Schema Lineage Extraction at Scale: Multilingual Pipelines, Composite Evaluation, and Language-Model Benchmarks

Jiaqi Yin
Microsoft
Redmond, WA
Jackie.Yin@microsoft.com

Yi-Wei Chen
Microsoft
Redmond, WA
yiweichen@microsoft.com

Meng-Lung Lee
Antra. Inc.
Seattle, WA
leemenglung1012@gmail.com

Xiya Liu
Microsoft
Redmond, WA
Xiya.Liu@microsoft.com

Abstract

Enterprise data pipelines, characterized by complex transformations across multiple programming languages, often cause a semantic disconnect between original metadata and downstream data. This "semantic drift" compromises data reproducibility and governance, and impairs the utility of services like retrieval-augmented generation (RAG) and text-to-SQL systems. To address this, a novel framework is proposed for the automated extraction of fine-grained schema lineage from multilingual enterprise pipeline scripts. This method identifies four key components: source schemas, source tables, transformation logic, and aggregation operations, creating a standardized representation of data transformations. For the rigorous evaluation of lineage quality, this paper introduces the Schema Lineage Composite Evaluation (SLiCE), a metric that assesses both structural correctness and semantic fidelity. A new benchmark is also presented, comprising 1,700 manually annotated lineages from real-world industrial scripts. Experiments were conducted with 12 language models, from 1.3B to 32B small language models (SLMs) to large language models (LLMs) like GPT-4o and GPT-4.1. The results demonstrate that the performance of schema lineage extraction scales with model size and the sophistication of prompting techniques. Specially, a 32B open-source model, using a single reasoning trace, can achieve performance comparable to the GPT series under standard prompting. This finding suggests a scalable and economical approach for deploying schema-aware agents in practical applications.

1 Introduction

Enterprise databases are foundational repositories powering critical business activities, with data scientists and analysts relying extensively on these systems to extract actionable insights. Typically, comprehensive metadata documentation is created alongside initial datasets, providing valuable context for interpretation and utilization. However, this metadata rapidly becomes outdated as data undergoes complex transformations through multi-stage processing pipelines employing heterogeneous programming languages such as SQL, Python, and C#. These transformations fundamentally alter original data schemas, creating substantial semantic gaps between initial metadata and derived datasets used for analytics and machine learning.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Evaluating the Evolving LLM Lifecycle: Benchmarks, Emergent Abilities, and Scaling.

This disconnect introduces severe documentation challenges known as "semantic drift"[1, 2]. The semantic gap also impairs the utility of services like retrieval-augmented generation (RAG) and text-to-SQL systems, which rely on accurate semantic understanding to function effectively. Consequently, organizations face substantial difficulties in data governance and rely heavily on a limited number of technical specialists familiar with transformation logic, severely constraining scalable data-driven decision-making. While large language models (LLMs) pretrained on general corpora show promise for automating metadata documentation tasks [3], they face significant limitations in enterprise environments due to privacy constraints and lack of domain-specific context [4]. Even when deployed internally, general-purpose LLMs struggle to capture nuanced semantics of transformed schemas without task-specific fine-tuning [5, 6].

We address these challenges by introducing a formal framework for schema lineage extraction from enterprise data pipeline code. We define schema lineage as a structured representation capturing four essential components: source schemas, source tables, transformation logic, and aggregation operations. This compact yet expressive format captures the complete semantic path from data origins to final outputs across complex, multi-language transformation scripts.

To support benchmarking, we manually annotated 1,700 schema lineages across 50 real-world enterprise pipeline scripts written in SQL, Python, and C#. We introduce **SLiCE** (Schema Lineage Composite Evaluation), a comprehensive metric that combines structural validity and semantic correctness into a unified score. Our extensive experiments across 12 language models using three prompting strategies reveal key trends on how model scale, prompt design, and script complexity affect extraction quality. Specially, we demonstrate that a 32B open-source model with chain-of-thought prompting achieves performance comparable to GPT-series models, offering cost-effective deployment paths for enterprise adoption.

In summary, we make four key contributions: (1) a formal definition of schema lineage tailored to multi-language enterprise pipelines, capturing source-to-output semantics across transformation logic and aggregation; (2) a high-quality benchmark of 1,700 manually annotated schema lineages from 50 real-world scripts; (3) the SLiCE metric, a comprehensive evaluation framework that enables fine-grained assessment of extraction quality; and (4) extensive experiments across 12 language models, demonstrating how model scale, prompting strategy, and script complexity influence extraction performance.

2 Dataset and Schema Lineage Definition

2.1 Enterprise Data Pipeline Collection

Industries routinely employ sophisticated, multi-stage, and multi-language transformation pipelines to support diverse analytical workflows. These pipelines typically begin with large-scale preprocessing using frameworks like PySpark and Scope [7] and transition to downstream metric computation in SQL or Python, reflecting the heterogeneity of real-world data engineering environments.

To capture these complexities, we curated a comprehensive dataset comprising 50 representative enterprise data pipeline scripts. These scripts span multiple programming languages, including SQL, C#, and Python (including PySpark). Each script, actively deployed within Microsoft, serves distinct analytical purposes, ranging from business metrics computation to marketing analytics, product insights, and user experience optimization.

We categorized scripts into three difficulty levels (easy, medium, hard) based on quantitative criteria detailed in Appendix A.1. Our dataset includes 19 easy scripts (averaging 26 schemas and 921 tokens per script), 22 medium scripts (averaging 28 schemas and 1,806 tokens per script), and 9 hard scripts (averaging 67 schemas and 4,687 tokens per script), collectively amounting to 1,700 schema annotations (Table 1).

While the full dataset is based on real, production-level scripts, we simulate a hard example in Appendix A.2 to illustrate their structural and logical characteristics. These examples preserve the multi-stage, multi-language complexity of the original pipelines while changing sensitive business logic.

Table 1: Overview of enterprise data pipeline scripts categorized by complexity level, detailing token count and schema statistics

Difficulty	Scripts	Token Count			Schema Count			
		Avg.	Min	Max	Total	Avg.	Min	Max
All	50	1,988.52	139	17,447	1,700	34.00	5	391
Easy	19	921.26	139	2,153	488	25.68	5	118
Medium	22	1,806.23	274	6,882	610	27.73	6	109
Hard	9	4,687.22	751	17,447	602	66.89	10	391

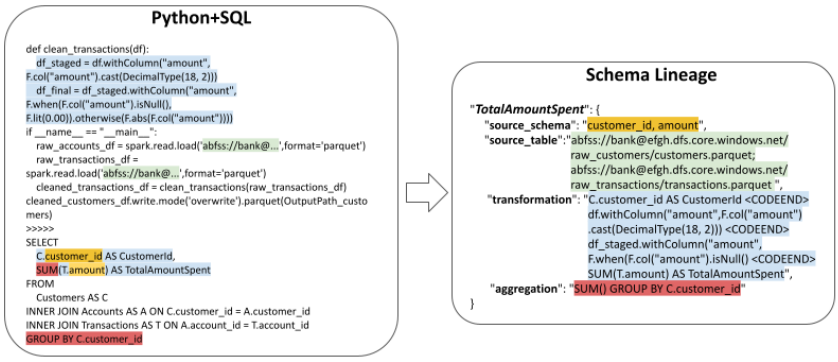


Figure 1: A visual illustration of schema lineage definition and annotation, based on the formal structure introduced in Section 2.2. The example demonstrates how raw data pipeline scripts combined with Python and SQL code is analyzed to extract the four core components (source schemas, source tables, transformation logic, and aggregation operations) of schema lineage for `TotalAmountSpent`. The resulting structured lineage represents a human-labeled gold annotation used for model evaluation and training.

