\_hash \leftarrow LeafNodes[N - 1]$   $\triangleright$  Get hash of the last actual leaf
- 26:     **for**  $k \leftarrow 1$  **to**  $N_{pow2} - N$  **do**
- 27:       Add  $last\_leaf\_hash$  to  $LeafNodes$   $\triangleright$  Pad by duplicating the last leaf's hash
- 28:     **end for**
- 29:   **end if**
- 30: **end if**
- // Phase 3: Building the Tree Recursively*
- 31:  $CurrentLevelNodes \leftarrow LeafNodes$
- 32: **while**  $\text{length}(CurrentLevelNodes) > 1$  **do**
- 33:    $NextLevelNodes \leftarrow \emptyset$
- 34:   **for**  $k \leftarrow 0$  **to**  $(\text{length}(CurrentLevelNodes)/2) - 1$  **do**
- 35:      $leftChild \leftarrow CurrentLevelNodes[2k]$
- 36:      $rightChild \leftarrow CurrentLevelNodes[2k + 1]$
- 37:      $parentHash \leftarrow H(leftChild \parallel rightChild)$
- 38:     Add  $parentHash$  to  $NextLevelNodes$
- 39:   **end for**
- 40:    $CurrentLevelNodes \leftarrow NextLevelNodes$
- 41: **end while**
- 42: **if**  $\text{length}(CurrentLevelNodes) = 1$  **then**
- 43:    $MR \leftarrow CurrentLevelNodes[0]$   $\triangleright$  The single remaining node is the Merkle Root
- 44: **else**  $\triangleright$  Handles  $N = 0$  and  $N_{pow2} = 0$ , resulting in an empty  $CurrentLevelNodes$
- 45:    $MR \leftarrow H("")$   $\triangleright$  Define Merkle Root for an empty set of tokens, e.g., hash of empty string
- 46: **end if**
- 47: **return**  $MR$

---

### D.3 Workflow of CoIn

Algorithm 4 illustrates the complete verification procedure of CoIn.

---

**Algorithm 4** Multi-Round Verification in CoIn

---

**Require:** COLA Response (containing reasoning blocks  $\mathcal{B}_{\text{total}}$  and final answer  $A$ )  
**Require:** Fraction  $\gamma$  of blocks for initial verification (e.g., 0.1)  
**Require:** Pre-trained matching heads  $\text{MH}_{\text{tb}}(\cdot, \cdot)$ ,  $\text{MH}_{\text{ba}}(\cdot, \cdot)$   
**Require:** Embedding function  $\text{Embd}(\cdot)$   
**Require:** Verification threshold  $\tau$   
**Ensure:** Audit decision: "Successful" or "COLA Flagged for Inflation"

