TABARD: A Novel Benchmark for Tabular Anomaly Analysis, Reasoning and Detection

Anonymous ACL submission

Abstract

We study the capabilities of large language models (LLMs) in detecting fine-grained anomalies in tabular data. Specifically, we examine: (1) how well LLMs can identify diverse anomaly types—including factual, logical, temporal, and value-based errors; (2) the impact of prompt design and prompting strategies; and (3) the effect of table structure and anomaly type on detection accuracy. To this end, we introduce TABARD, a new benchmark constructed by perturbing tables from WikiTQ, FeTaQA, Spider, and BEAVER. The dataset spans multiple domains and eight anomaly categories, including paired clean and corrupted tables. We evaluate LLMs using direct, indirect, and Chain-of-Thought (CoT) prompting. Our results reveal notable limitations in standard prompting, especially for complex reasoning tasks and longer tables. To overcome these issues, we propose a unified framework combining multi-step prompting, self-verification, and constraint-based rule execution. Our approach significantly improves precision and recall, offering a promising direction for robust and interpretable anomaly detection in tables.

1 Introduction

Tables are a core data format across domains such as finance, healthcare, scientific research, and government reporting. They have gained increasing attention in machine learning (ML), natural language processing (NLP), and broader AI research, supporting tasks like question answering, table-to-text generation, schema understanding, and data integration. However, real-world tables is often noisy, incomplete, or inconsistent (Figure 1)—issues that can severely impact model performance as LLMs and analytics systems rely on such data. Even subtle anomalies can cascade into downstream errors, affecting outputs and decisions. Tables are especially prone to a variety of anomalies, including incorrect values, logical inconsistencies, temporal misalignments, arithmetic errors, and security flaws, all of which compromise data reliability.

As LLMs increasingly move to reasoning over tabular data (e.g., TableQA, fact verification), their reliability depends on the ability to detect subtle inconsistencies. However, anomaly detection in tables remains underexplored compared to text and vision, with limited benchmarks, tools, and systematic evaluation. Prior work has largely focused on unstructured or semi-structured data, with minimal attention to relational tables (Li et al., 2024). The absence of standardized benchmarks has contributed to the limited exploration of tabular anomaly detection, emphasizing the need for a systematic framework to evaluate model performance across diverse anomaly types.

To bridge this gap, we introduce TABARD, a comprehensive benchmark for evaluating anomaly detection in tabular data. TABARD is constructed by aggregating and perturbing tables from widely-used sources such as WikiTQ (Pasupat and Liang, 2015), FeTaQA (Nan et al., 2021), Spider (Yu, 2018), and BEAVER (Chen et al., 2024). TABARD features tables with systematically injected anomalies—including value, factual, logical, temporal, arithmetic, security, normalization, and consistency violations—while preserving clean counterparts to support supervised evaluation.

We evaluate large language models (LLMs) using a four-tier prompting framework—ranging from zero-shot to few-shot settings, with and without chain-of-thought (CoT) reasoning. This comprehensive analysis exposes key limitations in the models' ability to detect and explain anomalies in tabular data. To address these challenges, we propose three novel methods: (1) a multi-reasoning self-verification strategy that uses CoT to iteratively refine outputs; (2) a CoT-based approach enhanced with recursive attention prompting to improve contextual coherence and consistency; and (3) a neuro-symbolic, constraint-driven technique that trans-

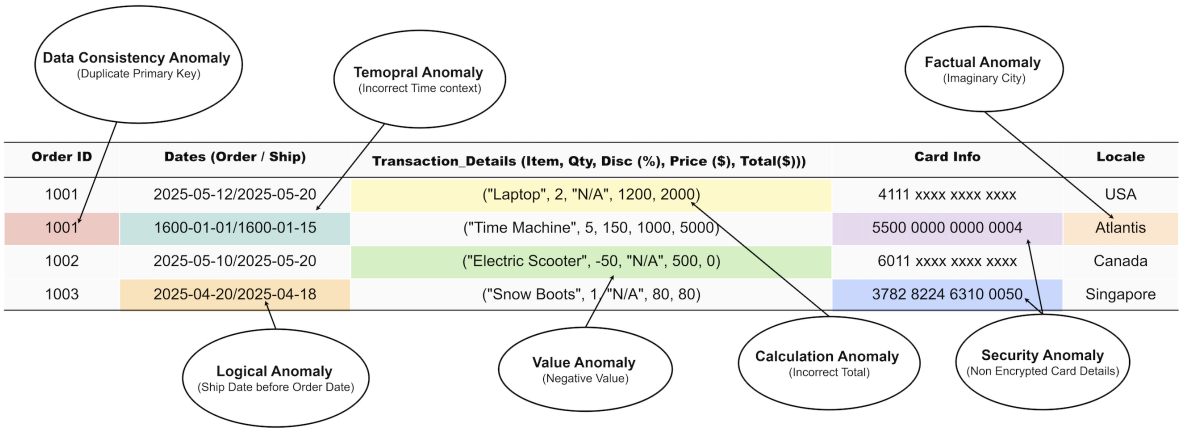

Figure 1: A comprehensive table containing instances of all anomaly types. Each arrow highlights an anomalous cell, annotated with the anomaly category and a brief explanation of its cause.

lates structured constraints into executable Python code, enabling robust factual validation through external knowledge integration. Our main contributions are as follows:

- We define the novel task of fine-grained anomaly detection in tabular data and propose a taxonomy of eight anomaly types: value, factual, logical, temporal, calculation, security, normalization, and consistency violations.
- We introduce TABARD, a human-verified benchmark constructed via controlled perturbations using large language models to simulate diverse anomaly types.
- We conduct a systematic evaluation of LLMs under various prompting strategies—zero-shot, few-shot, and Chain-of-Thought (CoT)—across different levels of prompt specificity.
- We propose three novel detection methods: MUSEVE, SEVCOT, and NSCM, a neuro-symbolic approach that converts LLM-generated constraints into executable Python code, improving anomaly coverage and detection accuracy.

Our benchmark and methods lay the groundwork for robust, interpretable LLM-based anomaly detection in structured data. We will release the dataset, code, and evaluation tools to support future research.

2 Table Anomaly Detection

Given a table $T = \{T_{i,j}\}$ with m rows and n columns, where each cell $T_{i,j}$ can take values from a mixed-type domain $\mathcal{D}$ (e.g., text, number, date), the goal is to learn a binary function $f(T_{i,j}) \in \{0, 1\}$ indicating whether the cell is anomalous. For each cell $T_{i,j}$, the model predicts a binary label $y_{i,j} \in \{0, 1\}$, where:

$$y_{i,j} = \begin{cases} 1, & \text{if } T_{i,j} \text{ is anomalous} \\ 0, & \text{otherwise} \end{cases}$$

This can be framed as a binary classification task over each cell $T_{i,j}$, where the model learns a function:

$$f : T_{i,j} \to \{0, 1\}$$

that maps each table cell to an anomaly label, optionally leveraging full table context, row/column metadata, and external knowledge. The objective is to maximize the recall of the anomaly detection function f by correctly identifying true anomalies while minimizing false negatives.

3 Our TABARD Dataset

We introduce TABARD, a benchmark for anomaly detection, analysis, and reasoning in tabular data, specifically designed to evaluate the capabilities of large language models (LLMs). TABARD is constructed by selecting and cleaning tables from four widely-used datasets: WikiTQ (Pasupat and Liang, 2015), FeTaQA (Nan et al., 2021), Spider (Yu, 2018), and BEAVER (Chen et al., 2024). The benchmark covers eight distinct types of anomalies:

- **Value Anomaly:** Deviation from valid cell-level values based on domain constraints.
- **Factual Anomaly:** Conflicts with established real-world knowledge.
- **Logical Anomaly:** Violations of logical relationships between columns in the same row.
- **Temporal Anomaly:** Inconsistencies or impossibilities in temporal data or sequences.

- **Calculation-Based Anomaly:** Errors in values derived from arithmetic computations.
- **Security Anomaly:** Leakage or improper handling of sensitive or restricted information.
- **Normalization Anomaly:** Structural issues that violate database normalization principles.
- **Data Consistency Anomaly:** Irregularities in formatting, structure, or representation across similar entries.

As illustrated in Figure 1, the benchmark includes diverse examples: a duplicate *Order ID* in the second row reflects a data consistency anomaly; the year *1600* in the *Date (Order/Ship)* column is temporally implausible; an *Order Date* occurring after the *Ship Date* indicates a logical anomaly. A negative quantity in the *Qty* column represents a value anomaly, while a miscalculated *Total* in the first row exemplifies a calculation-based anomaly. Exposure of unencrypted card details denotes a security anomaly, and assigning "Atlantis" as the *Locale* illustrates a factual anomaly.

TABARD **Generation** TABARD is constructed by perturbing original tables from multiple sources using large language models (LLMs) to introduce diverse types of anomalies as defined earlier. We use numeric and mixed-type tables from public datasets such as WikiTQ, Spider, BEAVER, and FeTaQA as our **table sources**. Tables are filtered based on column-level metadata—such as data type histograms, keyword matches, and composite-key heuristics—to guide downstream anomaly generation. We retain only those tables that contain at least a few numeric columns and have no missing cells.

Each selected table, along with its original schema, is passed to an LLM using specialized prompts designed to induce different anomaly types. We employ eight distinct perturbation prompts, each corresponding to a specific anomaly category. To generate ground truth labels, we track all cells modified by the LLM and label them as anomalous. Additionally, we store metadata including the anomaly type (i.e., the prompt used) and the LLM's explanation for the perturbation, which aids in subsequent data verification. Since TABARD preserves the original schema of the tables, all SQL and analytic queries valid on the clean version remain executable on its perturbed counterpart. This ensures structural robustness, with anomalies that survive downstream processes such as type casting and serialization.

TABARD **Verification.** We perform human verification to assess the validity of anomalies generated by the LLM (GPT-4o). A random 15% subset of the perturbed tables is sampled for manual review.

Sample	Cohen Kappa	Jaccard Coef.
α_1	94.48	92.67
α_2	94.23	93.56

Table 1: Human validation of LLM-generated anomalies based on agreement scores. Samples α_1 and α_2 correspond to 7.5% random, disjoint subsets of the TABARD dataset.

Two annotators evaluate the anomalies using both the perturbed table and its log (LLM-generated reasoning for the pertubed data), determining whether the injected anomaly is contextually justified. We report human-LLM agreement as shown in Table 1 scores as a measure of alignment between model-generated anomalies and human judgement. The high agreement scores indicate that the anomalies produced by the LLM are accurate and of high quality.

Dataset Statistics. The tables in TABARD are categorized into two types based on their length. Short tables are sourced from the WikiTQ and FeTaQA datasets and long tables are drawn from the Spider and BEAVER datasets which can be seen in Table 2. In each table, an average of $\lceil 0.5\,|\text{rows}| \rceil$ anomalies are injected (but varies a lot depending upon the table), with at least one anomaly per table. To evaluate robustness, a subset of tables is left unperturbed to test LLM performance on clean inputs. Notably, the standard deviation of row counts in long tables is significantly higher, reflecting a much broader variance in structure and complexity. This diversity enables robust evaluation of models

Statistic	Wiki+FeTa	Spi+BEA	Combined
Total tables	4,840	455	5,295
Avg	(17, 5)	(349, 7)	(33, 5)
Median	(13, 6)	(51, 7)	(14, 6)
Std Dev	(20, 1)	(2011, 6)	(443, 2)
Min	(2, 3)	(11, 2)	(2, 2)
Max	(479, 21)	(30000, 37)	(30000, 37)

Table 2: Dataset statistics for TABARD. For the metrics *Average*, *Median*, *Standard Deviation*, *Minimum*, and *Maximum*, values are reported in the format $(\lceil \text{rows} \rceil, \lceil \text{columns} \rceil)$, representing the per-table statistics for the number of rows and columns, respectively.

across all scales (small or large) table scenarios.

4 Modeling Approaches

We introduce three categories of methods (problem modeling approaches) with increasing complexity. The first category consists of Direct and Indirect prompts, each organized into four levels, with and without Chain-of-Thought reasoning. Then we have a multi-reasoning self-verification method that leverages CoT, a self-verifying CoT method augmented with recursive attention prompting technique, and a neuro symbolic constraint based method.

4.1 Direct and Indirect Prompts

We design four levels of prompts for both direct and indirect evaluation methods, where Levels 1 and 2 follow an indirect approach, while Levels 3 and 4 adopt a more direct evaluation strategy. As the prompt level increases, more task-specific information is provided, making the prompts progressively more fine-tuned and targeted.

1. **Just 'Problem' Mentioned** - "L1",(w/ and w/o CoT): There may be some problems present in the table, without mentioning anomalies or examples.

2. **Anomalies Mentioned** - "L2", (w/ and w/o CoT): This prompt replaces "problems" with the explicit term "anomalies", providing clearer task framing without examples.

3. **'X' type of Anomaly Mentioned** - "L3", (w/ and w/o CoT): Here prompt specifies the exact anomaly type (e.g., "factual anomaly", "value anomaly") while still omitting examples.

4. **'X' type of Anomaly with Example Mentioned** - "L4", (w/ and w/o CoT): these prompts enhance specificity further by including both the anomaly type and an illustrative few-shot example.

Note: From this point onward, we refer to these prompting configurations using the shorthand terminology: **L1–L4 (with and without CoT)**. This notation is consistently used in the results, figures, and discussions throughout the paper.
Each level is tested both with and without Chain-of-Thought (CoT) reasoning. In all configurations, anomalies are flagged using the `(index,column_name)` format. Then task is framed as binary classification—determining whether a given table cell contains an anomaly or not and compare it with the ground truth data.