2.2 Schema Lineage Definition and Annotation

We formally define schema lineage as a structured representation capturing the semantics of data transformations within enterprise data pipelines. A typical pipeline script reads from one or more source tables and produces one output table as a result of transformation logic. The output table consists of multiple schemas, corresponding to the columns or fields. For each schema in an output table, we extract a distinct schema lineage that traces its derivation from the original data sources. We conceptualize schema lineage as a structured mapping comprising four essential components:
1. Source Schemas: The original schema elements from which lineage originates, indicating the foundational data fields contributing to the resultant schema. Multiple source schemas, if applicable, are comma-separated;
2. Source Tables: The initial data tables containing source schemas, acting as primary data origins;
3. Transformation: The explicit code snippet or operational logic applied to transform source schemas into the resultant schema. A sequence of transformations is delimited using the `<CODEEND>` separator;
4. Aggregation: Aggregation operations applied throughout transformation, such as `GROUP BY`, `SUM`, `COUNT`, `MAX`, or `MIN`, alongside their grouping keys. A sequence of aggregations is similarly separated using the `<CODEEND>` delimiter.
Those components are essential because schema lineage serves as the connective tissue between raw data and downstream outputs. Without understanding how each schema element was derived, it becomes impossible to reconstruct the full context of a dataset, explain business metrics, or enable AI agents to operate reliably. For instance, tracing the lineage of a metric like `TotalAmountSpent` showed in Figure 1 requires more than matching column names. It demands precise reconstruction of how those values were computed, transformed, and aggregated from their original tables.

Our dataset includes schema lineages manually annotated by human experts, following strict consistency guidelines to ensure reliable experimental evaluation. Through meticulous annotation, we have produced 1,700 high-quality schema lineages covering diverse complexity levels and

transformation patterns, creating a robust gold standard for evaluating language models. Moreover, detailed reasoning traces were generated for each script to support varied prompting strategies during evaluation. These traces strengthen our assessment framework, ensuring comprehensive and rigorous evaluations of automated schema lineage extraction.

3 Schema Lineage Composite Evaluation (SLiCE)

Schema lineage requires accurately identifying the original source columns, tracing transformation logic, and capturing aggregation operations, even when they are distributed across multiple abstraction layers or languages. To meet these requirements, we propose a novel evaluation metrics called, **SLiCE**, specifically designed for **S**chema **Li**neage **C**omposite **E**valuation. Our approach recognizes that successful lineage extraction must satisfy multiple criteria simultaneously: structural correctness, semantic accuracy, and practical utility for enterprise applications.

3.1 Problem Statement

Given an enterprise data pipeline script and a target schema from the output table, our objective is to extract the corresponding schema lineage that traces the data transformation process from source to target. Let $\mathcal{S}$ represent a data pipeline script of multiple programming languages (SQL, C#, Python), and let σ denote a target schema in the output table generated by $\mathcal{S}$. Our goal is to map the script-schema pair to a structured schema lineage L. Each schema lineage L is defined as a structured dictionary with four required keys: `source_schema, source_table, transformation, aggregation`. This key-value format is essential for both evaluation and downstream parsing.

To streamline our mathematical formulation, we denote the value corresponding to each key using the following symbols: $\mathcal{C}$ for `source_schema` representing the set of source columns, $\mathcal{T}$ for `source_table` denoting the set of tables, $\mathcal{F}$ for `transformation`, the transformation logic, and $\mathcal{A}$ for `aggregation`, the final aggregation expression. We thus represent the schema lineage as a structured mapping:

$$L = \{\texttt{source\_schema} : \mathcal{C}, \texttt{source\_table} : \mathcal{T}, \texttt{transformation} : \mathcal{F}, \texttt{aggregation} : \mathcal{A}\} \quad (1)$$

During evaluation, we consider a predicted lineage $\widehat{L}$ generated by a language model and a gold standard lineage $L^\star$ annotated by experts. This dictionary-based representation ensures alignment with both the model's output structure and the evaluation interface.

3.2 SLiCE Definition

The SLiCE integrates structural validity [8] and semantic correctness [9] into a single value $\texttt{SLiCE}(\widehat{L}, L^\star) \in [0, 1]$, while still exposing component-level diagnostics (format, source, tables, transformation, aggregation).

Format Correctness. Schema lineage extraction requires strict adherence to both the dictionary structure and the output scaffolding imposed by the prompting strategy. Depending on whether the model is prompted with or without intermediate reasoning, the response must conform to one of the following formats: **with reasoning trace:** the response must contain both reasoning and answer blocks:

```
<think> ... reasoning trace ... </think>
<answer> ... schema lineage dictionary ... </answer>;
```

without reasoning trace: the response must contain only the answer block: `<answer> ... schema lineage dictionary ... </answer>`. In both cases, the lineage content inside the `<answer> </answer>` tag must follow a strict key-value dictionary format, containing exactly four keys: `source_schema`, `source_table`, `transformation`, and `aggregation`. The format correctness score enforces all these format constraints jointly:

$$\mathcal{M}_{\text{FMT}}(\widehat{L}) = \begin{cases} 1 & \text{if } \widehat{L} \text{ satisfies all <tag> structure and dictionary key requirements} \\ 0 & \text{otherwise} \end{cases}. \quad (2)$$

Any deviation, such as malformed tags, incorrect key names, or missing fields , results in immediate failure. This reflects the importance of strict structural adherence in automated parsing systems.

Source Schema Evaluation. Source schemas represent the foundational elements of lineage extraction, specifying which original columns contribute to the target schema. We compute a binary match based on exact set equality:

$$\mathcal{M}_{\text{SRC}}(\widehat{L}, L^\star) = \begin{cases} 1 & \text{if } \widehat{\mathcal{C}} = \mathcal{C}^\star \\ 0 & \text{otherwise} \end{cases}. \tag{3}$$

This metric enforces strict case-sensitive, order-insensitive matching of column names.

Source Table Evaluation. There are variations in source table naming conventions and hierarchies (e.g., database.schema.table vs. table), a simple exact-match metric is insufficient. $\mathcal{M}_{\text{TBL}}$ is proposed to combine a strict exact-match $F1$ score with a more flexible fuzzy similarity score F_u:

$$\mathcal{M}_{\text{TBL}}(\widehat{L}, L^\star) = w_1^{\text{TBL}} \cdot F1(\widehat{\mathcal{T}}, \mathcal{T}^\star) + w_2^{\text{TBL}} \cdot F_u(\widehat{\mathcal{T}}, \mathcal{T}^\star), \tag{4}$$

where the weights $w_1^{\text{TBL}} + w_2^{\text{TBL}} = 1$. F_u is defined to provides partial credit for predictions that are textually similar but not identical to the ground truth:

$$F_u(\widehat{\mathcal{T}}, \mathcal{T}^\star) = \frac{1}{2}\left[\frac{1}{|\widehat{\mathcal{T}}|}\sum_{t_i \in \widehat{\mathcal{T}}} \max_{t_j \in \mathcal{T}^\star} \text{FuzzyMatch}(t_i, t_j) + \frac{1}{|\mathcal{T}^\star|}\sum_{t_j \in \mathcal{T}^\star} \max_{t_i \in \widehat{\mathcal{T}}} \text{FuzzyMatch}(t_i, t_j)\right]. \tag{5}$$

The $\text{FuzzyMatch}(t_i, t_j)$ function computes a normalized similarity ratio of table name based on the Levenshtein distance [10]. The first term in Equation 5, Fuzzy precision, measures how well each predicted table matches the best candidate in the ground-truth set, while the second term, Fuzzy recall, measures how well each ground-truth table is represented by its best match in the predicted set. The `max` operator ensures a table is only scored against its most similar counterpart. This hybrid approach provides a more nuanced evaluation that rewards exactness while accommodating common naming variations.

Transformation and Aggregation Evaluation. Transformation ($\mathcal{F}$) and aggregation ($\mathcal{A}$) fields contain code snippets in various programming languages. These components are challenging to evaluate due to the possibility of logical equivalence despite syntactic variation. We define a novel Multi-AST similarity in Equation 6 that supports multilingual code comparison. First, we compute language-aware AST similarity:

$$\text{AST}_{\text{multi}}(\widehat{x}, x^\star) = \sum_{l \in \mathcal{L}} w_l \cdot \text{AST}_l(\widehat{x}, x^\star), \tag{6}$$

where $x \in \{\mathcal{F}, \mathcal{A}\}$, $\mathcal{L}$ is the set of candidate languages, and w_l is the confidence that x belongs to language l (with $\sum_l w_l = 1$). The weight w_l is computed as the normalized proportion of language-specific keywords observed in x, serving as a proxy for language attribution.