*// Initialization*  
1:  $\mathcal{B}_{\text{unverified}} \leftarrow \mathcal{B}_{\text{total}}$   
2:  $\mathcal{B}_{\text{verified}} \leftarrow \emptyset$   
3: *audit\_successful*  $\leftarrow$  **false**  
4: *all\_blocks\_audited*  $\leftarrow$  **false**  
*// Initial round of verification*  
5: Select an initial set of blocks  $\mathcal{B}_{\text{current\_round}} \subseteq \mathcal{B}_{\text{unverified}}$  of size  $\lceil \gamma \cdot |\mathcal{B}_{\text{total}}| \rceil$   
6: **if**  $\mathcal{B}_{\text{current\_round}}$  is empty **and**  $|\mathcal{B}_{\text{total}}| > 0$  **then**  
7:    $\mathcal{B}_{\text{current\_round}} \leftarrow$  one randomly selected block from  $\mathcal{B}_{\text{unverified}}$   
8: **end if**  
9: **while not** *audit\_successful* **and not** *all\_blocks\_audited* **do**  
10:   **if**  $\mathcal{B}_{\text{current\_round}}$  is empty **then**  
11:     *all\_blocks\_audited*  $\leftarrow$  **true**  $\triangleright$  No more blocks to check  
12:     **goto** *FinalDecision*  
13:   **end if**  
14:   *round\_scores*  $\leftarrow []$   $\triangleright$  Initialize as an empty list/array  
15:   **for** each block  $B_j \in \mathcal{B}_{\text{current\_round}}$  **do**  
16:     Randomly select a subset of reasoning tokens  $\{r_i\}_{i=1}^k$  from  $B_j$  (e.g., 10)  
17:      $E_{\text{tokens}} \leftarrow \text{AVG}(\{\text{Embd}(r_i)\}_{i=1}^k)$   $\triangleright$  Average embedding of selected tokens  
18:      $E_{\text{block}} \leftarrow \text{Embd}(B_j)$   
19:      $E_{\text{answer}} \leftarrow \text{Embd}(A)$   
20:      $S_{\text{tb}} \leftarrow \text{MH}_{\text{tb}}(E_{\text{tokens}}, E_{\text{block}})$   $\triangleright$  Token-to-Block score  
21:      $S_{\text{ba}} \leftarrow \text{MH}_{\text{ba}}(E_{\text{block}}, E_{\text{answer}})$   $\triangleright$  Block-to-Answer score  
22:     Add pair  $(S_{\text{tb}}, S_{\text{ba}})$  to *round\_scores*  
23:     CoIn performs Merkle Proofs on selected tokens in  $B_j$  (verification of token integrity)  
24:   **end for**  
25:    $\mathcal{B}_{\text{verified}} \leftarrow \mathcal{B}_{\text{verified}} \cup \mathcal{B}_{\text{current\_round}}$   
26:    $\mathcal{B}_{\text{unverified}} \leftarrow \mathcal{B}_{\text{unverified}} \setminus \mathcal{B}_{\text{current\_round}}$   
27:   *verifier\_decision*  $\leftarrow \text{VERIFIER}(\text{round\_scores}, \tau)$   $\triangleright$  Verifier can be rule-based or learning-based  
28:   **if** *verifier\_decision* = **Accept** **then**  
29:     *audit\_successful*  $\leftarrow$  **true**  
30:   **else**  
31:     **if**  $\mathcal{B}_{\text{unverified}}$  is empty **then**  
32:       *all\_blocks\_audited*  $\leftarrow$  **true**  
33:     **else**  
34:       Select one new random block  $B_{\text{next}}$  from  $\mathcal{B}_{\text{unverified}}$   
35:        $\mathcal{B}_{\text{current\_round}} \leftarrow \{B_{\text{next}}\}$   $\triangleright$  Next round verifies this single block  
36:     **end if**  
37:   **end if**  
38: **end while**  
*FinalDecision:*  
39: **if** *audit\_successful* **then**  
40:   **return** "Audit Successful"  
41: **else**  
42:   **return** "COLA Flagged for Token Inflation"  $\triangleright$  User may request COLA to justify charges  
43: **end if**

44: **function**  $\text{VERIFIER}(\text{scores\_list}, \tau)$   $\triangleright$  Example: Rule-based verifier  
45:   **if** *scores\_list* is empty **then** **return** "Reject"  
46:   **end if**  
47:   *avg\_S\_tb*  $\leftarrow$  average of all  $S_{\text{tb}}$  in *scores\_list*  
48:   *avg\_S\_ba*  $\leftarrow$  average of all  $S_{\text{ba}}$  in *scores\_list*  
49:   **if** *avg\_S\_tb*  $> \tau$  **and** *avg\_S\_ba*  $> \tau$  **then**  
50:     **return** "Accept"  
51:   **else**  
52:     **return** "Reject"  
53:   **end if**  $\triangleright$  Alternatively, a learning-based verifier (e.g., DeepSets) could be used here.  
54: **end function**