4.2 Multi-Reasoning & Self-Verify Prompts

We have developed two new tailor made prompting techniques for this problem which includes many commonly used prompting techniques which are weaved in an intricate way to tackle this problem. MuSeVe (Multi-Reasoning Self Verification) and SeVCoT (Self Verification Chain of Thoughts) enhance anomaly detection performance. MuSeVe first employs self-consistency prompting by generating multiple independent reasoning paths, each using distinct logical frameworks to flag anomalous cells. Each path applies chain-of-thought (CoT) reasoning, performs self-verification, and outputs anomalies (index,column_name) format. A majority voting mechanism consolidates consistent flags, followed by a re-reading phase for final CoT-based refinement.

SeVCoT follows a similar structure but omits the self-consistency step. It performs a single CoT-based reasoning pass, followed by self-verification and a final re-reading stage. The key difference lies in the absence of multiple independent reasoning paths in SeVCoT. In both methods, we generate the Yes/No tables following the procedure described in the previous section and compare them with the ground truth Yes/No tables to compute the evaluation metrics. For all Direct and Indirect methods, as well as MuSeVe and SeVCoT, we apply dynamic chunking on long tables from the Spider and BEAVER datasets, since the large number of rows often causes the combined input and output tokens to exceed the model's context window.

4.3 Neuro-Symbolic Constraint Method

We propose NSCM a Neuro Symbolic Constraint based method deterministic data validation pipeline that leverages the generative capabilities of large language models (LLMs) while maintaining strict structural and logical control. A carefully engineered prompt, enriched with schema and representative values, is used to elicit a structured *validation dictionary* per table. This dictionary contains constraints across multiple categories such as domain, logical, temporal, calculation-based, etc. along with an `external_knowledge_validation` list for factual verification.

We begin by providing the schema and up to twenty unique cell values per column as input to

the LLM to generate constraints for the respective columns. So, let the schema used be S and the unique value set $U = \{U_1, U_2, \ldots, U_n\}$. Let.

$$V = \{C_1, C_2, \ldots, C_k\} \cup \{E_1, E_2, \ldots, E_j\},$$

where C_i are the intrinsic constraints and E_j is the set of external knowledge constraints. Both constraint sets may be empty, i.e.,

$$\{C_1, C_2, \ldots, C_k\} = \emptyset \quad \text{or}$$
$$\{E_1, E_2, \ldots, E_j\} = \emptyset$$

depending on the table structure and prompt output.

Let $e \in E_j$ denote a tuple extracted by the LLM using its internal knowledge and web-based retrieval capabilities. Here, e is defined as:

$$e = (\mathcal{X}, \mathcal{K}, y, \phi, k),$$

denoting context columns $\mathcal{X}$, knowledge columns $\mathcal{K}$, target column y, constraint function ϕ, and knowledge specification k.

For each unique combination $x \in \text{Domain}(\mathcal{X})$, we extract external knowledge $k_x \to \mathcal{K}(x)$, and augment the original dataset D to obtain:

$$D' = D \cup \{\mathcal{K}(x)\}.$$

Then, all constraints V are passed to a *statement generation agent*, which maps each constraint $\phi \in V$ to a Python `if` statement:

$$\phi : (r \in D') \mapsto \text{bool}.$$

Here, Φ is defined as a programmatic rule set:

$$\Phi = \{\phi_1, \phi_2, \ldots, \phi_m\},$$

executable over the data.

Each rule $\phi_i \in \Phi$ is embedded into a script $\mathcal{P}$, which is executed over the augmented dataset D' to yield the set of anomalies:

$$\mathcal{A} = \{(i, j) \mid \neg\phi_k(r_i),\ r_i[j] \text{ violates } \phi_k\}.$$

This unified execution framework handles both intrinsic and factual validations over the enriched data. Final outputs are grouped by constraint category and evaluated against ground truth labels G, allowing computation of precision, recall, and rule coverage. The pipeline can be summarized as:

$$(V, D, S, U) \xrightarrow{\text{LLM}} \Phi \xrightarrow{\mathcal{P}(D')} \mathcal{A}.$$

Illustrative Example: Temporal Constraint on `Dates` Column

We illustrate our constraint-based pipeline using the `Dates (Order / Ship)` column from Figure 1. The second row contains a temporal anomaly: `1600-01-01/1600-01-15`, which lies far outside a plausible e-commerce date range.

- Input Preparation: The table schema S and up to 20 representative values per column $U = \{U_1, \ldots, U_n\}$ are provided to the LLM.
- Constraint Generation: The LLM returns a validation dictionary:
  $$V = \{C_1, C_2, \ldots\} \cup \{E_1, E_2, \ldots\},$$
  where a temporal constraint C_i may state: *"Dates must lie in [2000, 2030] and order date $\leq$ ship date."*
- Code Synthesis: Each constraint $\phi \in V$ is translated into an executable `if` statement, forming a rule set Φ.
- Execution: The rules Φ are embedded in a script $\mathcal{P}$, run over the table D, and produce anomalies:
  $$\mathcal{A} = \{(1, \texttt{"Dates"})\}.$$
- Evaluation: Predicted anomalies $\mathcal{A}$ are compared to ground truth G to compute metrics such as precision and recall.

This example highlights how our approach enables LLM-generated constraints to be deterministically executed for anomaly detection, even in temporally structured data.

5 Experimental Evaluation

Through our experiments, we aim to investigate the following research questions: (a) How challenging is our benchmark dataset, TABARD, for current state-of-the-art models? (b) To what extent do various prompting strategies—such as few-shot learning, Chain-of-Thought (CoT), and Self Verification—improve anomaly detection performance? (c) How do different language models perform on the specific task of fine-grained anomaly detection in tables?

Evaluation: Our primary task is binary prediction, so we evaluate model performance using Precision (P) and Recall (R). F1 scores are also computed and reported in the appendix for completeness. All evaluations assume the current month and year to be May 2025.

	FeTaQA		Spi+BEA		WikiTQ		FeTaQA		Spi+BEA		WikiTQ	
Prompt	P	R	P	R	P	R	P	R	P	R	P	R
	ChatGPT-4o						**Gemini-1.5-Pro**					
	w/o CoT											
L1	38.5	55.7	43.8	43.3	50.1	52.6	29.2	56.9	45.3	46.5	31.0	58.6
L2	39.9	54.4	43.9	42.6	50.2	52.0	32.8	56.1	38.9	47.1	40.0	57.4
L3	44.2	54.5	44.0	39.5	50.0	51.4	38.3	54.5	37.4	47.4	45.4	55.1
L4	43.2	55.1	46.7	40.9	48.0	50.9	40.6	55.3	44.0	46.3	49.1	53.5
	with CoT											
L1	36.6	56.1	42.7	42.7	48.2	58.4	32.1	56.4	43.6	47.1	35.1	58.9
L2	40.0	56.5	44.5	42.5	49.9	55.7	35.3	56.6	41.7	47.3	42.2	54.8
L3	43.6	54.9	44.9	41.9	50.5	53.1	38.9	53.2	34.8	46.1	43.4	53.6
L4	44.1	56.6	46.4	43.1	52.3	55.2	37.1	56.0	41.4	46.7	41.5	54.2
MuSeVe	**48.9**	50.2	44.0	39.0	52.9	44.0	35.1	44.8	42.2	44.9	46.2	53.1
SeVCoT	42.8	51.0	**46.8**	39.9	**56.9**	52.5	28.5	55.4	39.5	47.0	41.9	56.1
NSCM	18.1	**63.6**	21.7	**52.6**	30.8	**66.6**	**39.3**	**71.3**	**52.4**	**59.6**	**51.2**	**60.0**

Table 3: This table highlights average Precision (P) and Recall (R) across various anomaly categories on the FeTaQA, Spider+BEAVER, and WikiTQ datasets, evaluated using four LLMs under different prompting strategies. $\mathbf{L_i}$ denotes the ith prompt level, with `-w/ocot` and `-wcot` indicating absence and presence of Chain-of-Thought reasoning, respectively. MuSeVe and SeVCoT represent multi-reasoning and self-verification variants.

Models. We evaluate four advanced language models: ChatGPT-4o, Gemini 1.5 Pro, LLaMA 3.1 70B Instruct, and DeepSeek-V3. Our constraint-based neuro-symbolic method is executed on ChatGPT-4o and Gemini 1.5 Pro. Due to limitations in external tool access, we were unable to deploy the constraint-based method on LLaMA and DeepSeek.

5.1 Findings: Results & Analysis

TABARD poses a significant **challenge**: Anomaly detection remains challenging across datasets, with both precision and recall falling short of expectations.

Do LLMs consistently benefit from Chain-of-Thought reasoning across prompt strategies? Across all prompt levels (L1–L4), adding Chain-of-Thought (CoT) reasoning consistently improves recall for ChatGPT-4o and Gemini-1.5-Pro. For example, on FeTaQA, ChatGPT-4o's recall increases from 55.7% (L1 w/o CoT) to 56.1% (L1 with CoT), and from 55.1% to 56.6% at L4. Gemini shows similar gains, with recall rising from 55.3% to 56.0% between L4 w/o CoT and with CoT (Table 3). While precision occasionally dips slightly with CoT (e.g., ChatGPT-4o L1 drops from 38.5% to 36.6%), the overall trend shows that CoT enhances reasoning depth, particularly for recall-sensitive tasks.

Do enhanced prompting techniques (MuSeVe, SeVCoT, NSCM) outperform standard CoT-based methods? Our enhanced prompting strategies demonstrate diverse strengths, as shown in Table 3. MuSeVe yields the highest precision on FeTaQA for ChatGPT-4o (48.9%), outperforming even the CoT-enhanced L4. SeVCoT leads in precision on Spider+BEA (46.8%) and WikiTQ (56.9%) for ChatGPT-4o. Meanwhile, NSCM dominates recall across all datasets and models. For instance, reaching 66.6% recall on WikiTQ for ChatGPT-4o and 71.3% on FeTaQA for Gemini. These results show that while CoT improves baseline prompting, our targeted strategies provide specialized improvements in either precision (MuSeVe, SeVCoT) or recall (NSCM).

Do certain prompting strategies work better on specific datasets? Performance trends differ across datasets. FeTaQA and WikiTQ, which contain shorter and more regular tables, show consistent gains from CoT and enhanced prompts. For example, NSCM reaches 66.6% recall on WikiTQ and 63.6% on FeTaQA with ChatGPT-4o, as shown in Table 3. In contrast, Spider+BEA—comprising longer and more heterogeneous tables—shows smaller gains from CoT and larger improvements from SeVCoT and NSCM. These results suggest that structured prompting is effective for simpler tables, while more complex tables require symbolic or constraint-based reasoning to achieve reliable performance.

Do prompting strategies generalize equally well across models and datasets? As shown in Table 3 Gemini-1.5-Pro consistently outperforms ChatGPT-4o in recall on complex datasets and CoT-enhanced prompts. For instance, under L4 with CoT, Gemini achieves 56.0% recall on FeTaQA, matching or exceeding ChatGPT-4o's 56.6%. On

	FeTaQA		Spi+BEA		WikiTQ		FeTaQA		Spi+BEA		WikiTQ	
Category	P	R	P	R	P	R	P	R	P	R	P	R
	ChatGPT-4o						**Gemini-1.5-Pro**					
	MuSeVe											
Calculation	41.4	41.6	46.3	30.3	76.0	63.0	33.6	35.6	60.5	50.5	90.0	90.0
Factual	40.9	37.9	43.7	37.0	42.0	30.0	25.8	28.3	35.1	41.9	32.0	36.0
Normalization	80.0	66.7	41.7	19.9	75.0	47.0	37.2	74.5	46.4	30.9	74.0	69.0
Logical	36.5	52.8	44.6	45.7	39.0	37.0	22.1	49.3	37.6	46.7	26.0	42.0
Temporal	51.5	51.3	25.4	31.1	40.0	42.0	51.1	57.3	21.9	41.2	29.0	56.0
Security	46.2	43.2	36.7	43.4	39.0	51.0	35.5	28.4	33.7	51.9	24.0	40.0
Consistency	42.5	29.2	50.8	33.6	48.0	29.0	28.9	16.8	43.3	29.0	39.0	32.0
Value	51.9	79.3	62.6	70.7	64.0	53.0	47.0	68.2	58.9	67.2	56.0	60.0
	SeVCoT											
Calculation	37.2	43.1	52.8	33.1	81.0	79.0	20.0	55.1	61.5	59.9	85.0	89.0
Factual	36.0	43.5	43.6	35.4	48.0	39.0	21.8	40.6	31.0	41.1	31.0	36.0
Normalization	69.7	69.3	45.9	20.4	79.0	58.0	36.2	72.5	40.8	31.4	61.0	76.0
Logical	29.2	56.5	49.4	47.7	38.0	43.0	17.0	54.5	31.5	47.7	24.0	47.0
Temporal	53.9	53.7	23.1	31.2	64.0	50.0	44.8	71.6	20.8	43.3	39.0	58.0
Security	38.2	36.2	39.7	45.6	32.0	42.0	28.4	39.5	35.7	56.9	18.0	39.0
Consistency	32.3	27.3	50.7	32.8	48.0	39.0	29.0	33.4	40.2	30.6	26.0	42.0
Value	46.2	78.4	65.2	72.7	65.0	70.0	31.2	76.0	54.7	64.9	51.0	62.0
	NSCM											
Calculation	10.3	57.9	6.5	22.9	35.3	67.3	30.0	70.6	55.1	72.4	75.6	85.6
Factual	10.0	51.5	21.5	29.6	12.3	46.1	20.7	36.9	38.4	30.5	31.1	20.3
Normalization	15.4	69.0	12.5	51.5	36.7	64.2	19.6	84.0	44.8	47.5	51.2	62.5
Logical	16.7	67.1	19.9	54.8	18.1	61.4	42.4	75.0	59.8	62.1	39.5	55.6
Temporal	9.4	33.0	21.4	50.8	9.9	49.0	74.6	94.6	25.1	46.7	18.2	21.6
Security	19.1	68.2	23.1	60.8	33.1	81.2	21.5	37.1	55.6	58.5	57.0	76.9
Consistency	23.4	54.8	33.2	50.3	26.7	53.4	36.2	42.3	58.4	51.5	32.8	29.7
Value	32.8	95.0	35.0	77.0	43.3	84.2	50.5	95.2	68.2	78.1	72.2	83.8

Table 4: This table summarizes the average precision (P) and Recall (R) achieved by ChatGPT-4o and Gemini-1.5-Pro across eight anomaly categories in the FeTaQA, Spider+BEAVER, and WikiTQ datasets., evaluated using four LLMs under different prompting strategies. $\mathbf{L_i}$ denotes the ith prompt level, with `-w/ocot` and `-wcot` indicating absence and presence of Chain-of-Thought reasoning, respectively. MuSeVe and SeVCoT represent multi-reasoning and self-verification variants.