Motivated by CodeBLEU [9], we define the component metrics for transformation and aggregation, repectively, as a weighted average of standard BLEU [11], weighted BLEU (as introduced in CodeBLEU), and a modified AST-based similarity:

$$\mathcal{M}_{\text{TRF}}(\widehat{L}, L^\star) = w_1^{\text{TRF}} \cdot \text{BLEU}(\widehat{\mathcal{F}}, \mathcal{F}^\star) + w_2^{\text{TRF}} \cdot \text{BLEU}_{\text{weight}}(\widehat{\mathcal{F}}, \mathcal{F}^\star) + w_3^{\text{TRF}} \cdot \text{AST}_{\text{multi}}(\widehat{\mathcal{F}}, \mathcal{F}^\star), \tag{7}$$

$$\mathcal{M}_{\text{AGG}}(\widehat{L}, L^\star) = w_1^{\text{AGG}} \cdot \text{BLEU}(\widehat{\mathcal{A}}, \mathcal{A}^\star) + w_2^{\text{AGG}} \cdot \text{BLEU}_{\text{weight}}(\widehat{\mathcal{A}}, \mathcal{A}^\star) + w_3^{\text{AGG}} \cdot \text{AST}_{\text{multi}}(\widehat{\mathcal{A}}, \mathcal{A}^\star). \tag{8}$$

Each set of weights satisfies $\sum_{i=1}^{3} w_i^{\text{TRF}} = 1$ and $\sum_{i=1}^{3} w_i^{\text{AGG}} = 1$. While CodeBLEU includes AST similarity for single-language code, our approach extends this term to support multi-language settings by computing a language-aware aggregation over candidate AST parsers. We exclude the data-flow matching term from CodeBLEU, as schema lineage transformations often consist of partial and non-executable code snippets.

Composite Performance Score. The final evaluation metric is defined as:

$$\texttt{SLiCE}(\widehat{L}, L^\star) = \mathcal{M}_{\text{FMT}}(\widehat{L}) \cdot \mathcal{M}_{\text{SRC}}(\widehat{L}, L^\star) \cdot \left[\omega_{\text{TBL}} \cdot \mathcal{M}_{\text{TBL}} + \omega_{\text{TRF}} \cdot \mathcal{M}_{\text{TRF}} + \omega_{\text{AGG}} \cdot \mathcal{M}_{\text{AGG}}\right], \tag{9}$$

where the weights are predefined and satisfy $\omega_{\text{TBL}} + \omega_{\text{TRF}} + \omega_{\text{AGG}} = 1$. Other weights $(w_1^{\text{TBL}}, w_2^{\text{TBL}})$ in M_{TBL}, $(w_1^{\text{TRF}}, w_2^{\text{TRF}}, w_3^{\text{TRF}})$ in M_{TRF}, $(w_1^{\text{AGG}}, w_2^{\text{AGG}}, w_3^{\text{AGG}})$ in M_{AGG} are also predefined. Note that $\mathcal{M}_{\text{FMT}}$ and $\mathcal{M}_{\text{SRC}}$ are binary values. Equation 9 ensures that violations in basic structural constraints (e.g., incorrect format or source columns) nullify downstream correctness, reflecting how such errors propagate through real-world systems.

The proposed metric, SLiCE, offers a principled foundation for systematic performance analysis and model diagnostics. The fine-grained, component-wise scoring enables detailed benchmarking of language model capabilities across distinct aspects of schema lineage extraction, as demonstrated in our experiments. Importantly, the structured formulation of the SLiCE metric is not only useful for evaluation but also well-suited to serve as a reward signal in future supervised fine-tuning or reinforcement learning frameworks [8, 12].

To systematically investigate the level of contextual richness on performance of schema lineage extraction, we design three hierarchical prompting categories: **Base, Few-Shot, CoT**. Their detailed definition and prompt examples can be found in Appendix A.3. Note that we apply PagedAttention [13], a key-value caching mechanism, for open source small language models. It virtualizes the key-value cache memory to prevent fragmentation and optimize reuse. Since the constructed prompts, including scripts, extraction instructions, and provided examples, remain invariant for different schema queries within the same pipeline script, we compute and cache the model's key-value pairs once per pipeline. This optimization significantly reduces redundant computational efforts and accelerates the schema lineage extraction process.

4 Related Work

Early schema lineage extraction relied on conventional code analysis techniques, including abstract syntax tree (AST) parsing [14, 15], metadata mapping [16], and runtime analysis [17]. While reliable in single-language, static environments, these methods struggle with multi-stage, multi-language pipelines and evolving codebases.

Recent advances leverage large language models (LLMs) for automated lineage extraction [18, 19], reframing it as a code understanding task. Chain-of-Thought prompting with examples [18] enables high-quality lineage generation without fine-tuning, though existing approaches typically generate table- and operation-level lineages separately. Fine-tuned solutions like LLiM [19] capture anomalous patterns but may lack generalization for fundamental schema dependencies. Our work advances the no-fine-tuning paradigm by simultaneously generating both lineage types in a single query, specifically targeting data understanding rather than anomaly detection.

Open lineage datasets are rare due to sensitive business logic. While benchmarks like TPC-H [20] serve as synthesis templates [18], they lack the complexity of real-world multi-language implementations. Enterprise lineage graphs [21] better reflect real dependencies but aren't tailored for LLM-based extraction. Our dataset comprises real-world multi-language scripts, inheriting structural benefits from lineage graphs while explicitly supporting retrieval augmentation generation (RAG) [22] and Text-to-SQL applications [23].

Existing code generation metrics, execution-based pass@k [3] and semantic CodeBLEU [9] have fundamental limitations for lineage extraction. The pass@k metric requires complete executable programs, incompatible with partial transformation logic. CodeBLEU's applicability is constrained by heterogeneous multi-language transformations and incomplete code fragments that preclude traditional AST and data-flow analyses. Our SLiCE metric preserves CodeBLEU's foundations while providing native support for partial, multilingual code evaluation and incorporating LLM fine-tuning considerations [8].

5 Experiments

Our experimental evaluation is designed to investigate several key aspects of schema lineage extraction. We compare the performance of state-of-the-art LLMs with specialized SLMs to understand their relative capabilities on this task across data pipelines of varying difficulty. Methodologically, we assess the impact of different prompting strategies on extraction accuracy and validate the effectiveness of our proposed lineage metrics.

Table 2: Benchmark results of 12 language models evaluated on schema lineage extraction from 50 data pipeline scripts using three prompting strategies: base (zero-shot), one-shot, and chain-of-thought with a single reasoning trace (CoT-1). Mean corpus-level SLiCE scores and standard deviations are reported across six random seeds, ordered by model size.