---



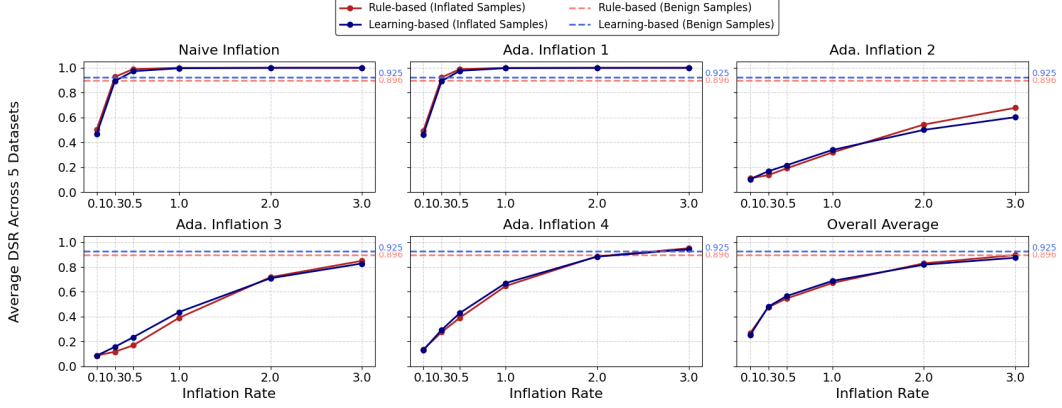


Figure 7: Performance of CoIn across different inflation methods and verifiers (Block Size = 512). The red lines and the blue lines represent the DSR of rule-based verifier and learning-based verifier, respectively.  $\gamma$

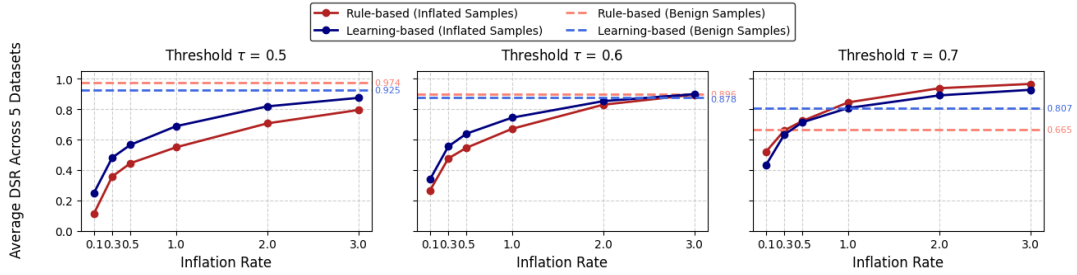


Figure 8: Impact of threshold  $\tau$  on DSR (Block Size = 512).

## E Detection Performance of CoIn

We show the comparison of the two verifiers and the impact of  $\tau$  under different block sizes, as shown in Figure 7,8,9,10

## F Prompts Used in Discussion Section

Prompt 11 is used to explore the question “Can the original text be recovered from the tokens and embeddings exposed by COLA?”, while Prompt 12 is used to explore “How difficult is the dataset we constructed?”.