WikiTQ, Gemini under NSCM reaches 60.0% recall versus ChatGPT-4o's 66.6%. While ChatGPT-4o shows more substantial precision under MuSeVe and SeVCoT, Gemini excels in leveraging deeper prompt levels for recall. Such results demonstrate that multi-step reasoning is processed differently across models, reinforcing the importance of architecture-aware prompt design.

Are LLMs equally effective across all anomaly categories? We examine performance trends across different anomaly types in Table 4. Value anomalies consistently yield the highest recall for ChatGPT-4o and Gemini-1.5-Pro, regardless of the prompting strategy. Under NSCM, ChatGPT-4o achieves a recall of 95.0% on FeTaQA and 84.2% on WikiTQ, while Gemini-1.5-Pro reaches 95.2% and 83.8%, respectively. These high scores suggest that value-based outliers are easier for models to identify, likely due to their more distinguishable statistical or formatting irregularities.

Are certain categories systematically more difficult? Table 4 also reveals that Factual and Temporal anomalies remain consistently challenging, for example, under MuSeVe, factual recall falls below 38% for ChatGPT-4o across all datasets, and under SeVCoT, Gemini's factual recall is limited to 36% on WikiTQ and 41.1% on Spider+BEA. These categories likely require external world knowledge or nuanced temporal interpretation, both known limitations for current LLMs.

Which methods and datasets show the strongest category-specific performance? As shown in Table 4,NSCM stands out in recall across many categories. For instance, on temporal anomalies, Gemini reaches 94.6% recall on FeTaQA under NSCM, while on logical anomalies it scores 75.0%—substantially higher than with MuSeVe or SeVCoT. Similarly, ChatGPT-4o achieves 81.2% recall on security in WikiTQ with NSCM, compared to 51.0% with MuSeVe or 42.0% with SeVCoT. These improvements show that incorporating constraints into prompts significantly improves the model's ability to reason methodically over structured inputs.

Do models behave differently depending on

the dataset characteristics? Model performance varies significantly with the structural properties of each dataset, as shown in Table 4. FeTaQA and WikiTQ, which consist of shorter and more uniformly structured tables, produce higher performance in most categories of anomalies. In contrast, Spider+BEA contains longer and more heterogeneous tables that challenge the models' ability to maintain contextual coherence. For example, in the logical category under SEVCOT, ChatGPT-4o achieves a recall of 56.5% on FeTaQA, but this drops to 47.7% on Spider+BEA. This pattern holds across categories and methods, highlighting the adverse effect of table length and schema complexity on anomaly detection accuracy.

Overall, the results from Table 3 and Table 4 reinforce the trends observed earlier: while CoT and enhanced prompting help, dataset characteristics such as table length and schema complexity strongly influence performance. Our constraint-based NSCM method demonstrates clear advantages in handling false negatives and identifying true Positives in these challenges, particularly when standard prompting strategies fail.

6 Related Works

To the best of our knowledge, this work is the first to investigate fine-grained, cell-level anomaly detection in tabular data using LLMs, extending prior anomaly detection efforts beyond unstructured and coarse-grained settings.

In the visual domain, vision-language models have improved anomaly detection in images and videos (Cao et al., 2023; Gu et al., 2023; Zhu et al., 2024; Yang et al., 2024). In multimodal settings, these models have also been employed to detect fake news by integrating textual and visual information (Jin et al., 2024; Liu et al., 2024). In the domain of log data, LLMs have been fine-tuned to identify anomalies in system logs, effectively handling both structured and unstructured formats (Han et al., 2023; Yamanaka et al., 2024; Lee et al., 2023; Hadadi et al., 2024).

Within tabular data, Li et al., 2024 demonstrated the potential of LLMs as zero-shot batch-level anomaly detectors and proposed synthetic data generation and fine-tuning strategies to enhance performance, achieving results on par with state-of-the-art methods. ANOLLM (Tsai et al., 2025) introduced a framework for unsupervised tabular anomaly detection that operates directly on raw textual features and showed competitive performance on several datasets.

Despite recent progress, fine-grained cell-level anomaly detection in tables remains underexplored. To address this, we introduce TABARD, a benchmark spanning eight anomaly types, and evaluate LLMs across diverse prompting strategies. We also propose three detection methods—including two prompting-based and one constraint-based approach—that outperform baselines and improve explainability in tabular anomaly detection.

7 Conclusion and Future Work

This work introduces TABARD, a comprehensive benchmark and evaluation framework for fine-grained anomaly detection in tabular data using large language models (LLMs). By perturbing real-world tables with diverse anomaly types and evaluating multiple prompting strategies—including multi-reasoning, self-verification, and a novel neuro-symbolic constraint-based method—the study reveals fundamental strengths and limitations of current LLMs. While traditional prompting with CoT enhances recall, especially on simpler datasets, our advanced methods like MUSEVE, SEVCOT, and NSCM achieve substantial improvements in precision, recall, and explainability across more complex settings. TABARD establishes a principled foundation for future research on robust, interpretable, and scalable anomaly detection systems in structured data.

We identify several promising directions for future research. We aim to detect mixed and task-driven anomalies, address incomplete or missing table data, and extend anomaly detection to multimodal tables with visual elements. Additionally, we plan to strengthen our constraint-based framework by incorporating feedback-guided refinement and meta-level modeling for more reliable constraint generation. We provide a detailed discussion of these directions in the Appendix Section-B.

Limitations

While our work makes significant strides in fine-grained, cell-level anomaly detection in tabular data using LLMs, it has certain limitations. First, our benchmark and methods are restricted to anomalies within individual tables. We do not address multi-table settings, such as anomalies arising from relationships across multiple tables, inconsistencies between linked tables, or anoma-

lies requiring inter-table reasoning, which are common in many real-world tabular datasets. Furthermore, our work assumes access to full and complete tables, whereas real-world tabular data often contains missing or incomplete entries, which can significantly affect anomaly detection performance. Additionally, our study primarily focuses on prompting strategies without exploring fine-tuning LLMs specifically for tabular anomaly detection, which presents an opportunity for further enhancing model performance.

Also, the higher false positive rate in our constraint-based method primarily stems from the conservative design of the constraint generation phase. The LLM, given limited contextual information (i.e., only the schema and up to 20 unique values), may over-generalize certain validation rules. This can lead to constraints that are syntactically or statistically reasonable, but overly strict when applied to edge-case values in the full dataset. For example, a domain constraint inferred from a partial value distribution may reject rare-but-valid entries, flagging them incorrectly as anomalous. Additionally, logical or calculation-based constraints derived from sparse patterns may fail to capture all valid relational variants, further contributing to false positives. To address this issue, we have outlined two potential approaches for future work.

Ethics Statement

This research complies with the ethical guidelines of the Association for Computational Linguistics (ACL) and EMNLP. Our benchmark dataset, TABARD, is constructed by perturbing publicly available, anonymized tabular datasets (WikiTQ, FeTaQA, Spider, and BEAVER) intended for academic use. No personally identifiable information (PII) or sensitive user data is included, ensuring no direct privacy risks.

All anomalies were synthetically generated using controlled prompts to large language models (LLMs), simulating realistic yet artificial data inconsistencies across domains such as finance and education. Sensitive-looking values (e.g., credit card numbers) are entirely fabricated. TABARD contains no human subject data, no private information, and no harm-inducing content. It is built exclusively from publicly available sources and is designed to be harmless and ethically sound. A subset of the data was verified by independent annotators to ensure contextual correctness; annotators were compensated fairly and exposed to no sensitive content. The authors have done human annotation with detailed instruction given.

To mitigate risks associated with automation and misuse, our methods are fully transparent and reproducible. TABARD is released with a comprehensive datasheet specifying its use strictly for research and benchmarking. We discourage its use for training deployed systems without additional safeguards. Our neuro-symbolic method further enhances interpretability by translating LLM-generated constraints into executable rules.

Additionally, AI assistance—including language models—was used to support parts of the paper writing and research process (e.g., prompt development and explanation formatting). All outputs were carefully reviewed by human authors.

In conclusion, we prioritize data privacy, model transparency, annotator welfare, and responsible AI use in all aspects of this work.

References

Yunkang Cao, Xiaohao Xu, Chen Sun, Xiaonan Huang, and Weiming Shen. 2023. Towards generic anomaly detection and understanding: Large-scale visual-linguistic model (gpt-4v) takes the lead.

Peter Baile Chen, Fabian Wenz, Yi Zhang, Devin Yang, Justin Choi, Nesime Tatbul, Michael Cafarella, Çağatay Demiralp, and Michael Stonebraker. 2024. Beaver: an enterprise benchmark for text-to-sql. *arXiv preprint arXiv:2409.02038*.

Zhaopeng Gu, Bingke Zhu, Guibo Zhu, Yingying Chen, Ming Tang, and Jinqiao Wang. 2023. Anomalygpt: Detecting industrial anomalies using large vision-language models. *arXiv preprint arXiv:2308.15366*.

Fatemeh Hadadi, Qinghua Xu, Domenico Bianculli, and Lionel Briand. 2024. Anomaly detection on unstable logs with gpt models. *arXiv preprint arXiv:2406.07467*.

Xiao Han, Shuhan Yuan, and Mohamed Trabelsi. 2023. Loggpt: Log anomaly detection via gpt. In *2023 IEEE International Conference on Big Data (BigData)*, pages 1117–1122. IEEE.

Ruihan Jin, Ruibo Fu, Zhengqi Wen, Shuai Zhang, Yukun Liu, and Jianhua Tao. 2024. Fake news detection and manipulation reasoning via large vision-language models. *arXiv preprint arXiv:2407.02042*.

Yukyung Lee, Jina Kim, and Pilsung Kang. 2023. Lanobert: System log anomaly detection based on bert masked language model. *Applied Soft Computing*, 146:110689.

Aodong Li, Yunhan Zhao, Chen Qiu, Marius Kloft, Padhraic Smyth, Maja Rudolph, and Stephan Mandt. 2024. Anomaly detection of tabular data using llms. *arXiv preprint arXiv:2406.16308*.

Xuannan Liu, Peipei Li, Huaibo Huang, Zekun Li, Xing Cui, Jiahao Liang, Lixiong Qin, Weihong Deng, and Zhaofeng He. 2024. Fka-owl: Advancing multimodal fake news detection through knowledge-augmented lvlms. In *Proceedings of the 32nd ACM International Conference on Multimedia*, pages 10154–10163.

Linyong Nan, Chiachun Hsieh, Ziming Mao, Xi Victoria Lin, Neha Verma, Rui Zhang, Wojciech Kryściński, Nick Schoelkopf, Riley Kong, Xiangru Tang, Murori Mutuma, Ben Rosand, Isabel Trindade, Renusree Bandaru, Jacob Cunningham, Caiming Xiong, and Dragomir Radev. 2021. Fetaqa: Free-form table question answering.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables.

Che-Ping Tsai, Ganyu Teng, Phil Wallis, and Wei Ding. 2025. Anollm: Large language models for tabular anomaly detection.

Yuuki Yamanaka, Tomokatsu Takahashi, Takuya Minami, and Yoshiaki Nakajima. 2024. Logelectra: Self-supervised anomaly detection for unstructured logs. *arXiv preprint arXiv:2402.10397*.

Yuchen Yang, Kwonjoon Lee, Behzad Dariush, Yinzhi Cao, and Shao-Yuan Lo. 2024. Follow the rules: reasoning for video anomaly detection with large language models. In *European Conference on Computer Vision*, pages 304–322. Springer.

Tao et al. Yu. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Jiaqi Zhu, Shaofeng Cai, Fang Deng, Beng Chin Ooi, and Junran Wu. 2024. Do llms understand visual anomalies? uncovering llm's capabilities in zero-shot anomaly detection. In *Proceedings of the 32nd ACM International Conference on Multimedia*, pages 48–57.

Appendix

A Data Generation and Preprocessing

We initiated our data generation process using ChatGPT-4o with a temperature setting of 0.7 and a maximum token limit of 1000. To ensure flexibility across varying table lengths and structures, we applied dynamic chunking to segment tables contextually.