Model	Size	Base	One-Shot	CoT-1
LLMs				
GPT-4.1 [24]	-	0.418 ± 0.005	0.673 ± 0.008	0.767 ± 0.007
GPT-4o [25]	-	0.284 ± 0.003	0.654 ± 0.007	0.759 ± 0.008
SLMs				
DeepSeek-Coder [26]	1.3B	0.000 ± 0.000	0.054 ± 0.015	0.038 ± 0.017
Qwen2.5-Coder [27]	1.5B	0.014 ± 0.002	0.309 ± 0.006	0.304 ± 0.017
Qwen2.5-Coder [27]	3B	0.100 ± 0.004	0.391 ± 0.015	0.445 ± 0.010
DeepSeek-Coder [26]	6.7B	0.003 ± 0.003	0.084 ± 0.018	0.509 ± 0.007
Mistral [28]	7B	0.026 ± 0.003	0.331 ± 0.005	0.227 ± 0.009
Qwen2.5-Coder [27]	7B	0.167 ± 0.005	0.487 ± 0.018	0.556 ± 0.009
Phi-4 [29]	14B	0.016 ± 0.003	0.511 ± 0.005	0.648 ± 0.005
Qwen2.5-Coder [27]	14B	0.286 ± 0.004	0.547 ± 0.005	0.646 ± 0.007
Codestral [30]	22B	0.126 ± 0.004	0.511 ± 0.005	0.662 ± 0.008
Qwen2.5-Coder [27]	32B	0.355 ± 0.004	0.623 ± 0.004	0.734 ± 0.007

5.1 Experimental Setup

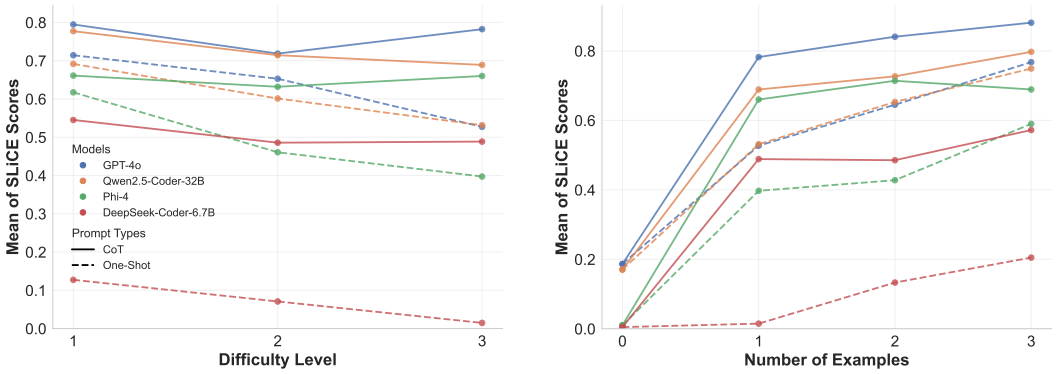
Our evaluation involves 12 language models (two LLMs and 10 SLMs) across 50 data pipeline scripts. Each SLM is deployed on a dedicated compute instance equipped with two NVIDIA H100 GPUs. We implement three core prompting categories, resulting in seven distinct strategies that are scaled according to script complexity. Detailed examples of these prompting strategies are provided in Appendix A.3. The experimental framework includes six randomized trials resulting in over 70,000 individual extraction tasks across all the conditions. We report the mean and standard deviation of corpus-level metric $\mathcal{M}_{\text{MOD}}(\Theta)$ as defined in Equation 11, which provides insights into metric stability and variability of model performance. Complete experimental setup details are available in Appendix B.

5.2 Results

Table 2 presents corpus-level performance across 12 language models, evaluated using three prompting strategies: base (zero-shot), one-shot, and chain-of-thought with one reasoning trace (CoT-1). The comprehensive results are in Table B.3 (Part 1-3). The consistently low standard deviation observed across random seeds underscores the robustness and reliability of our evaluation metrics.

Several key patterns emerge from our results. First, base prompting consistently yields the lowest performance across all models. Introducing a single output example (one-shot) substantially improves extraction accuracy. For instance, GPT-4.1 improves its SLiCE score by 61%, and the SLM Qwen2.5-Coder-32B sees a 75% increase. Interestingly, general-purpose language models such as GPT-4o, Mistral, and Phi-4 exhibit even greater results with one-shot prompting, achieving improvements exceeding 100%. Adding a reasoning trace (CoT-1) further enhances performance by over another 10% for models with size $\geq$ 3B, demonstrating the effectiveness of CoT reasoning in guiding schema lineage extraction. Secondly, there is positive correlation between model size and extraction performance. Within the same model families, holding the prompting strategy unchanged, larger models consistently outperform smaller models, highlighting model size as a significant factor; as seen in Figure B.2. Under CoT-1 prompting, Qwen2.5-Coder-32B achieves the highest SLiCE score of 0.734, more than doubling the performance of its smallest variant (1.5B), which scores 0.304.

We observe that CoT prompting yields diminishing returns for models with fewer than 3B parameters. For instance, DeepSeek-Coder-1.3B and Qwen2.5-Coder-1.5B exhibit decreased lineage extraction performance when moving from one-shot to CoT-1 prompting. Two potential explanations account for this trend. First, chain-of-thought reasoning is widely considered an emergent capability that typically arises in larger models, aligning with prior findings by Wei et al. [31]. Second, the longer



(a) Average SLiCE scores across three script difficulty levels (1: easy, 2: medium, 3: hard) for four models under one-shot and CoT-1 prompting.

(b) Average SLiCE scores on **hard** scripts with increasing numbers of examples (1–3) for few-shot and CoT prompting strategies.

Figure 2: Schema lineage extraction performance comparison across prompting strategies and script complexities for four models (GPT-4o, Qwen2.5-Coder-32B-Instruct, Phi-4-14B, and DeepSeek-Coder-6.7B). Line styles denote prompting strategies; colors indicate model variants. (a) shows the effect of script difficulty under different prompting strategies. (b) shows the effect of varying the number of examples in both few-shot and CoT prompting for hard scripts.

prompt lengths inherent to CoT may overwhelm small models with limited context windows. This degradation is consistent with observations by Liu et al. [32]. Comparatively, Qwen2.5-Coder-32B achieves performance on par with GPT-4o and GPT-4.1: its base prompting accuracy surpasses GPT-4o, while its one-shot and CoT-1 results are comparable to those of both proprietary LLMs.

To understand the impact of script complexity on extraction performance, we further stratify the SLiCE scores and illustrate the trends in Figure 2a. We select four representative models with varying scales: GPT-4o, Qwen2.5-Coder-32B, Phi-4, and DeepSeek-Coder-6.7B, using one-shot and CoT-1 prompting strategies across script difficulties. The rest of model performance is in Appendix C.

Figure 2a reveals that schema lineage extraction performance decreases as script complexity increases across most scenarios which aligns with our design intuition. When transitioning from one-shot to CoT-1 prompting, all models exhibit increased SLiCE scores, effectively mitigating the adverse effect of higher script complexity. This result underscores the significant benefit of incorporating even a single high-quality reasoning trace provided by a human expert into the prompt. For instance, Phi-4 (green color) achieves a SLiCE score of 0.660 on hard scripts using CoT-1 prompting (solid line), markedly surpassing the 0.397 score achieved with one-shot prompting (dash line). Additionally, the Qwen-2.5-Coder-32B under CoT-1 prompting (orange solid line) surpasses GPT-4o's performance under one-shot prompting (blue dash line) for scripts at all difficulty levels. This outcome is practically significant as it demonstrates that a 32B model, which can be internally deployed, can achieve performance comparable to the expensive GPT-4o.

We further investigate the effect of increasing the number of examples on schema lineage extraction performance, by analyzing average SLiCE scores for the hard scripts across the four representative models in Figure 2b. Increasing the number of examples consistently enhances the SLiCE scores across all models, demonstrating a clear positive correlation between example quantity and performance improvement. CoT prompting generally outperforms few-shot prompting across all configurations. However, while CoT with 2-3 examples achieves superior performance, the magnitude of improvement remains modest. For instance, the Qwen2.5-Coder-32B model experiences a substantial increase of 23% (from 0.531 to 0.653) from one-shot to two-shot prompting, whereas the improvement from CoT-1 to CoT-2 is considerably smaller at only 6% (from 0.689 to 0.727). This pattern suggests that schema lineage extraction benefits substantially from a single high-quality reasoning trace, with additional reasoning traces yielding diminishing returns.

6 Discussion

Our experiments demonstrate that the proposed SLiCE metric effectively captures schema lineage extraction performance across varying language models and prompting strategies. While proprietary LLMs deliver strong extraction performance, each prompt must contain complete data pipeline scripts, which can exceed hundreds of thousands of tokens, leading to cost escalation. Our work reveals that open-source models at the 32B parameter scale, when augmented with chain-of-thought reasoning traces, achieve extraction performance comparable to proprietary state-of-the-art LLMs such as GPT-4o and GPT-4.1. Incorporating even a single high-quality reasoning trace remarkably enhances performance.