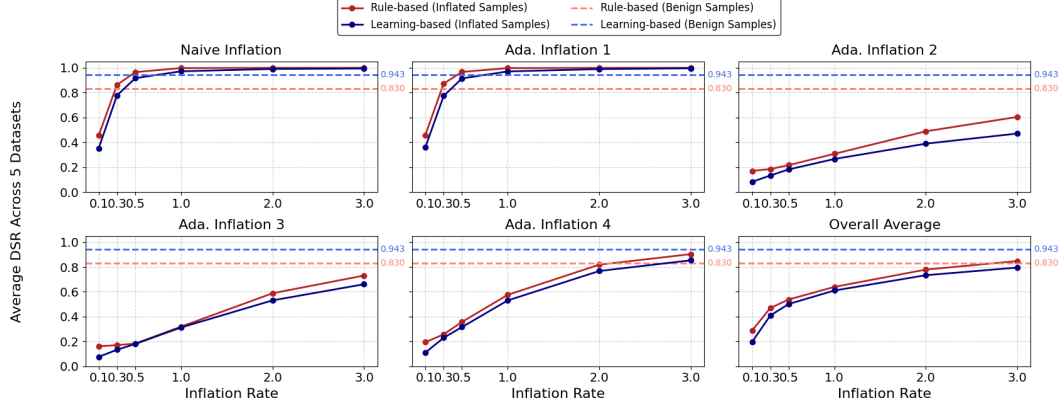


Figure 9: Performance of CoIn across different inflation methods and verifiers (Block Size = 1024). The red lines and the blue lines represent the DSR of rule-based verifier and learning-based verifier, respectively.  $\gamma$

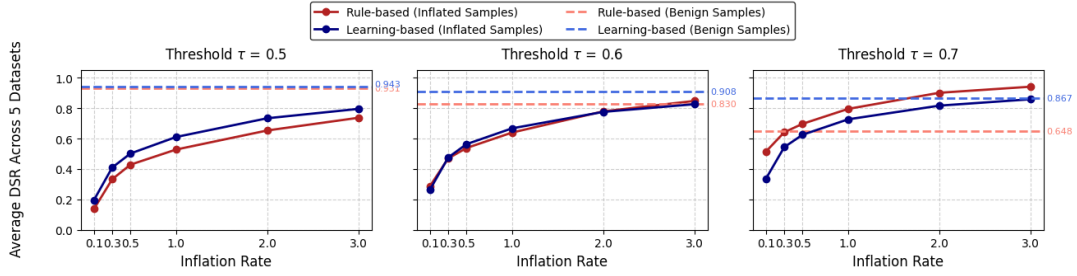


Figure 10: Impact of threshold  $\tau$  on DSR (Block Size = 1024).

### Prompt for Reconstructing Hidden Reasoning Passage

#### Reconstructing Hidden Reasoning Prompt

You are an expert in natural language reasoning and semantic retrieval.

Your task is to help recover a semantically meaningful and logically connected hidden passage that bridges a <Question> and an <Answer>.

This passage has been lost, but we know it is semantically related to both the <Question> and the <Answer>, and lies between them.

Given a – \*\*<Question>\*\*:  
{question}

And the – \*\*<Answer>\*\*:  
{answer}

We also know that some tokens from the original passage are still visible:  
{sampled\_token\_text}

And we retrieved related documents from Wikipedia using the embedding of the original passage:  
{retrieved\_rag\_docs}

Now, please help recover the most likely content of the hidden passage.  
Return your answer strictly in the following JSON format:

```
\\recovered_json {  
  "recovered_text": "<your reconstructed passage here>"  
}
```

Figure 11: Prompt for Recovering a Hidden Reasoning Passage Using Question, Answer, Token Clues and Retrieved Wikipedia Documents.

### Prompt for Evaluating Reasoning Passage Relevance

#### Evaluating Reasoning Process Prompt

You are a logical reasoning analyst.

Given a final answer and a randomly selected text passage, your task is to assess whether the text passage represents a reasoning process that leads to or supports the final answer.

The passage may or may not be relevant to the answer.

Your task is **not** to verify factual correctness, but to determine whether the passage semantically or logically connects to the answer and explains or justifies it in any meaningful way.

**\*\*Random Text Passage\*\*:**  
{reason}

**\*\*Final Answer\*\*:**  
{answer}

Please answer the following questions:

1. Is the text passage a plausible reasoning process that leads to the final answer?
2. Does it provide logical or semantic justification for the answer?

Respond in the following JSON format:

```
\\reasoning_assessment
{
  "is_reasoning_process": true/false,
  "justification": "<your brief explanation of why the passage is or isn't
a reasoning process for the answer>"
}
```

Figure 12: Prompt for Judging Whether a Block Supports or Explains a Final Answer.