Our dataset construction is based on aggregating and perturbing tables from four established benchmarks: WikiTQ (Pasupat and Liang, 2015), FeTaQA (Nan et al., 2021), Spider (Yu, 2018), and BEAVER (Chen et al., 2024). This aggregated resource, which we refer to as TABARD, introduces diverse types of anomalies while retaining realistic table semantics.

Perturbations were applied at the cell level to simulate factual, logical, temporal, and consistency-based errors. To enable precise anomaly localization during evaluation, we inserted a marker token (‘@@@_‘) at the beginning of each modified cell. This annotation allows systematic tracking of anomalous content during both training and inference.

For each table, a binary yes/no table is generated using the `(index, column_name)` pairs predicted by the LLMs. This table mirrors the structure and schema of the original table, but each cell is replaced with either `"Yes"` (indicating an anomaly) or `"No"` (indicating normal data). During the data generation phase, anomalies are introduced by prepending a unique identifier token (`@@@_`) to the corresponding cells. Using this information, ground truth yes/no tables are constructed in the same format. The predicted and ground truth tables are then compared to compute evaluation metrics including Precision, Recall, and F1 score. The counts for true positives, false positives, and false negatives are determined as follows:

```
if prediction[prediction_key] == "Yes" and
   label[label_key] == "Yes":
    true_positives += 1
elif prediction[prediction_key] == "Yes" and
     label[label_key] == "No":
    false_positives += 1
elif prediction[prediction_key] == "No" and
     label[label_key] == "Yes":
    false_negatives += 1
```

Overall, this pipeline supports the generation of fine-grained anomaly instances aligned with real-world scenarios over structured data.

***Note**: Anomalies are often misunderstood as outright errors, but this is not always the case. An anomaly refers to a data point that is rare, misleading, or inconsistent with the rest of the dataset—not necessarily incorrect. Its interpretation can be subjective and context-dependent. For instance, if most dates in a column follow the* `MM/DD/YYYY` *format but one entry uses* `DD/MM/YYYY`*, the value may still be valid, yet inconsistent with the column's format. This illustrates a data consistency anomaly, where the deviation disrupts structural uniformity rather than factual correctness.*

B Future Works

Our study opens several promising directions for future research. One avenue is exploring mixed anomalies, where a single cell may contain multiple anomaly types simultaneously (e.g., factual and logical errors), as well as scenarios where multiple types of anomalies appear across different cells within the same table. Another important extension is handling incomplete tables with missing or empty cells. Future work could investigate task-driven anomaly detection, where anomalies are identified indirectly through the performance of downstream tasks, such as answering questions or summarizing. Detecting performance drops or inconsistencies in such tasks when operating on corrupted versus clean tables could offer practical, interpretable signals of anomalies. We have also planned to make some future advancements in our NSCM to address some of our methods limitations which includes:

a) **Pre-Trained Meta Model**: Use feedback from prior executions to evaluate each constraint's precision and recall on held-out data, ranking them by F1 to identify weak patterns. This information can then guide a lightweight meta-model or prompt-tuning setup that learns to prefer high-quality constraint patterns (e.g., robust range checks over brittle list matches), allowing the LLM to progressively refine its generation toward more reliable and dataset-aligned rules.

b) **Feedback-Guided Refinement**: Use historical execution feedback (e.g., common false-positive patterns or constraint violation frequencies across clean data) to iteratively fine-tune the constraint generation prompt. This enables the system to learn which constraint types are too aggressive and adapt generation accordingly.

Finally, extending anomaly detection to multimodal tables that include visual elements—such

as images embedded in cells—presents an exciting direction, requiring models to jointly reason over both textual and visual information.

C Prompts Used in Experiments

We present the prompt formulations used during experimentation. We designed each prompt to evaluate the performance of the model in various anomaly detection settings.

C.1 Just 'Problem' Mentioned- L1 Prompts

L1-w CoT

Here is the JSON data: {json_string} which might have some problems in its cells.
Analyze the data and identify anomalies in the data. Follow a structured step-by-step Chain-of-Thought (CoT) approach before returning the final output. Identify and return them in the format [(index, column_name), (index, column_name)] where *index* corresponds to the index in the list and *column_name* is the name of the column you think there is a problem. Just generate the list format output so I can easily parse it.

L1-w/o CoT

Here is the JSON data: {json_string} which might have some problems. Identify and return them in the format `[(index, column_name), (index, column_name)]` where `index` corresponds to the index in the list and `column_name` is the name of the column you think there is a problem. Just generate the list format output so I can easily parse it.

C.2 Anomalies Mentioned- L2 Prompts

L2-w/o CoT

Here is the JSON data: {json_string}. Can you identify the anomalous cells and return them in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the index in the list and `column_name` is the name of the column you think there is an anomaly in. Just generate the list format output so I can easily parse it.

L2-w CoT

Here is the JSON data: {json_string}
Task:
Analyze the data and *identify anomalies* in the data. Follow a structured *step-by-step Chain-of-Thought (CoT) approach* before returning the final output.

Step 1: Understand the Data Structure
1. Parse the JSON and read the table very carefully to find anomalies in it.

Step 2: Generate the Anomalous Cells
- Identify the anomalous cells with anomalies present in them.
- Return the output in the format `[(index, column_name), (index, column_name)]` where `index` corresponds to the index in the list and `column_name` is the name of the column you think there is an anomaly. Just generate the list format output so I can easily parse it.

C.3 'X' type of Anomaly Mentioned- L3 Prompts

L3-w/o CoT

Here is the JSON data: {json_string}. Can you identify the cells with {anomaly} anomalies and return them in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the index in the list and `column_name` is the name of the column you think there is a {anomaly} anomaly. Just generate the list format output so I can easily parse it.

L3-w CoT

Here is the JSON data: {json_string}
Task:
Analyze the data and *identify {anomaly} anomalies*. Follow a structured *step-by-step Chain-of-Thought (CoT) approach* before returning the final output.

Step 1: Understand the Data Structure
1. Parse the JSON and *identify the key fields* relevant to {anomaly} anomalies.

Step 2: Generate the Anomalous Cells
- Identify the anomalous cells with {anomaly} anomaly present in them.
- Return the output in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the index in the list and `column_name` is the name of the column you think contains a {anomaly} anomaly. Just generate the list format output so I can easily parse it.

C.4 'X' type of Anomaly with Example Mentioned - L4 Prompts

L4-w CoT (Data Consistency)

Here is the JSON data: {json_string}
Task:
Analyze the data and *identify data consistency anomalies*. Follow a structured *step-by-step Chain-of-Thought (CoT) approach* before returning the final output.

Step 1: Understand the Data Structure
1. Parse the JSON and *identify the key fields* relevant to data consistency anomalies.

Step 2: Find Out the Anomalies Present in the Table
For each record in the dataset, check for anomalies. Below are examples of data consistency anomalies:
1. Inconsistent Formats: A `"phone number"` column where values use different formats (e.g., `+1-234-567-8901`, `(123) 456-7890`, or unformatted like `1234567890`).
2. Mismatched Categories: Different naming conventions used for the same category (e.g., `"HR"`, `"Human Resources"`, and `"H.R."`).
3. Cross-Table Inconsistencies: Salary or payment values that differ between related tables (e.g., `expected_pay` vs. `exact_pay`).

Step 3: Generate the Anomalous Cells
Return the output in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the index in the list and `column_name` is the name of the column where a data consistency anomaly is present. Just generate the list format output so I can easily parse it.

L4-w/o CoT (Data Consistency)

Here is the JSON data: {json_string}. Analyze the data and identify the cells with data consistency anomalies and return them in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the index in the list and `column_name` is the name of the column you think contains a data consistency anomaly. Just generate the list format output so I can easily parse it.
Here are a few examples of data consistency anomalies to help you identify them:
1. Inconsistent Formats: A `"phone number"` column where values use different formats (e.g., `+1-234-567-8901`, `(123) 456-7890`, or unformatted like `1234567890`).
2. Mismatched Categories: Different naming conventions used for the same category (e.g., `"HR"`, `"Human Resources"`, and `"H.R."`).
3. Cross-Table Inconsistencies: Salary or payment values that differ between related tables (e.g., `expected_pay` vs. `exact_pay`).

L4-w CoT (Security)

Here is the JSON data: {json_string}
Task:
Analyze the data and *identify security anomalies*. Follow a structured *step-by-step Chain-of-Thought (CoT) approach* before returning the final output.

Step 1: Understand the Data Structure
1. Parse the JSON and *identify the key fields* relevant to security anomalies.

Step 2: Find Out the Anomalies Present in the Table
For each record in the dataset, check for anomalies. Below are examples of security

anomalies:
1. **Suspended Role Conflict**: A user marked as `"inactive"` still holds a `"manager"` role, posing a security risk if the account is used maliciously.
2. **Missing Audit Logs**: Failed login attempts with blank fields for `"Browser Version"` or `"User-Agent"`.
3. **Suspicious Activity**: A user logs in from a different country every time, whereas they normally log in from a single region or office location.

Step 3: Generate the Anomalous Cells
Return the output in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the index in the list and `column_name` is the name of the column where a security anomaly is present. Just generate the list format output so I can easily parse it.

L4-w/o CoT (Security)

Here is the JSON data: {json_string}. Analyze the data and identify the cells with security anomalies and return them in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the index in the list and `column_name` is the name of the column you think contains a security anomaly. Just generate the list format output so I can easily parse it.
Here are a few examples of security anomalies to help you identify them:
1. **Suspended Role Conflict**: A user marked as `"inactive"` still holds a `"manager"` role, posing a security risk if the account is used maliciously.
2. **Missing Audit Logs**: Failed login attempts with blank fields for `"Browser Version"` or `"User-Agent"`.
3. **Suspicious Activity**: A user logs in from a different country every time, whereas they normally log in from a single region or office location.

L4-w CoT (Value)

Here is the JSON data: {json_string}
Task:
Analyze the data and *identify value anomalies*. Follow a structured *step-by-step Chain-of-Thought (CoT) approach* before returning the final output.

Step 1: Understand the Data Structure
1. Parse the JSON and *identify the key fields* relevant to value anomalies.

Step 2: Find Out the Anomalies Present in the Table
For each record in the dataset, check for anomalies. Below are examples of value anomalies:

- **Missing or Null Values:**
  - A critical column like `"Product ID"` or `"Customer Name"` containing empty or null values where such information is required.
  - A numeric field like `"Price"` or `"Quantity"` left blank in a sales record.
- **Illogical Negative Values:**
  - A `"Price"` column with negative values, which is not possible for most products.
  - A `"Salary"` field showing a negative amount for an employee.
- **Extreme Outlier Values:**
  - A house listed with a price of $1 or $1 billion in a neighborhood where the typical range is $300,000 to $500,000.
  - A recorded temperature of 150°C in a location where temperatures do not exceed 50°C.

Step 3: Generate the Anomalous Cells
Return the output in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the index in the list and `column_name`

is the name of the column where a value anomaly is present. Just generate the list format output so I can easily parse it.

L4-w/o CoT (Value)

Here is the JSON data: {json_string}. Analyze the data and identify the cells with value anomalies and return them in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the index in the list and `column_name` is the name of the column you think contains a value anomaly. Just generate the list format output so I can easily parse it.
Here are a few examples of value anomalies to help you identify them:

- **Missing or Null Values:**
  - A critical column like `"Product ID"` or `"Customer Name"` containing empty or null values where such information is required.
  - A numeric field like `"Price"` or `"Quantity"` left blank in a sales record.
- **Illogical Negative Values:**
  - A `"Price"` column with negative values, which is not possible for most products.
  - A `"Salary"` field showing a negative amount for an employee.
- **Extreme Outlier Values:**
  - A house listed with a price of $1 or $1 billion in a neighborhood where the typical range is $300,000 to $500,000.
  - A recorded temperature of 150°C in a location where temperatures do not exceed 50°C.

L4-w CoT (Normalization)

Here is the JSON data: {json_string}
Task:
Analyze the data and *identify normalization anomalies*. Follow a structured *step-by-step Chain-of-Thought (CoT) approach* before returning the final output.

Step 1: Understand the Data Structure
1. Parse the JSON and *identify the key fields* relevant to normalization anomalies.

Step 2: Find Out the Anomalies Present in the Table
For each record in the dataset, check for anomalies. Below are examples of normalization anomalies:

- **Partial Dependencies (2NF Violation):**
  In an `"Orders"` table with `OrderID` and `ProductID` as a composite key, some rows may include `CustomerName` depending only on `CustomerID`, violating 2NF.
- **Transitive Dependencies (3NF Violation):**
  In an `"Employees"` table, `OfficeLocation` depends on `Department`, which depends on `EmployeeID`, causing a transitive dependency.
- **Denormalization:**
  A column like `TotalSalary` stores the sum of `BaseSalary` and `Bonus` directly, potentially leading to inconsistencies.
- **Combined Attributes:**
  A single field storing values like `"West Bengal, India, 721306"` instead of separating into City, Country, and Zip.
- **Repeating Groups (1NF Violation):**
  A `"Skills"` column containing `"MongoDB, C++, C"` instead of being split into individual entries.

Step 3: Generate the Anomalous Cells
Return the output in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the index in the list and `column_name` is

the name of the column where a normalization anomaly is present. Just generate the list format output so I can easily parse it.