A primary application enabled by accurate schema lineage extraction is the automated creation of high-quality documentation alongside dynamic data pipeline scripts. This documentation subsequently serves as a robust knowledge base for RAG systems. Take the schema lineage extraction in Figure 1 as an example. The schema `TotalAmountSpent` originates from the database columns `customer_id` and `amount`, with their definitions sourced from the database's metadata. Schema lineage explicitly traces transformations and aggregations, empowering the LLM to generate a precise and contextual business statement: *"TotalAmountSpent shows the total amount spent by each customer by aggregating individual transaction amounts. ...<business impact provided by LLM knowledge>".* Such detailed, dynamic, and domain-specific documentation significantly enriches downstream AI applications. Furthermore, accurate schema lineage substantially improves text-to-SQL tasks by providing precise definitions and relevant business contexts, ultimately enhancing AI-driven analytical workflows from human queries.

However, our approach faces several limitations. First, the requirement for human experts to provide reasoning trace examples presents a scalability challenge in production environments with diverse, unseen script patterns. Potential solutions include human-in-the-loop feedback mechanisms or developing specialized models fine-tuned for industrial pipeline reasoning. Second, lengthy pipeline scripts may exceed context windows of smaller language models, limiting their applicability despite their cost advantages. Third, the SLiCE metric employs predefined weights that may not generalize across all domains or reflect varying error criticality in specific business contexts. Future work could explore adaptive weighting schemes based on downstream task performance.

7 Conclusion

In this paper, we proposed an innovative framework for automated schema lineage extraction tailored to multi-language enterprise data pipelines. Recognizing the inherent semantic drift due to complex data transformations, our approach systematically captures schema lineage details (source schemas, tables, transformation logic, and aggregation operations) directly from pipeline scripts. We curated a robust benchmark dataset consisting of 1,700 schema annotations stratified across varying script complexities, representative of real-world industry scenarios. Central to our methodology is the SLiCE score, a composite evaluation metric that combines structural correctness with semantic precision. This metric enables granular diagnosis for the lineage of real-world applications. mportantly, provides a well-structured reward signal that can be leveraged for fine-tuning language models in future work, offering a direct path toward improving model alignment with schema lineage extraction tasks.

Our experimental analysis examined multiple state-of-the-art language models under diverse prompting strategies. Key findings revealed that the performance of schema lineage extraction significantly improves with increasing model size and contextual richness in prompts. Specifically, chain-of-thought reasoning significantly enhance extraction performance. We observed that 32B SLM achieves performance levels comparable to proprietary LLMs, highlighting their viability for enterprise deployment.

The proposed method directly facilitates high-quality dynamic documentation, significantly enhancing downstream applications such as RAG and text-to-SQL systems. By providing accurate, contextually-rich schema documentation, our approach empowers enterprises to maintain rigorous data governance and analytical reproducibility, effectively bridging the semantic gap in enterprise data transformation processes.

Acknowledgments and Disclosure of Funding

This work is supported by our manager Cheng Wu. We thank Lili Che and Naga Sai Kiran Kambhampati for curating the high-quality data pipeline scripts and annotating the schema lineage.

References

[1] J. P. Müller and T. Stein. A framework for measuring semantic drift in ontologies. In *Joint Proceedings of the Workshops on the Semantic Web: Semantics, Analytics, and Visualisation, SAW, and Trends in the Semantic Web, SemAW*, volume 1695, pages 31–38. CEUR-WS, 2016. URL https://ceur-ws.org/Vol-1695/paper42.pdf.

[2] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *VLDB*, 12 (1):41–58, May 2003. doi: 10.1007/s00778-002-0083-8. URL https://doi.org/10.1007/s00778-002-0083-8.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, et al. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

[4] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, et al. On the opportunities and risks of foundation models, 2022. URL https://arxiv.org/abs/2108.07258.

[5] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021. URL https://arxiv.org/abs/2005.11401.

[6] Scott Barnett, Stefanus Kurniawan, Srikanth Thudumu, Zach Brannelly, and Mohamed Abdelrazek. Seven failure points when engineering a retrieval augmented generation system, 2024. URL https://arxiv.org/abs/2401.05856.

[7] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. SCOPE: parallel databases meet MapReduce. *VLDB Journal*, 21(5):611–636, 2012. doi: 10.1007/s00778-011-0231-0.

[8] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

[9] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, et al. Codebleu: a method for automatic evaluation of code synthesis, 2020. URL https://arxiv.org/abs/2009.10297.

[10] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[11] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proc. Association for Computational Linguistics (ACL)*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. doi: 10.3115/1073083.1073135. URL https://aclanthology.org/P02-1040/.

[12] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. Execution-based code generation using deep reinforcement learning, 2023. URL https://arxiv.org/abs/2301.13816.

[13] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, et al. Efficient memory management for large language model serving with pagedattention, 2023. URL https://arxiv.org/abs/2309.06180.

[14] Andi Albrecht, Victor Uriarte, Jesús Leganés-Combarro, Jon Dufresne, Adam Greenhall, Simon Heisterkamp, et al. sqlparse: a non-validating sql parser for python. Python package (PyPI, Read the Docs), 2025. URL https://pypi.org/project/sqlparse/. Online; accessed 2025-07-30.

[15] Sqlfluff: The sql linter for humans. Open source project (sqlfluff.com, GitHub), 2025. URL https://sqlfluff.com/. Online; accessed 2025-07-30.

[16] Microsoft. Data lineage in classic data catalog. Microsoft Learn, Jul 2025. URL https://learn.microsoft.com/en-us/purview/data-gov-classic-lineage. Online; accessed 2025-07-30.

[17] Foundational, Inc. Automated data lineage tool. Product description (foundational.io), 2025. URL https://www.foundational.io/product/data-lineage. Online; accessed 2025-07-30.

[18] Zhangti Li, Wenbin Guo, Yabing Gao, Di Yang, and Lin Kang. A large language model-based approach for data lineage parsing. *Electronics*, 14(9):1762, April 2025. doi: 10.3390/electronics14091762. URL https://www.mdpi.com/2079-9292/14/9/1762.

[19] Volodymyr Kuznetsov. Introducing the Large Lineage Model (LLiM): Our Path to Securing the Future of Data. Cyberhaven Engineering Blog, March 2025. URL https://www.cyberhaven.com/engineering-blog/large-lineage-model-llim-our-path-securing-data/. Online; accessed 2025-07-30.

[20] The Apache Doris Project. Tpc-h benchmark – apache doris documentation. https://doris.apache.org/docs/benchmark/tpch/. Accessed: 2025-07-30.

[21] Yunpeng Chen, Ying Zhao, Xuanjing Li, Jiang Zhang, Jiang Long, and Fangfang Zhou. An open dataset of data lineage graphs for data governance research. *Visual Informatics*, 8(1):1–5, 2024. URL http://dblp.uni-trier.de/db/journals/vi/vi8.html#ChenZLZLZ24.

[22] Jerry Liu. LlamaIndex, 11 2022. URL https://github.com/jerryjliu/llama_index.

[23] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task, 2019. URL https://arxiv.org/abs/1809.08887.

[24] OpenAI. Gpt-4.1, 2025. URL https://openai.com/index/gpt-4-1/. Accessed: 2025-07-30.

[25] OpenAI. Hello gpt-4o, May 2024. URL https://openai.com/index/hello-gpt-4o/.

[26] DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence, 2024. URL https://arxiv.org/abs/2406.11931.

[27] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, et al. Qwen2.5-coder technical report, 2024. URL https://arxiv.org/abs/2409.12186.

[28] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, et al. Mistral 7b, 2023. URL https://arxiv.org/abs/2310.06825.

[29] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, et al. Phi-4 technical report, 2024. URL https://arxiv.org/abs/2412.08905.

[30] Mistral AI. Codestral, 2024. URL https://mistral.ai/news/codestral/.

[31] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, et al. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

[32] Nelson F. Liu, Kevin Dabrol, John Bradshaw, Bryan McMahan, William Fedus, Noam Shazeer, Kuang-Huei Li, and Adams Wei Yu. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638. URL https://aclanthology.org/2024.tacl-1.9.