L4-w/o CoT (Normalization)

Here is the JSON data: {json_string}. Analyze the data and identify the cells with normalization anomalies and return them in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the index in the list and `column_name` is the name of the column you think contains a normalization anomaly. Just generate the list format output so I can easily parse it.
Here are a few examples of normalization anomalies to help you identify them:

- **Partial Dependencies (2NF Violation):**
  In an `"Orders"` table with `OrderID` and `ProductID` as a composite key, some rows may include `CustomerName` depending only on `CustomerID`, violating 2NF.
- **Transitive Dependencies (3NF Violation):**
  In an `"Employees"` table, `OfficeLocation` depends on `Department`, which depends on `EmployeeID`, causing a transitive dependency.
- **Denormalization:**
  A column like `TotalSalary` stores the sum of `BaseSalary` and `Bonus` directly, potentially leading to inconsistencies.
- **Combined Attributes:**
  A single field storing values like `"West Bengal, India, 721306"` instead of separating into City, Country, and Zip.
- **Repeating Groups (1NF Violation):**
  A `"Skills"` column containing `"MongoDB, C++, C"` instead of being split into individual entries.

L4-w CoT (Factual)

Here is the JSON data: {json_string}
Task:
Analyze the data and *identify factual anomalies*. Follow a structured *step-by-step Chain-of-Thought (CoT) approach* before returning the final output.

Step 1: Understand the Data Structure
1. Parse the JSON and *identify the key fields* relevant to factual anomalies.

Step 2: Find Out the Anomalies Present in the Table
For each record in the dataset, check for anomalies. Examples include:

- **Contradictions:**
  - An `"assistant"` with a higher salary than a `"senior manager"`.
  - A `"student"` listed with a professional title like `"Professor"`.
- **Unrealistic Values:**
  - A product price of $1 for a high-end smartphone or $10,000 for a notebook.
  - A building height of 5000 meters for a residential structure.
- **Geographical Mismatches:**
  - A city listed as `"New York"` with a postal code for Los Angeles.
  - A 50°C temperature recorded in the Arctic.
- **Ambiguities:**
  - A currency field with `"Dollar"` but no specification (USD or CAD).
  - A date like `"5/7/22"` that could mean May 7 or July 5.
- **Record-Breaking Claims:**
  - A 100-meter race time of 8 seconds.
  - A business reporting 200% profit margins.

- **Unlikely Proportions:**
  - 95% expenses relative to revenue where 70% is the norm.
  - A household using 100,000 kWh in one month.
- **Other Factual Anomalies:**
  - Any data that contradicts realistic expectations based on context or domain.

Step 3: Generate the Anomalous Cells
Return the output in the format `[(index, column_name), (index, column_name)]`, where `index` refers to the list index and `column_name` is the name of the column you believe contains a factual anomaly. Just generate the list format output so I can easily parse it.

L4-w/o CoT (Factual)

Here is the JSON data: {json_string}. Analyze the data and identify the cells with factual anomalies and return them in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the index in the list and `column_name` is the name of the column you think contains a factual anomaly. Just generate the list format output so I can easily parse it.
Here are a few examples of factual anomalies to help you identify them:

- **Contradictions:**
  - An `"assistant"` with a higher salary than a `"senior manager"`.
  - A `"student"` listed with a professional title like `"Professor"`.
- **Unrealistic Values:**
  - A product price of $1 for a high-end smartphone or $10,000 for a notebook.
  - A building height of 5000 meters for a residential structure.
- **Geographical Mismatches:**
  - A city listed as `"New York"` with a postal code for Los Angeles.
  - A 50°C temperature recorded in the Arctic.
- **Ambiguities:**
  - A currency field with `"Dollar"` but no specification (USD or CAD).
  - A date like `"5/7/22"` that could mean May 7 or July 5.
- **Record-Breaking Claims:**
  - A 100-meter race time of 8 seconds.
  - A business reporting 200% profit margins.
- **Unlikely Proportions:**
  - 95% expenses relative to revenue where 70% is the norm.
  - A household using 100,000 kWh in one month.
- **Other Factual Anomalies:**
  - Any data that contradicts realistic expectations based on context or domain.

L4-w CoT (Temporal)

Here is the JSON data: {json_string}
Task:
Analyze the data and *identify temporal anomalies*. Follow a structured *step-by-step Chain-of-Thought (CoT) approach* before returning the final output.

Step 1: Understand the Data Structure
1. Parse the JSON and *identify the key fields* relevant to temporal anomalies.

Step 2: Find Out the Anomalies Present in the Table
For each record, examine the following types of temporal anomalies:

- **Conflicting Schedules:**
  - A meeting that starts before the

previous one has ended.
  - A task scheduled to begin while its prerequisite is still ongoing.

- **Illogical Durations:**
  - A marathon listed as lasting 1 second.
  - A 24-hour flight for a 2-hour route.

- **Chronological Inconsistencies:**
  - A `departure_time` earlier than the `arrival_time`.
  - An `event_end_time` before the `event_start_time`.

- **Unrealistic Temporal Outliers:**
  - An international delivery marked as complete 30 seconds after ordering.
  - A task completed in negative time (e.g., -2 minutes).

- **Timezone Discrepancies:**
  - Misaligned start and end times due to missing or incorrect time zones.
  - An event listed at 9:00 AM in one timezone but 10:00 AM in another.

- **Invalid Temporal Sequences:**
  - A work shift ending before it begins.
  - A follow-up call scheduled before the initial consultation.

Step 3: Generate the Anomalous Cells
Return the output in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the list index and `column_name` is the name of the column with a temporal anomaly. Just generate the list format output so I can easily parse it.

L4-w/o CoT (Temporal)

Here is the JSON data: {json_string}. Analyze the data and identify the cells with temporal anomalies and return them in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the index in the list and `column_name` is the name of the column with the temporal anomaly. Just generate the list format output so I can easily parse it.
Here are a few examples of temporal anomalies to help you identify them:

- **Conflicting Schedules:**
  - Meetings that overlap or start before the previous one ends.
  - Tasks starting while prerequisite tasks are still in progress.

- **Illogical Durations:**
  - A 1-second marathon.
  - A 24-hour flight on a 2-hour route.

- **Chronological Inconsistencies:**
  - `departure_time` earlier than `arrival_time`.
  - `event_end_time` before `event_start_time`.

- **Unrealistic Temporal Outliers:**
  - A delivery completed within 30 seconds of an international order.
  - Tasks marked complete in negative time.

- **Timezone Discrepancies:**
  - Event times misaligned across timezones.
  - A webinar incorrectly converted between zones (e.g., 9:00 AM shown as 10:00 AM).

- **Invalid Temporal Sequences:**
  - A shift that ends before it starts.
  - A follow-up event occurring before the initial one.

L4-w CoT (Calculation)

Here is the JSON data: {json_string}
Task:
Analyze the data and *identify calculation anomalies*. Follow a structured *step-by-step Chain-of-Thought (CoT) approach* before returning the final output.

Step 1: Understand the Data Structure
1. Parse the JSON and *identify the key fields* relevant to calculation anomalies.

Step 2: Find Out the Anomalies Present in the Table
For each record in the dataset, check for the following types of calculation anomalies:

- **Incorrect Totals:**
  A `"grand total"` field that does not match the sum of individual transaction amounts.
- **Incorrect Formula:**
  Use of inaccurate or logically invalid formulas (e.g., calculating body fat percentage as `waist circumference / height`).
- **Missing Dependencies:**
  A `"discounted price"` column referring to missing or undefined `"original price"` values.
- **Logical Violations:**
  Calculations yielding implausible results (e.g., an age of 200 or a negative quantity in inventory).
- **Rounding Errors:**
  Inconsistent rounding across financial fields, such as tax being rounded inconsistently across entries.

Step 3: Generate the Anomalous Cells
Return the output in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the index in the list and `column_name` is the name of the column where a calculation anomaly is found. Just generate the list format output so I can easily parse it.

L4-w/o CoT (Calculation)

Here is the JSON data: {json_string}. Analyze the data and identify the cells with calculation anomalies and return them in the format `[(index, column_name), (index, column_name)]`, where `index` corresponds to the index in the list and `column_name` is the name of the column you think contains a calculation anomaly. Just generate the list format output so I can easily parse it.
Here are a few examples of calculation anomalies to help you identify them:

- **Incorrect Totals:**
  A `"grand total"` field that does not reflect the actual sum of transaction components.
- **Incorrect Formula:**
  Use of an incorrect expression, such as body fat percentage `= waist circumference / height`.
- **Missing Dependencies:**
  Fields like `"discounted price"` that rely on missing data in `"original price"`.
- **Logical Violations:**
  Results that are outside a logical range (e.g., negative age or quantity).
- **Rounding Errors:**
  Financial fields rounded inconsistently, leading to discrepancies in totals.

L4-w CoT (Logical)

Here is the JSON data: {json_string}
Task:
Analyze the data and *identify logical anomalies*. Follow a structured *step-by-step Chain-of-Thought (CoT) approach* before returning the final output.

Step 1: Understand the Data Structure
1. Parse the JSON and *identify the key fields* relevant to logical anomalies.

Step 2: Find Out the Anomalies Present in the Table
For each record, check for the following types of logical inconsistencies:

- **Illogical Temporal Relationships:**
  - `registration_date` occurs after `termination_date`.
  - `delivery_date` occurs before `order_date`.
- **Biological or Physical Impossibilities:**
  - An age value of 180 years or more.
  - A speed exceeding 5000 km/h in standard transport records.
- **Inconsistent Financial Data:**
  - A discount greater than the `total_price`.
  - A `refund_amount` exceeding the original transaction.
- **Categorical Misclassifications:**
  - `"CEO"` as a job title for a person with salary below minimum wage.
  - `"Reptile"` listed in a mammal classification.
- **Anachronisms or Technological Impossibilities:**
  - A device `manufacture_date` predating its invention (e.g., smartphone from 1990).
  - A `passport_issued_date` before the `birth_date`.
- **Referential Inconsistencies:**
  - `customer_id` not found in the customer registry.
  - A `state` field with a nonexistent or misspelled geographic entity.

Step 3: Generate the Anomalous Cells
Return the output in the format `[(index, column_name), (index, column_name)]`, where `index` is the record index and `column_name` is the column containing a logical anomaly. Just generate the list format output so I can easily parse it.

L4-w/o CoT (Logical)

Here is the JSON data: {json_string}. Analyze the data and identify the cells with logical anomalies and return them in the format `[(index, column_name), (index, column_name)]`, where `index` is the index in the list and `column_name` is the column with the logical anomaly. Just generate the list format output so I can easily parse it.
Here are a few examples of logical anomalies to help you identify them:

- **Illogical Temporal Relationships:**
  - `registration_date` occurs after `termination_date`.
  - `delivery_date` occurs before `order_date`.
- **Biological or Physical Impossibilities:**
  - An age value of 180 years or more.
  - A speed value exceeding 5000 km/h in ground transport.
- **Inconsistent Financial Data:**
  - A discount greater than the `total_price`.
  - A refund larger than the transaction amount.
- **Categorical Misclassifications:**
  - A `"CEO"` job title with an implausibly low salary.
  - A mammal labeled as a `"Reptile"`.
- **Anachronisms or Technological Impossibilities:**
  - Devices with manufacture dates before their invention.
  - A passport issued before the person's birth.

- **Referential Inconsistencies:**
  - Invalid `customer_ids` not found in master data.
  - Misspelled or nonexistent states/regions in the `state` column.

C.5 MuSeVe

MuSeVe

`prompt = f"{{json_string}}"` will have some anomalies in its cells.

Task: Structured Anomaly Detection in Semi-Structured Tables

You are an advanced anomaly detection system trained to analyze semi-structured tables. Your goal is to *detect anomalies at the cell level* using a structured *step-by-step approach* that ensures high accuracy, logical consistency, and explainability.

Step 1: Generate Multiple Independent Reasoning Paths (Self-Consistency Prompting)

- Perform *multiple independent analyses* of the table, each using a unique reasoning approach.
- Ensure that *each reasoning path is completely independent* and may use different logical frameworks.
- Each reasoning path should *only flag specific table cells as anomalies*, not entire rows or columns.

Step 2: Apply Chain-of-Thought (CoT) for Each Reasoning Path

For each independent reasoning path, use *step-by-step logical reasoning*:

1. Identify patterns in the data.
2. Compare against expected norms (historical data, rules, domain-specific expectations).
3. Detect outliers or logical inconsistencies in individual cells.
4. Analyze cross-field relationships (e.g., role vs. status, date vs. event).
5. Conclude whether the flagged cells are anomalous or valid.

Step 3: Anomaly Detection at the Cell Level

- Flag only specific table cells that contain anomalies.
- Use a structured output format `(index, column_name)` where:
  - `index` corresponds to the index in the list.
  - `column_name` corresponds to the specific field with the anomaly.

Step 4: Self-Verification for Each Anomaly (True/False Check)

Each reasoning path must verify its own flagged anomalies:

1. Ask yourself: "Is this anomaly truly incorrect?"
2. Cross-check with:
   - Expected value distributions.
   - Logical consistency.
   - Historical data references.
3. Final Decision:
   - **True** → The flagged cell is a confirmed anomaly.
   - **False** → The flagged cell is valid and should not be marked.

Step 5: Majority Voting

- Collect flagged anomalies from all reasoning paths.
- If more than 70% of reasoning paths agree that a flagged cell is anomalous, confirm the anomaly.

- These majority-voted anomalies will be the final anomalies reported.

Step 6: Re-Reading & Final Chain-of-Thought Verification
Before finalizing results:

1. Re-read the table one last time.
2. Apply a final structured CoT reasoning process.
3. Ensure no valid cells are incorrectly flagged.
4. Make any necessary corrections to the anomaly list.

Final Output: A refined, validated anomaly list.

Final Output Format
Return *only* the structured list of confirmed anomalies. Use the format: `[(index, column_name), (index, column_name)]`, where `index` refers to the index in the list and `column_name` is the column with the anomaly.

C.6 SeVCoT

SeVCoT

Here is the JSON data: {json_string}
Task:
Analyze the data and identify anomalies. Follow a structured *step-by-step Chain-of-Thought (CoT) approach* before returning the final output.

Step 1: Understand the Data Structure
1. Parse the JSON and identify the key fields relevant to .

Step 2: Find Out the Anomalies Present in the Table
For each record in the dataset, check for anomalies. Examples of anomalies include:

- **Suspended Role Conflict:** A user marked as `"inactive"` still holds a `"manager"` role, posing a risk if the account is used maliciously.
- **Missing Audit Logs:** Failed login attempts with blank fields for `"Browser Version"` or `"User-Agent"`.
- **Suspicious Activity:** A user logs in from a different country every time, whereas they normally log in from a single region or office location.

Step 3: Find the Anomalous Cells
Now, find the anomalous cells where you believe a anomaly is present. Note down all such cells.

Step 4: Self Verification (True/False Check)
Verify the flagged anomalies:

1. Ask yourself: "Is the flagged cell truly an anomaly?"
2. Cross-check with:
   - Expected value distributions
   - Logical consistency
   - Historical data references
3. Final Decision:
   - **True** – The flagged cell is a confirmed anomaly.
   - **False** – The flagged cell is actually valid and should not be marked.
4. Retain only **True** confirmed anomalies for the final output.

Step 5: Re-reading and Final CoT Checking
Before finalizing, re-read the table one last time and apply structured CoT reasoning:

- Scan the flagged anomalies again.
- Ensure no valid cells are incorrectly flagged.
- Make corrections to the list if necessary.

Final Output: A refined, validated anomaly list.

Step 6: Final Output Generation
Return the final output in the format: `[(index, column_name), (index, column_name)]`, where `index` refers to the list index and `column_name` is the column with a anomaly.
Only output the list in this format for easy parsing.

C.7 NSCM

C.7.1 Constraint Prompt - NSCM

Constraint Prompt

You are an expert in data validation. You will be provided with a JSON table where each column includes a subset of its unique values, some of which may be anomalies. You will also be given some examples of the same table so that you can understand any inter-column dependencies. Your task is to analyze these unique values in combination with the column names and generate generic, domain-informed constraints to exclude anomalous values.
Critically important: You must not create set-based constraints for numeric columns. For example, do not write constraints like `if Year not in [1980, 1990, 2020]` — such constraints are strictly forbidden for numeric data. Instead, derive range-based or pattern-based rules based on plausible domain logic.
For instance, if the values for Year are [2001, 2002, 1980, 19890, 1900, 1500, 2026], then values like 1500, 2026, and 19890 are likely anomalies. A correct constraint would be:

```
if not (1900 <= Year <= 2025)
```

not a list-based check on specific values.
You must assume the provided unique values are examples, not the exhaustive set. Do not assume the list of values is complete or definitive.
Constraints must follow these principles:

- Be generalizable and not tied to specific data points.
- Avoid `value in [..]` checks unless absolutely necessary.
- For non-numeric columns, set-based constraints may be used sparingly, and only after filtering out `null`, `None`, or empty string values.
- Always prioritize data type checks, logical bounds, format patterns, or value consistency rules over literal matches.
- Never include contradictory or logically inconsistent rules.
- Think carefully and analytically — your goal is to identify what should be valid, not just what has been seen.

These are examples of constraints you should consider under each constraint type:
1. Domain Constraints

- *Uniqueness:* `transaction_id` must be unique.
- *Code and Identifier Standards:* `country_code` must match ISO-3166-1 alpha-2 format.
- *Range Constraints:* `age` must be between 0 and 120.
- *Length and Format Constraints:* `email` must match standard email format.
- *Valid Categorical Values:* `athletic_event` must be from {`100m, 200m, 400m`}.

2. Logical Constraints

- *Cross-Column Logic:* `start_date` $\leq$ `end_date`.
- *Mathematical Relationships:* `total_price = unit_price * quantity`.
- *Value Interdependencies:* `100m_time < 200m_time` for same athlete.

- *Valid Ranges by Context:* `score_percent` must be between 0–100.

3. Temporal Constraints

- *Realistic Time Values:* `registration_date` must not be in the future.
- *Sequence Validations:* `login_time < logout_time`.
- *Unrealistic Time Gaps:* Manual entries should not occur in the same millisecond.

4. External Knowledge Constraints

- *Real-World Limits:* `world_record_100m` must not be < 9.5 seconds.
- *Contextual Validations:* `population` should not be 0 for an official country.

5. Security Constraints

- *Sensitive Data Protection:* `credit_card_number` must be encrypted or masked.
- *Injection Checks:* `username`, `comment` fields must be free of SQL/script tags.

6. Inter-Row Constraints

- *Temporal Consistency Across Rows:* `subscription start_date` must follow previous `end_date`.
- *Grouped Aggregates:* `balance = previous + deposits withdrawals`.

7. Data Consistency Constraints

- Detect anomalies in formats.
  Example: `Year = [1980, 1999, 2000, 2001, Nineteen-hundred-and-twenty-two, 2023]`
  Expected pattern: `YYYY`, Anomaly: `Nineteen-hundred-and-twenty-two`

All constraints should be written as:

`x = y + z`, `x < y`, `x != None`, `unique(x)`, regex, etc.
Avoid data type constraints.

Step 2 – Identify Factual Constraints for Validation
Identify all columns with factual info and validate using external knowledge. For each:

- `column_requiring_validation`
- `context_columns`
- `external_knowledge_columns`
- `external_knowledge_constraint`
- `required_external_knowledge`

Final JSON Output Structure:

```
{
  "domain_constraints": ["constraint_1", ...],
  "logical_constraints": ["constraint_1", ...],
  "temporal_constraints": ["constraint_1", ...],
  "calculation_based_constraints": ["constraint_1", ...],
  "security_constraints": ["constraint_1", ...],
  "inter_row_constraints": ["constraint_1", ...],
  "other_constraints": ["constraint_1", ...],
  "external_knowledge_validation": [
    {
      "context_columns": ["context_column_1",..],
      "external_knowledge_columns": ["column_name_1.."],
    "column_requiring_validation": "column name from original data",
    "external_knowledge_constraint": "logical or mathematical rule",
      "required_external_knowledge": "name of external fact"
    }
  ]
}
```

C.7.2 Statement Generation Prompt

Statement Generation Prompt

I have a list of strings that describe validation rules for a JSON object. I want you to convert each rule into a Python `if` statement that validates a corresponding key in a dictionary called `data`. The output should be a Python list of `if` statements in string format, such as:

```
["if not isinstance(data.get('Rating'), float):", "if not re.match(...):", ...]
```

Here are the rules (each line is one rule):

```
["constraints"]
```

Furthermore, you will also be given the JSON object representing the table schema

and the unique values each column has, some of which might be anomalies. From the combination of the column name and the common values, you need to figure out the anomalous values and write the `if` statements. You need to use the rules to validate the data in the JSON object. The output should be a dictionary containing all violations found. The dictionary should have the following structure:

Notes:

- Use `re.match()` for regex-based validations.
- For dates, assume that `import datetime` is already included.
- Treat `integer` as `int` in Python.
- Uniqueness checks should be mentioned as comments (or optionally include sample logic to track uniqueness across multiple records).
- The output should be a Python list of stringified `if` statements.
- You may assume `import re` is already included.
- For comparisons and calculations involving numbers, if the value is a string type, parse the numbers from the string for comparison.
- Constraints should not be complicated. They should **not** be followed by `try-except` blocks or error raising.

The `if` statements should be simple and straightforward, like:

- `"if not re.match(...):"`
- `"if data['column_name'] < 0:"`
- `"if data['column_name'] > 100:"`
- `"if data['column_name'] != None:"`
- `"if data['column_name_1'] != data['column_name_2'] + data['column_name_3']:"`

Please return only the list of `if` statements as described.

Output schema:

```
{
    "if_statements": [
      "if not isinstance(data.get('column_name'), data_type):",
        "if not re.match(...):",
        ... more if statements if applicable ...
    ]
}
```

D Prompt-Level Evaluation and Overlap Analysis

This section presents extended experimental results and analyses that complement the findings in the main paper. These analyses aim to deepen our understanding of model choice, and anomaly category characteristics interact in complex anomaly category tasks. Specifically, we include:

1. **Category-wise Overlap Analysis:** We compute the percentage overlap between anomaly categories within each dataset (FeTaQA, Spider+BEA, and WikiTQ), showing how frequently instances belong to multiple categories. This reveals inter-category dependencies that highlight the compositional nature of anomalies and motivates the design of prompt-level evaluation.

2. **Prompt-Level Model Performance:** We report precision and recall scores for multiple large language models (ChatGPT-4o, Gemini-1.5-Pro, LLaMA-3.1-70B-Instruct, and Deepseek-V3) across a range of prompting strategies. These include four baseline prompt levels (L1–L4), each reflecting increasing depth of reasoning, evaluated both with and without Chain-of-Thought (CoT) reasoning. We additionally evaluate two advanced prompting variants: MUSEVE, a multi-reasoning formulation, and SEVCOT, which incorporates self-verification mechanisms along with CoT. The results show that model performance varies not only across datasets and categories but also based on prompt formulation, emphasizing the critical role of prompt engineering in anomaly detection tasks.

These extended analyses provide a deeper view of model behavior and highlight the importance of structured prompting in anomaly detection. All prompts are included in Section C for reproducibility.

D.1 Category Overlap Analysis.

In Table 5, We observe high inter-category overlap among factual, logical, consistency, and value anomalies—especially in FeTaQA and Spider+BEA—indicating that these anomalies often co-occur and may be semantically entangled. For instance, in Spider+BEA, over 87% of consistency instances also align with logical and value anomalies, while in FeTaQA, logical anomalies have nearly 100% overlap with factual ones. In contrast, normalization and calculation anomalies show more isolation, particularly in WikiTQ, where overlap with other types remains below 20%. These trends highlight that certain anomaly types are inherently multi-faceted, requiring models to disentangle overlapping error signals to achieve accurate detection.

D.2 Trend Analysis (LLaMA-3.1-70B-Instruct vs. Deepseek-V3).

In Table 6, across all datasets, Deepseek-V3 consistently outperforms LLaMA-3.1-70B-Instruct in both precision and recall, especially in lower prompt levels and without CoT. CoT reasoning provides marginal recall improvements for LLaMA but yields clearer gains for Deepseek, particularly on FeTaQA and WikiTQ. Among prompt strategies, performance peaks around L3–L4 with CoT, while SEVCOT shows stronger overall recall compared to MUSEVE. These trends suggest Deepseek is better at leveraging prompt complexity and CoT, whereas LLaMA benefits less from reasoning scaffolds.

D.3 Prompt-Level Trend Analysis

At L-4 in Table 7, both ChatGPT-4o and Gemini-1.5-Pro show consistent improvements in recall when Chain-of-Thought (CoT) reasoning is applied, particularly for value, temporal, and logical anomalies. CoT also boosts precision in several cases, especially for normalization and calculation in FeTaQA and WikiTQ. However, factual and security anomalies remain difficult across both models, with only marginal gains even under CoT. Gemini generally demonstrates stronger recall in complex reasoning categories, while ChatGPT-4o slightly edges ahead in security and consistency precision. These results suggest that CoT helps deepen model reasoning, but its effectiveness varies significantly by anomaly type.

Extending this observation to earlier prompt levels, Table 8 shows that prompting strategies at L2 and L3 outperform L1 in both precision and recall across datasets. These gains are more pronounced when paired with CoT, confirming that step-by-step prompting scaffolds model reasoning effectively. Categories such as value, calculation, and normalization benefit the most, especially for Gemini-1.5-Pro, which consistently shows stronger performance under enriched prompts. Conversely, anomaly types involving implicit semantics—such as factual and security—see little improvement even with deeper prompt formulations. This suggests that while CoT strengthens structural reasoning, it alone is insufficient for addressing context-heavy or knowledge-dependent errors.

The same trend continues in Table 9, where Deepseek-V3 clearly outperforms LLaMA-3.1-70B-Instruct across prompt levels and datasets. Deepseek achieves particularly strong recall in calculation and value anomalies—reaching up to 90.0% recall at L-3 with CoT—demonstrating its superior scalability with prompt complexity. LLaMA, by contrast, exhibits weaker improvements from deeper prompts and CoT scaffolding, often plateauing in recall regardless of prompt design. Notably, both models still struggle with factual and security errors, revealing persistent challenges in capturing external knowledge and complex policy violations.

This pattern holds at L-4 as well (Table 10), where Deepseek continues to outperform LLaMA in most categories, particularly under the SEVCOT and MUSEVE strategies. CoT leads to moderate recall improvements for logical and temporal categories in both models, but Deepseek again benefits more robustly. Even so, low recall on factual and consistency anomalies persists, indicating that current models—even with rich prompts and structured verification—struggle with semantically subtle or context-heavy reasoning tasks.