[33] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, et al. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

[34] Haoran Xu, Baolin Peng, Hany Awadalla, Dongdong Chen, Yen-Chun Chen, Mei Gao, et al. Phi-4-mini-reasoning: Exploring the limits of small reasoning language models in math, 2025. URL https://arxiv.org/abs/2504.21233.

[35] Marah Abdin, Sahaj Agarwal, Ahmed Awadallah, Vidhisha Balachandran, Harkirat Behl, Lingjiao Chen, et al. Phi-4-reasoning technical report, 2025. URL https://arxiv.org/abs/2504.21318.

[36] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation, 2024. URL https://arxiv.org/abs/2406.00515.

A Data Gallery

A.1 Script Difficulty

To quantify the complexity of data pipeline scripts, we introduce a scoring framework that evaluates each script based on its structural and operational features. Scripts are scored from 0 to 3 across three independent dimensions: data sources, transformations, and aggregations. A point is awarded for each dimension that demonstrates a higher level of complexity, as detailed below:

- **Data Sources**:
  - 0 points for scripts accessing one or two distinct data sources.
  - +1 point for scripts accessing three or more distinct data sources.
- **Transformation**:
  - 0 points for scripts with only basic transformations (e.g., column renaming, type casting).
  - +1 point for scripts that include a transformation chain, where the output of one operation serves as the input to another.
- **Aggregation**:
  - 0 points for scripts with no aggregation or pivot operations.
  - +1 point for scripts containing any aggregation function (e.g., `SUM`, `COUNT`, `PIVOT`).

The total complexity score for a script is the sum of the points from each dimension:

$$\text{Total Score} = \text{Points(Data Sources)} + \text{Points(Transformation)} + \text{Points(Aggregation)}$$

The final score determines the script's difficulty level, as defined in Table A.1.

Table A.1: Difficulty levels for script data based on scoring criteria.

Difficulty Level	Score	Description
Level 1: Easy	0 or 1	Scripts with minimal complexity, exhibiting at most one complexity factor (e.g., multiple data sources, a transformation chain, or an aggregation).
Level 2: Medium	2	Scripts incorporating two of the three complexity factors, such as multiple sources with a transformation chain but no aggregation.
Level 3: Hard	3	Scripts featuring all three complexity factors: multiple data sources (geq3), chained transformations, and at least one aggregation or pivot operation.

A.2 Scripts

To trace schema lineage from real-world scripts that frequently incorporate multiple programming languages, we developed a custom parsing strategy capable of handling multi-language code environments. Modern data processing workflows typically employ different programming languages optimized for specific computational tasks. Data engineers commonly utilize Python with specialized libraries such as PySpark, an interface for Apache Spark that enables distributed processing of large datasets across cluster computing environments. This approach facilitates efficient large-scale data cleaning and transformation operations. Subsequently, analysts and business users employ SQL for analytics and reporting tasks on the processed data. Listing 1 presents a synthetic script demonstrating the integration of Python and SQL components with level of difficulty as hard. We employ the delimiter >>>>> to denote programming language transitions during the parsing process. Our schema lineage tracing algorithm operates using a bottom-up traversal approach, initiating from a pre-specified target column and propagating upward through the computational graph. All transformation and aggregation operations that influence the target column are captured and recorded according to our standardized schema lineage representation format. One complete example of schema lineage is showed in Table A.2.

Listing 1: Python Code + SQL

```
import pyspark.sql.functions as F
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, IntegerType, StringType,
    DateType, TimestampType, DecimalType, DoubleType
from datetime import datetime, timedelta

def clean_customers(df):
    email_cleaned = df.withColumn(
        "email_address",
        F.when(
            F.col("email_address").isNull() |
            F.lower(F.col("email_address")).isin("", "invalid-email", "none"),
            F.lit("invalid_format@example.com")
        ).otherwise(F.col("email_address"))
    )

    names_formatted = email_cleaned \
        .withColumn("first_name", F.initcap("first_name")) \
        .withColumn("last_name", F.initcap("last_name"))

    gender_normalized = names_formatted.withColumn(
        "gender",
        F.when(
            F.lower("gender").isin("none", "prefer not to say"),
            F.lit("Prefer Not To Say")
        ).otherwise(F.initcap("gender"))
    )

    location_normalized = gender_normalized.withColumn(
        "state_province", F.upper("state_province")
    )

    phone_cleaned = location_normalized.withColumn(
        "phone_number", F.regexp_replace("phone_number", "[^0-9]", "")
    )

    registration_parsed = phone_cleaned.withColumn(
        "registration_date", F.to_timestamp("registration_date")
    ).filter(F.col("registration_date").isNotNull())

    premium_flagged = registration_parsed.withColumn(
        "is_premium_member",
        F.when(F.lower("is_premium_member").isin("true", "1", "yes"), F.lit(True))
         .otherwise(F.lit(False))
    )

    deduplicated = premium_flagged.dropDuplicates(["customer_id"])

    return deduplicated

def clean_accounts(df):
    balance_casted = df.withColumn("balance", F.col("balance").cast(DecimalType(18,
        2)))

    balance_cleaned = balance_casted.withColumn(
        "balance",
        F.when(F.col("balance").isNull() | (F.col("balance") < 0), F.lit(0.00))
        .otherwise(F.col("balance"))
    )

    account_type_cleaned = balance_cleaned.withColumn(
        "account_type",
```

```
        F.when(F.lower("account_type").isin("none", "unspecified"), F.lit("Unspecified"))
         .otherwise(F.initcap("account_type"))
    )

    status_formatted = account_type_cleaned.withColumn("status", F.initcap("status"))

    opening_date_parsed = status_formatted.withColumn(
        "opening_date", F.to_date("opening_date")
    ).filter(F.col("opening_date").isNotNull())

    interest_casted = opening_date_parsed.withColumn(
        "interest_rate", F.col("interest_rate").cast(DoubleType())
    )

    credit_limit_casted = interest_casted.withColumn(
        "credit_limit", F.col("credit_limit").cast(DecimalType(18, 2))
    )

    return credit_limit_casted.dropDuplicates(["account_id"])


def clean_transactions(df):
    df = df.withColumn("transaction_timestamp", F.to_timestamp("transaction_timestamp")) \
          .filter(F.col("transaction_timestamp").isNotNull()) \
          .withColumn(
              "transaction_timestamp",
              F.when(F.col("transaction_timestamp") > F.current_timestamp(), F.current_timestamp())
               .otherwise(F.col("transaction_timestamp"))
          ) \
          .withColumn("amount", F.col("amount").cast(DecimalType(18, 2))) \
          .withColumn("amount", F.when(F.col("amount").isNull(), F.lit(0.00)).otherwise(F.abs("amount"))) \
          .withColumn(
              "transaction_type",
              F.when(F.lower("transaction_type").isin("unknown", "none"), F.lit("Other"))
               .otherwise(F.initcap("transaction_type"))
          ) \
          .withColumn("status", F.initcap("status"))

    return df.dropDuplicates()

if __name__ == "__main__":
    spark.conf.set("spark.storage.synapse.linkedServiceName",linked_service_name)
    spark.conf.set("fs.azure.account.oauth.provider.type","com.bank.azure.synapse.tokenlibrary.LinkedServiceBasedTokenProvider")

    raw_customers_df = spark.read.load('abfss://bank@efgh.dfs.core.windows.net/raw_customers/customers.parquet',format='parquet')
    raw_accounts_df = spark.read.load('abfss://bank@efgh.dfs.core.windows.net/raw_accounts/accounts.parquet',format='parquet')
    raw_transactions_df = spark.read.load('abfss://bank@efgh.dfs.core.windows.net/raw_transactions/transactions.parquet',format='parquet')

    print("\n--- Applying Cleaning Transformations ---")
    cleaned_customers_df = clean_customers(raw_customers_df)
    cleaned_accounts_df = clean_accounts(raw_accounts_df)
    cleaned_transactions_df = clean_transactions(raw_transactions_df)

    OutputPath_customers='abfss://bank@efgh.dfs.core.windows.net/customersprod/customers.parquet'
```

```
    OutputPath_accounts='abfss://bank@efgh.dfs.core.windows.net/accountprod/accounts.parquet'
    OutputPath_transactions='abfss://bank@efgh.dfs.core.windows.net/transactionsprod/transactions.parquet'

    cleaned_customers_df.write.mode('overwrite').parquet(OutputPath_customers)
    cleaned_accounts_df.write.mode('overwrite').parquet(OutputPath_accounts)
    cleaned_transactions_df.write.mode('overwrite').parquet(OutputPath_transactions)


>>>>>

SELECT
    C.customer_id AS CustomerId,
    C.first_name AS FirstName,
    C.last_name AS LastName,
    C.is_premium_member AS IsPremiumMember,
    C.registration_date AS CustomerRegistrationDate,
    A.account_type AS AccountType,
    A.balance AS CurrentAccountBalance,
    A.credit_limit AS AccountCreditLimit,
    A.opening_date AS AccountOpeningDate,
    SUM(T.amount) AS TotalAmountSpent,
    COUNT(T.transaction_id) AS MonthlyTransactionCount,
    AVG(T.amount) AS AverageMonthlyTransactionAmount
FROM
    Customers AS C
INNER JOIN
    Accounts AS A ON C.customer_id = A.customer_id
INNER JOIN
    Transactions AS T ON A.account_id = T.account_id
WHERE
    T.transaction_timestamp >= '2025-05-01' AND T.transaction_timestamp < '2025-06-01'
    AND T.transaction_type IN ('Withdrawal', 'Purchase', 'Bill Payment', 'Transfer-Out')
    AND T.status = 'Completed'
GROUP BY
    C.customer_id,
    C.first_name,
    C.last_name,
    C.is_premium_member,
    C.registration_date,
    A.account_type,
    A.balance,
    A.credit_limit,
    A.opening_date
ORDER BY
    TotalAmountSpent DESC, C.customer_id, A.account_type;
```

Table A.2: Schema Lineage for the AverageMonthlyTransactionAmount column from Listing 1

Key	Value
source_schema	amount, customer_id, first_name, last_name, is_premium_member, registration_date, account_type, balance, credit_limit, opening_date, transaction_timestamp, transaction_type, status
source_table	abfss://bank@efgh.dfs.core.windows.net/raw_customers/customers.parquet; abfss://bank@efgh.dfs.core.windows.net/raw_accounts/accounts.parquet; abfss://bank@efgh.dfs.core.windows.net/raw_transactions/transactions.parquet
transformation	C.customer_id AS CustomerId <CODEEND> email_cleaned.withColumn(" first_name", F.initcap("first_name")) <CODEEND> C.first_name AS FirstName <CODEEND> email_cleaned.withColumn("last_name", F.initcap(" last_name")) <CODEEND> C.last_name AS LastName <CODEEND> registration_parsed.withColumn("is_premium_member", F.when(F.lower(" is_premium_member").isin("true", "1", "yes"), F.lit(True)).otherwise(F. lit(False))) <CODEEND> C.is_premium_member AS IsPremiumMember <CODEEND> phone_cleaned.withColumn("registration_date", F.to_timestamp(" registration_date")) <CODEEND> C.registration_date AS CustomerRegistrationDate <CODEEND> balance_cleaned.withColumn(" account_type", F.when(F.lower("account_type").isin("none", "unspecified "), F.lit("Unspecified")).otherwise(F.initcap("account_type"))) <CODEEND > A.account_type AS AccountType <CODEEND> df.withColumn("balance", F.col ("balance").cast(DecimalType(18, 2))) <CODEEND> balance_casted. withColumn("balance", F.when(F.col("balance").isNull() | (F.col("balance ") < 0), F.lit(0.00)).otherwise(F.col("balance"))) <CODEEND> A.balance AS CurrentAccountBalance <CODEEND> interest_casted.withColumn(" credit_limit", F.col("credit_limit").cast(DecimalType(18, 2))) <CODEEND> A.credit_limit AS AccountCreditLimit <CODEEND> status_formatted. withColumn("opening_date", F.to_date("opening_date")) <CODEEND> A. opening_date AS AccountOpeningDate <CODEEND> df.withColumn("amount", F. when(F.col("amount").isNull(), F.lit(0.00)).otherwise(F.abs("amount"))) <CODEEND> df.withColumn("amount", F.col("amount").cast(DecimalType(18, 2))) <CODEEND> AVG(T.amount) AS AverageMonthlyTransactionAmount
aggregation	AVG() GROUP BY C.customer_id, C.first_name, C.last_name, C. is_premium_member, C.registration_date, A.account_type, A.balance, A. credit_limit, A.opening_date

A.3 Prompts

Our three hierarchical prompting categories:

- **Base Prompting**: This strategy provides only the essential pipeline script along with explicit extraction instructions specifying target output formats and component definitions. It serves as a baseline by representing the minimal necessary context required for schema lineage extraction.
- **Few-Shot Prompting**: This strategy enhances the base prompting approach by integrating concrete input-output example pairs directly into the prompt, providing tangible references that guide the language model's understanding of expected outputs. We scale the quantity of these examples according to pipeline complexity, providing one example for easy pipelines, up to two for medium-complexity pipelines, and up to three for hard pipelines.
- **Chain-of-Thought (CoT)**: Building upon few-shot prompting, this advanced strategy incorporates detailed human-generated reasoning traces that illustrate step-by-step derivations of schema lineage from pipeline code. The inclusion of explicit reasoning processes aims to guide the language model through logical inference steps.

We present the prompt templates designed for our schema lineage extraction task, structured around three distinct prompting strategies: **Base**, **Few-Shot**, and **Chain-of-Thought (CoT)**. Each template incorporates placeholders for the data pipeline script, with examples included exclusively in the Few-Shot and CoT configurations. The number of examples provided scales according to input script complexity: one example for *Easy* cases, up to two for *Medium* complexity, and up to three for *Hard* scenarios. All prompt templates direct the model to generate structured output conforming to a specified JSON-like schema enclosed within <answer> </answer> tags. The CoT template uniquely incorporates an intermediate reasoning step delimited by <think> </think> tags to facilitate explicit reasoning processes.

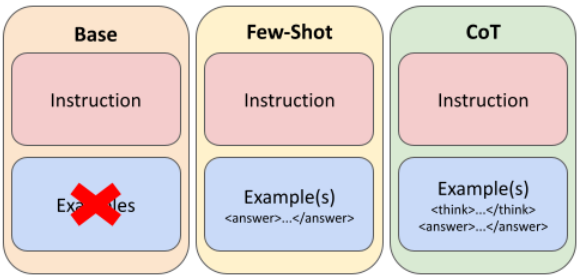


Figure A.1: Comparison of prompting strategies for schema lineage extraction. Base prompting provides only task instructions, few-shot prompting incorporates example outputs to demonstrate the expected format, and Chain-of-Thought (CoT) prompting includes explicit reasoning traces that guide the lineage process step-by-step.