In conclusion, across all prompt levels and strategies, we find that CoT reasoning consistently enhances model performance in structurally clear anomaly types (e.g., value, calculation, normalization), particularly when paired with deeper prompt formulations. Deepseek-V3 demonstrates stronger generalization and better use of multi-step reasoning compared to LLaMA and even state-of-the-art APIs like ChatGPT-4o and Gemini-1.5-Pro in many settings. However, all models exhibit persistent challenges in handling fact-based or policy-driven

FeTaQA								
Category(%)	**Calculation**	**Consistency**	**Factual**	**Logical**	**Normalization**	**Security**	**Temporal**	**Value**
Calculation	100.0	28.9	29.7	29.7	25.7	19.4	11.1	29.6
Consistency	28.9	100.0	98.8	98.8	8.5	56.6	54.4	98.6
Factual	29.7	98.8	100.0	100.0	8.7	56.4	54.7	99.7
Logical	29.7	98.8	100.0	100.0	8.7	56.4	54.7	99.7
Normalization	25.7	8.5	8.7	8.7	100.0	5.7	0.0	8.7
Security	19.4	56.6	56.4	56.4	5.7	100.0	45.2	56.2
Temporal	11.1	54.4	54.7	54.7	0.0	45.2	100.0	54.5
Value	29.6	98.6	99.7	99.7	8.7	56.2	54.5	100.0
Spi+BEA								
Category(%)	**Calculation**	**Consistency**	**Factual**	**Logical**	**Normalization**	**Security**	**Temporal**	**Value**
Calculation	100.0	80.6	67.8	71.0	38.2	69.8	31.6	76.9
Consistency	80.6	100.0	76.2	87.3	39.7	88.9	38.1	95.4
Factual	67.8	76.2	100.0	75.0	27.1	68.3	39.6	75.4
Logical	71.0	87.3	75.0	100.0	34.4	79.4	30.6	86.2
Normalization	38.2	39.7	27.1	34.4	100.0	43.1	50.0	40.0
Security	69.8	88.9	68.3	79.4	43.1	100.0	39.0	87.7
Temporal	31.6	38.1	39.6	30.6	50.0	39.0	100.0	38.5
Value	76.9	95.4	75.4	86.2	40.0	87.7	38.5	100.0
WikiTQ								
Category(%)	**Calculation**	**Consistency**	**Factual**	**Logical**	**Normalization**	**Security**	**Temporal**	**Value**
Calculation	100.0	15.9	15.8	15.8	89.5	15.2	0.0	7.7
Consistency	15.9	100.0	95.7	95.7	17.5	59.8	42.8	47.0
Factual	15.8	95.7	100.0	98.8	17.6	58.5	40.6	48.5
Logical	15.8	95.7	98.8	100.0	17.6	58.5	41.5	48.5
Normalization	89.5	17.5	17.6	17.6	100.0	15.8	1.1	8.6
Security	15.2	59.8	58.5	58.5	15.8	100.0	31.5	28.8
Temporal	0.0	42.8	40.6	41.5	1.1	31.5	100.0	20.1
Value	7.7	47.0	48.5	48.5	8.6	28.8	20.1	100.0

Table 5: Category-wise percentage overlap across the FeTaQA, Spider+BEA, and WikiTQ datasets. Each cell reports the proportion of instances in a given source category (row) that also appear in a target category (column), revealing the extent of semantic and structural co-occurrence among anomaly types. This supplementary analysis provides insights into category interdependence relevant to the anomaly detection methodology discussed in the main text.

	FeTaQA		Spi+BEA		WikiTQ		FeTaQA		Spi+BEA		WikiTQ	
Prompt	P	R	P	R	P	R	P	R	P	R	P	R
	LLaMA-3.1-70B-Instruct						**Deepseek-V3**					
	w/o CoT											
L1	26.2	41.5	23.9	28.5	40.8	22.9	42.0	56.6	49.7	43.6	47.0	50.9
L2	26.0	40.4	23.2	28.3	42.8	21.8	44.2	55.5	50.0	42.6	49.9	48.8
L3	28.4	39.8	24.2	28.6	45.1	15.5	47.5	51.3	50.1	40.6	50.6	48.2
L4	27.0	44.0	23.3	27.7	41.4	22.1	45.9	53.5	49.2	42.3	50.6	47.8
	with CoT											
L1	24.8	42.5	23.6	28.7	34.2	24.5	41.2	56.5	50.2	42.3	49.2	52.1
L2	24.6	41.9	24.4	29.9	36.9	27.2	43.2	54.0	51.8	42.3	53.2	48.4
L3	27.9	41.5	24.4	29.4	37.5	24.4	45.0	50.5	51.1	41.4	56.5	45.2
L4	27.2	44.1	24.1	30.4	36.4	24.5	47.5	53.6	52.5	42.0	53.4	46.0
MuSeVe	23.7	38.9	22.0	26.8	35.6	21.4	42.3	52.4	47.9	40.6	49.0	44.5
SeVCoT	24.6	37.9	23.3	27.5	43.9	27.9	41.0	51.1	50.6	40.6	52.6	48.0

Table 6: This table highlights average Precision (P) and Recall (R) across the FeTaQA, Spider+BEA, and WikiTQ datasets for LLaMA-3.1-70B-Instruct and Deepseek-V3 under various prompting strategies. Prompt levels L_i correspond to increasing levels of reasoning complexity, evaluated both with (`-wcot`) and without (`-w/ocot`) Chain-of-Thought (CoT) reasoning. MuSeVe and SeVCoT denote multi-reasoning and self-verification prompting variants, respectively. This table summarizes model performance trends across datasets and reasoning strategies in the context of anomaly detection.

	FeTaQA		Spi+BEA		WikiTQ		FeTaQA		Spi+BEA		WikiTQ	
Category	P	R	P	R	P	R	P	R	P	R	P	R
	ChatGPT-4o						**Gemini-1.5-Pro**					
	L-4-w/o CoT											
Calculation	41.5	50.6	54.9	34.8	69.2	71.3	40.2	50.2	66.0	40.0	90.0	90.0
Factual	33.6	41.9	42.8	33.0	38.5	34.9	27.8	41.5	39.5	39.0	37.0	34.0
Normalization	57.5	68.0	48.7	30.2	54.3	67.2	58.5	74.5	41.0	43.8	65.0	75.0
Logical	27.7	54.3	47.4	45.7	34.0	42.4	26.3	55.8	42.9	46.5	22.0	36.0
Temporal	70.1	67.7	24.3	31.8	60.1	51.4	71.4	67.4	28.9	42.4	64.0	57.0
Security	34.2	30.6	38.1	38.9	27.9	30.0	33.9	32.7	27.0	50.4	21.0	31.0
Consistency	39.0	41.0	52.2	40.2	45.0	40.5	31.7	40.4	49.8	39.1	32.0	40.0
Value	41.9	86.9	65.0	72.3	55.3	69.3	35.4	79.6	56.9	68.8	62.0	65.0
	L-4-w CoT											
Calculation	40.0	50.9	57.6	35.6	75.0	84.0	37.7	52.1	67.8	47.2	85.0	90.0
Factual	31.9	41.4	43.5	35.8	41.0	41.0	26.7	42.1	35.7	41.1	35.0	36.0
Normalization	65.3	73.9	46.2	32.6	55.0	66.0	42.5	76.5	38.2	45.5	40.0	70.0
Logical	28.5	59.0	44.0	49.1	41.0	44.0	24.2	53.4	44.3	37.3	26.0	36.0
Temporal	66.4	65.8	25.0	33.7	65.0	56.0	70.1	70.4	25.9	40.8	48.0	56.0
Security	40.2	30.6	39.6	42.7	38.0	31.0	33.3	32.9	27.6	53.3	20.0	35.0
Consistency	39.7	43.6	52.9	40.8	46.2	46.6	27.3	40.3	38.9	37.2	29.0	43.0
Value	40.9	87.6	62.5	74.5	57.0	73.0	35.1	80.1	53.0	70.9	49.0	68.0

Table 7: This table summarizes Average Precision (P) and Recall (R) across eight anomaly categories in the FeTaQA, Spider+BEA, and WikiTQ datasets, evaluated using ChatGPT-4o and Gemini-1.5-Pro under various prompting strategies. The MuSeVe and SeVCoT configurations represent multi-reasoning and self-verification prompting variants.

anomalies, suggesting the need for external knowledge integration.

	FeTaQA		Spi+BEA		WikiTQ		FeTaQA		Spi+BEA		WikiTQ	
Category	P	R	P	R	P	R	P	R	P	R	P	R
	ChatGPT-4o						**Gemini-1.5-Pro**					
	L-1-w/o CoT											
Calculation	25.5	48.3	46.7	36.4	75.9	78.3	18.8	56.2	70.1	62.2	76.0	92.0
Factual	32.7	49.7	42.8	39.8	42.0	39.6	19.1	39.2	36.7	41.3	14.0	39.0
Normalization	59.6	69.3	43.6	22.7	70.1	56.3	48.3	75.8	54.6	24.8	52.0	75.0
Logical	25.6	59.3	44.3	50.1	29.4	41.9	20.1	58.6	39.0	48.4	13.0	47.0
Temporal	49.9	61.2	26.1	35.2	44.2	57.8	49.6	76.8	28.1	44.2	17.0	65.0
Security	38.0	40.1	35.5	49.8	27.0	42.6	24.8	37.7	33.7	50.8	13.0	47.0
Consistency	36.5	34.9	53.1	38.2	49.0	35.8	24.9	34.1	46.7	32.6	26.0	39.0
Value	39.8	82.6	58.2	74.6	62.9	68.4	27.7	77.0	53.6	67.9	37.0	65.0
	L-1-w CoT											
Calculation	27.6	50.9	46.7	35.7	81.0	83.0	21.4	51.3	62.9	61.4	84.0	92.0
Factual	31.8	49.6	42.0	39.2	33.0	42.0	20.9	42.9	35.9	41.3	17.0	39.0
Normalization	55.0	68.6	45.0	22.8	67.0	61.0	60.4	71.9	49.3	27.7	62.0	74.0
Logical	25.1	61.6	41.3	49.9	32.0	49.0	18.9	60.7	39.4	48.9	15.0	50.0
Temporal	46.1	60.1	25.0	35.6	41.0	60.0	52.3	74.1	24.8	42.3	20.0	60.0
Security	34.7	41.6	33.4	46.9	23.0	57.0	27.4	41.2	36.1	53.2	15.0	51.0
Consistency	34.2	34.4	49.9	37.0	49.9	42.8	24.9	34.1	45.5	33.5	25.0	41.0
Value	38.3	82.2	58.1	74.5	59.0	72.0	30.5	75.2	54.6	68.5	43.0	64.0
	L-2-w/o CoT											
Calculation	30.8	51.7	49.7	37.5	74.1	73.3	21.7	52.1	45.1	57.7	84.0	92.0
Factual	33.4	49.1	43.3	38.7	43.1	41.3	20.6	40.7	40.9	42.4	20.0	35.0
Normalization	60.3	59.5	42.6	22.5	71.2	56.0	60.4	79.7	45.9	29.1	65.0	75.0
Logical	26.7	61.0	45.6	49.0	30.6	45.5	20.8	56.1	31.4	48.5	21.0	52.0
Temporal	53.9	61.7	25.4	34.7	44.2	57.8	50.4	73.2	22.9	43.0	28.0	59.0
Security	37.3	38.8	36.1	47.0	26.9	40.2	27.0	37.2	32.3	55.6	18.0	46.0
Consistency	36.6	33.3	50.5	37.3	47.9	35.4	28.9	35.5	44.5	33.0	29.0	37.0
Value	40.0	80.5	58.2	73.6	63.7	66.5	32.8	74.2	48.5	67.4	55.0	63.0
	L-2-w CoT											
Calculation	30.1	48.3	49.7	35.8	84.0	81.0	26.8	51.3	53.2	57.3	90.0	91.0
Factual	31.6	45.1	40.9	38.0	41.0	43.0	22.2	39.5	37.6	40.8	27.0	38.0
Normalization	57.7	66.0	46.1	23.0	73.0	53.0	63.7	79.1	49.0	31.1	66.0	67.0
Logical	27.9	62.8	46.1	48.1	33.0	48.0	21.0	56.1	36.2	49.3	23.0	42.0
Temporal	58.2	71.1	25.2	37.6	39.0	61.0	52.7	74.5	26.9	42.2	28.0	57.0
Security	35.4	39.2	35.7	46.4	22.0	48.0	30.9	42.2	33.0	56.9	22.0	47.0
Consistency	39.2	35.9	53.4	36.9	49.6	39.3	29.6	33.8	47.3	32.7	31.0	34.0
Value	39.6	83.7	59.2	74.6	58.0	72.0	35.3	76.2	50.5	68.2	51.0	62.0
	L-3-w/o CoT											
Calculation	38.1	46.1	47.2	30.6	57.1	66.7	35.2	52.1	56.9	55.3	88.0	88.0
Factual	33.1	40.4	41.6	34.2	40.4	37.1	25.0	41.3	32.5	39.4	28.0	36.0
Normalization	67.9	74.5	49.6	22.6	70.1	66.4	64.0	77.8	41.4	40.9	67.0	74.0
Logical	28.9	57.1	40.3	45.5	28.1	44.2	22.9	52.7	28.9	46.0	20.0	46.0
Temporal	72.8	71.3	27.0	34.0	64.3	59.9	68.9	75.2	27.9	49.3	56.0	60.0
Security	35.4	30.3	33.9	43.5	23.7	31.7	28.0	28.8	20.4	47.8	14.0	35.0
Consistency	38.8	36.2	52.4	36.0	49.6	39.5	29.3	34.1	42.9	35.8	35.0	40.0
Value	38.7	80.5	59.7	70.0	66.8	65.6	33.0	74.0	48.8	64.4	55.0	62.0
	L-3-w CoT											
Calculation	42.7	51.7	48.7	38.5	77.0	76.0	37.0	53.6	55.7	52.7	90.0	89.0
Factual	33.1	43.0	40.2	34.8	40.0	41.0	24.4	39.4	29.9	39.5	29.0	36.0
Normalization	62.0	66.0	51.8	24.1	55.0	59.0	65.3	72.5	38.4	41.7	60.0	76.0
Logical	28.4	58.3	43.2	48.0	30.0	44.0	21.2	49.8	30.3	42.3	24.0	41.0
Temporal	68.1	68.1	29.4	37.6	61.0	58.0	69.4	73.4	23.1	44.7	50.0	60.0
Security	36.1	32.3	32.9	42.2	28.0	38.0	31.7	30.9	19.7	49.9	14.0	33.0
Consistency	37.6	37.6	52.4	37.0	48.1	40.6	29.2	36.4	39.8	35.6	29.0	36.0
Value	41.2	82.4	60.7	73.2	65.0	68.0	33.1	69.9	41.2	62.3	51.0	58.0

Table 8: This table summarizes the Average Precision (P) and Recall (R) across eight anomaly categories in the FeTaQA, Spider+BEA, and WikiTQ datasets, evaluated using ChatGPT-4o and Gemini-1.5-Pro under various prompting strategies. Each prompt level L1 - L3 corresponds to a different depth of reasoning, with `-w/ocot` and `-w CoT` indicating the absence and presence of Chain-of-Thought (CoT) reasoning, respectively.

	FeTaQA		Spi+BEA		WikiTQ		FeTaQA		Spi+BEA		WikiTQ	
Category	P	R	P	R	P	R	P	R	P	R	P	R
	llama-3.1-70B-Instruct						**Deepseek-V3**					
	L-1-w/o CoT											
Calculation	13.9	36.7	33.6	37.6	60.0	37.0	29.2	50.9	57.5	44.7	77.0	79.0
Factual	17.8	32.1	20.8	26.0	29.0	14.0	27.2	39.9	41.7	31.8	28.0	35.0
Normalization	34.5	44.4	28.8	21.6	50.0	28.0	75.3	71.9	49.3	25.0	76.0	61.0
Logical	20.7	49.5	19.1	27.7	30.0	19.0	25.1	57.5	49.2	47.5	25.0	37.0
Temporal	42.1	44.5	11.8	24.3	35.0	26.0	66.6	79.1	24.9	37.1	45.0	54.0
Security	25.9	28.2	17.6	30.3	29.0	20.0	37.1	39.7	43.4	50.7	27.0	43.0
Consistency	23.9	25.9	25.9	22.4	37.0	16.0	37.6	32.3	50.6	28.5	45.0	33.0
Value	30.5	71.0	33.9	38.5	56.0	23.0	38.0	81.3	80.6	83.2	53.0	65.0
	L-1-w CoT											
Calculation	14.0	37.1	33.0	35.4	48.0	41.0	28.5	50.6	56.0	42.3	83.0	79.0
Factual	17.6	32.6	20.8	27.4	29.0	16.0	27.7	40.1	35.8	31.0	27.0	34.0
Normalization	30.6	45.8	28.9	22.9	50.0	25.0	70.8	71.2	51.7	24.3	78.0	60.0
Logical	19.5	47.5	16.6	27.6	27.0	22.0	26.2	59.6	46.4	47.4	30.0	46.0
Temporal	38.4	46.6	11.5	24.2	19.0	21.0	62.8	76.1	30.1	35.7	46.0	57.0
Security	25.1	31.1	19.8	31.6	21.0	23.0	38.3	41.0	47.2	48.3	25.0	43.0
Consistency	22.9	27.4	25.3	22.1	32.0	16.0	37.6	33.0	53.0	27.9	45.0	34.0
Value	30.7	72.2	33.0	38.1	48.0	32.0	37.6	80.1	80.9	81.7	60.0	64.0
	L-2-w/o CoT											
Calculation	15.6	38.2	28.1	32.0	61.0	33.0	31.8	52.8	52.3	39.6	81.0	77.0
Factual	19.7	31.7	20.7	29.5	37.0	17.0	28.1	39.9	42.6	31.5	35.0	33.0
Normalization	36.9	45.1	31.6	23.6	55.0	26.0	84.3	66.7	49.0	24.6	65.0	57.0
Logical	20.2	44.5	17.1	27.5	31.0	20.0	27.9	58.4	44.4	49.6	26.0	37.0
Temporal	37.2	44.0	11.7	23.2	31.0	28.0	66.3	76.8	31.5	34.3	49.0	56.0
Security	25.5	29.1	17.7	31.5	34.0	18.0	38.6	38.2	49.0	51.6	26.0	40.0
Consistency	23.0	23.1	25.8	21.4	37.0	13.0	37.2	31.9	51.9	28.4	49.0	30.0
Value	29.5	67.3	32.9	37.3	56.0	19.0	39.0	79.3	79.6	81.3	68.0	60.0
	L-2-w CoT											
Calculation	13.6	36.0	31.9	36.9	45.0	40.0	31.2	49.4	54.5	39.1	89.0	79.0
Factual	14.3	30.0	22.7	27.6	30.0	19.0	28.3	39.7	40.0	30.1	29.0	31.0
Normalization	33.0	48.4	33.0	24.0	57.0	31.0	79.7	61.4	53.5	23.0	82.0	57.0
Logical	19.0	47.0	16.9	28.8	26.0	23.0	26.5	59.5	48.5	48.5	26.0	37.0
Temporal	43.9	50.7	13.7	27.4	25.0	28.0	66.9	74.5	31.9	38.2	53.0	52.0
Security	20.3	26.7	18.4	33.1	24.0	22.0	37.5	38.6	49.9	51.2	32.0	44.0
Consistency	23.0	27.1	25.5	22.8	33.0	17.0	36.8	30.2	54.9	27.1	48.0	29.0
Value	30.0	69.5	32.9	38.7	55.0	38.0	39.1	78.7	80.9	80.8	67.0	58.0
	L-3-w/o CoT											
Calculation	20.1	37.8	35.5	38.1	62.0	17.0	38.3	54.7	58.2	37.9	76.0	76.0
Factual	19.4	27.8	22.5	26.7	31.0	12.0	35.1	36.9	40.4	28.4	27.0	31.0
Normalization	36.3	45.1	29.8	25.0	55.0	19.0	75.6	58.8	43.1	24.4	74.0	58.0
Logical	20.4	42.5	18.1	26.3	39.0	20.0	31.4	51.9	49.7	45.5	25.0	38.0
Temporal	50.8	44.3	10.8	25.7	49.0	14.0	80.3	75.7	29.7	38.0	65.0	56.0
Security	27.3	27.9	18.1	28.7	33.0	16.0	43.5	31.5	43.5	46.0	26.0	36.0
Consistency	23.3	24.6	25.9	21.3	34.0	14.0	40.8	29.9	54.9	27.7	50.0	31.0
Value	29.9	68.7	32.6	37.0	58.0	12.0	35.2	70.8	81.3	77.0	62.0	60.0
	L-3-w CoT											
Calculation	18.2	39.3	37.5	38.4	71.0	19.0	43.8	54.7	65.2	43.3	90.0	66.0
Factual	16.0	28.3	24.0	27.5	26.0	17.0	33.9	36.6	44.3	30.4	34.0	30.0
Normalization	37.1	53.6	26.1	25.4	53.0	33.0	48.6	56.2	44.3	24.7	79.0	67.0
Logical	17.6	43.1	17.4	27.5	20.0	22.0	28.7	55.5	45.5	46.1	26.0	36.0
Temporal	59.7	50.2	13.6	27.0	39.0	27.0	80.6	71.6	30.0	38.9	66.0	48.0
Security	23.2	23.1	19.7	30.1	18.0	17.0	45.6	27.6	42.4	45.6	34.0	31.0
Consistency	23.2	28.2	24.8	21.9	31.0	18.0	40.8	30.6	56.2	26.4	51.0	30.0
Value	27.9	66.3	32.6	37.4	42.0	42.0	38.1	71.1	81.1	76.1	72.0	54.0

Table 9: This table summarizes Average Precision (P) and Recall (R) across eight anomaly categories in the FeTaQA, Spider+BEA, and WikiTQ datasets, evaluated using llama-3.1-70B-Instruct and Deepseek-V3 under various prompting strategies. Each prompt level L_i corresponds to a different depth of reasoning, with `-w/ocot` and `-w CoT` indicating the absence and presence of Chain-of-Thought (CoT) reasoning, respectively.

	FeTaQA		Spi+BEA		WikiTQ		FeTaQA		Spi+BEA		WikiTQ	
Category	P	R	P	R	P	R	P	R	P	R	P	R
	llama-3.1-70B-Instruct						**Deepseek-V3**					
	L-4-w/o CoT											
Calculation	17.3	41.2	35.6	37.1	67.0	16.0	46.2	52.4	62.2	39.6	77.0	71.0
Factual	16.7	30.3	18.5	22.8	31.0	20.0	34.6	39.8	39.4	30.6	29.0	32.0
Normalization	37.7	52.3	29.1	28.6	53.0	32.0	50.6	59.5	42.7	27.2	77.0	66.0
Logical	18.2	44.6	16.4	22.4	27.0	18.0	29.1	55.1	47.9	44.5	20.0	35.0
Temporal	47.8	48.9	8.7	20.9	45.0	33.0	79.1	72.0	25.8	35.9	60.0	47.0
Security	24.2	25.5	20.0	28.9	27.0	21.0	47.2	33.6	45.8	48.1	33.0	33.0
Consistency	24.8	37.6	23.3	23.1	31.0	14.0	40.0	33.7	48.4	30.9	47.0	34.0
Value	29.2	71.4	34.7	37.8	50.0	23.0	40.6	81.5	81.4	81.3	62.0	64.0
	L-4-w CoT											
Calculation	19.3	40.5	35.4	38.3	70.0	33.0	45.0	52.1	65.6	40.2	80.0	65.0
Factual	16.1	30.6	21.9	26.0	31.0	20.0	34.0	36.6	36.0	29.3	30.0	32.0
Normalization	36.6	53.6	31.4	33.3	39.0	23.0	68.1	71.2	40.5	28.4	76.0	66.0
Logical	18.1	42.2	17.3	26.3	25.0	22.0	33.8	56.5	55.4	43.1	28.0	38.0
Temporal	45.9	50.0	10.0	25.1	35.0	27.0	75.7	65.8	32.8	35.4	63.0	48.0
Security	27.2	27.9	18.0	29.4	24.0	23.0	43.3	29.4	53.3	46.5	36.0	28.0
Consistency	24.2	35.6	24.6	25.1	28.0	20.0	40.8	33.8	50.1	31.6	49.0	33.0
Value	29.9	72.3	33.8	40.0	39.0	28.0	39.6	83.8	86.0	81.8	65.0	58.0
	MuSeVe											
Calculation	12.4	33.0	31.1	35.0	47.0	22.0	30.9	48.3	55.7	38.3	70.0	56.0
Factual	16.1	28.5	15.2	28.6	29.0	17.0	29.0	39.3	39.6	30.3	36.0	30.0
Normalization	40.9	47.1	33.1	25.6	51.0	23.0	72.6	53.6	45.8	22.9	73.0	57.0
Logical	18.5	42.5	16.0	25.7	25.0	22.0	27.3	58.1	46.4	47.3	34.0	35.0
Temporal	32.8	43.1	7.1	14.7	24.0	20.0	65.7	72.5	26.0	35.5	39.0	49.0
Security	24.6	30.2	16.3	29.2	25.0	22.0	38.6	40.4	41.8	46.8	38.0	43.0
Consistency	18.9	24.1	23.7	20.1	32.0	12.0	37.1	30.6	49.1	25.7	45.0	30.0
Value	25.2	63.0	33.3	35.5	52.0	33.0	37.3	76.4	78.6	77.7	57.0	56.0
	SeVCoT											
Calculation	16.2	31.8	35.0	38.3	69.0	31.0	30.6	43.8	55.0	36.5	84.0	68.0
Factual	16.7	27.6	19.2	25.9	38.0	18.0	28.4	38.1	37.4	30.7	35.0	34.0
Normalization	35.3	42.5	30.4	22.4	57.0	47.0	61.2	53.6	53.3	22.9	76.0	66.0
Logical	18.8	43.4	16.3	26.5	32.0	19.0	27.5	57.5	50.5	47.3	26.0	39.0
Temporal	35.1	42.0	11.4	24.3	43.0	29.0	66.7	72.2	27.9	37.6	60.0	53.0
Security	25.1	28.5	17.4	28.2	22.0	21.0	38.5	37.9	50.3	46.9	33.0	34.0
Consistency	20.1	22.9	23.4	19.1	32.0	19.0	35.4	28.5	48.7	24.2	41.0	31.0
Value	29.1	64.9	33.6	35.6	58.0	39.0	39.3	76.9	81.6	78.7	66.0	59.0

Table 10: This table summarizes Average Precision (P) and Recall (R) across eight anomaly categories in the FeTaQA, Spider+BEA, and WikiTQ datasets, evaluated using llama-3.1-70B-Instruct and Deepseek-V3 under various prompting strategies. Each prompt level L4 corresponds to a different depth of reasoning, with `-w/ocot` and `-w CoT` indicating the absence and presence of Chain-of-Thought (CoT) reasoning, respectively. The MuSeVe and SeVCoT configurations represent multi-reasoning and self-verification prompting variants.