Base Prompt Template

```
You are a data lineage analysis assistant. Your task is to analyze the provided data
 generation script and trace the lineage of a specific column which is specified by
the user.

Your response must include <answer> </answer> part:
<answer> {
  "source_schema": "...",
  "source_table": "...",
  "transformation": "...",
  "aggregation": "..."
} </answer>.

... (additional instructions omitted for brevity) ...

Data Pipeline Script: YOUR PINELINE SCRIPT
```

Few-Shot Prompt Template (1, 2, or 3 examples)

```
You are a data lineage analysis assistant. Your task is to analyze the provided data
 generation script and trace the lineage of a specific column which is specified by
the user.

Your response must include <answer> </answer> part:
<answer> {
  "source_schema": "...",
  "source_table": "...",
  "transformation": "...",
  "aggregation": "..."
} </answer>.

... (additional instructions omitted for brevity) ...

Data Pipeline Script: YOUR PIPELINE SCRIPT

Examples: YOUR OUTPUT EXAMPLE(S)
```

```
"""
```

Chain-of-Thought Prompt Template (1, 2, or 3 examples)

```
You are a data lineage analysis assistant. Your task is to analyze the provided data
 generation script and trace the lineage of a specific column which is specified by
the user.

Your response must include two parts:
1. <think> ... </think>
2. <answer> {{
  "source_schema": "...",
  "source_table": "...",
  "transformation": "...",
  "aggregation": "..."
}} </answer>.

... (additional instructions omitted for brevity) ...

Data Pipeline Script: YOUR PIPELINE SCRIPT

Examples: YOUR OUTPUT EXAMPLE(S)
"""
```

B Experiment Setup

Initially, we evaluated a comprehensive set of language models, encompassing two LLMs (GPT-4.1[24] and GPT-4o [25]), alongside 16 distinct SLMs. The SLM cohort included Qwen2.5-Coder variants (1.5B, 3B, 7B, 14B, 32B) [27], Mistral-7B [28], Codestral-22B [30], CodeLlama variants (7B, 13B, 34B) [33], DeepSeek-Coder variants (1.3B, 6.7B, 16B)) [26], and Phi-4 configurations (mini [34], 14B [29], reasoning-14B [35]). The majority of these models underwent pretraining on code corpora or subsequent alignment for coding tasks, establishing their reputation for robust coding capabilities [36]. Mistral-7B [28] and Phi-4 [29] series represents general-purpose architectures to assess the performance characteristics of domain-agnostic SLMs in coding contexts. After preliminary assessments, we excluded six models due to either excessive inference time or consistently poor performance, resulting in the final selection: Qwen2.5-Coder (1.5B, 3B, 7B, 14B, 32B), Mistral-7B, Codestral-22B, DeepSeek-Coder (1.3B, 6.7B, and Phi-4 (14B).

We extract schema lineage across 50 data pipeline scripts using three categories of prompting strategies detailed in Section A.3. Data experts crafted human reasoning traces to support the CoT prompting strategy: one reasoning trace per easy script, two per medium script, and three per hard script. Consequently, we implemented seven distinct prompting strategies: base, few-shot with one example (one-shot), few-shot with two examples (two-shot), few-shot with three examples (three-shot), CoT with one reasoning trace (CoT-1), CoT with two reasoning traces (CoT-2), and CoT with three reasoning traces (CoT-3). This comprehensive design allows us to investigate how different prompting strategies influence extraction accuracy across varying script complexities. We parse these outputs and evaluate the predicted schema lineage ($\widehat{L}$) against expert-annotated ground truth ($L^\star$) using `SLiCE` scores. The weights of `SLiCE` are assigned for the all experiments, $w_1^{\text{TBL}} = 0.7, w_2^{\text{TBL}} = 0.3; w_1^{\text{TRF}} = w_1^{\text{AGG}} = 0.5, w_2^{\text{TRF}} = w_2^{\text{AGG}} = 0.3, w_3^{\text{TRF}} = w_3^{\text{AGG}} = 0.2; \omega_{\text{TBL}} = 0.4, \omega_{\text{TRF}} = 0.4, \omega_{\text{AGG}} = 0.2$.

Evaluation Protocol. For each language model Θ, model predictions are scored against expert annotations using the `SLiCE` metric defined in Section 3.2. For each script s_i containing schemas σ_{ik}, we compute a *script-level* score by averaging schema-level scores:

$$\mathcal{M}_{\text{SCR}}(s_i, \Theta) = \frac{1}{K_i} \sum_{k=1}^{K_i} \texttt{SLiCE}\big(\widehat{L}_{ik},\, L^\star_{ik}\big), \tag{10}$$

where $\widehat{L}_{ik}$ and $L^\star_{ik}$ represent predicted and gold lineage, respectively. To derive a *corpus-level* evaluation, we average across all scripts:

$$\mathcal{M}_{\text{MOD}}(\Theta) = \frac{1}{I} \sum_{i=1}^{I} \mathcal{M}_{\text{SCR}}(s_i, \Theta) = \frac{1}{I} \sum_{i=1}^{I} \frac{1}{K_i} \sum_{k=1}^{K_i} \texttt{SLiCE}\big(\widehat{L}_{ik},\, L^\star_{ik}\big). \tag{11}$$

C Additional Results

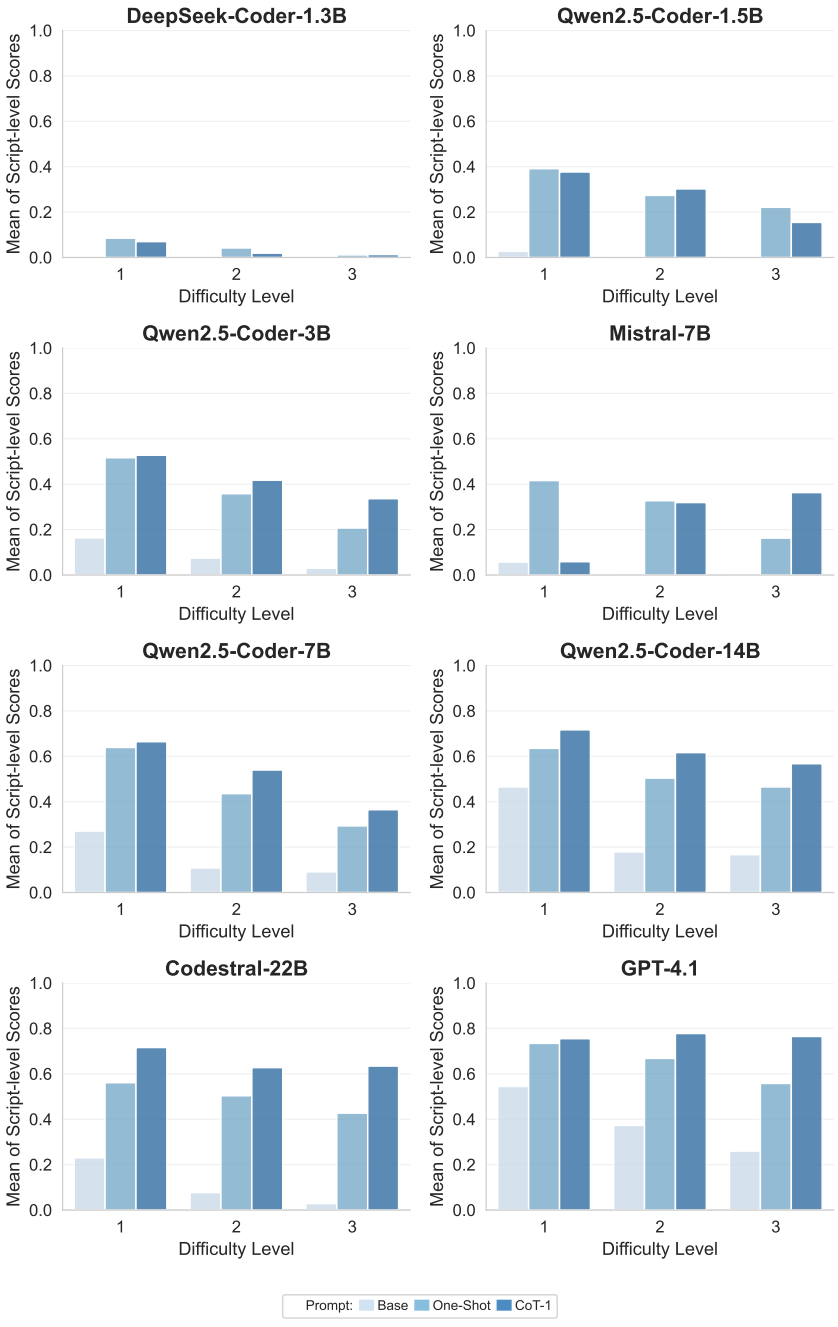


Figure B.1: **Performance comparison across models and prompt strategies by script difficulty**. Bar plots show mean script-level scores for eight language models across three prompting strategies: base (no additional output examples), few-shot (one example), and CoT (one reasoning trace exmaple). Scripts are grouped by difficulty level (1-3), with higher difficulty indicating more complex reasoning requirements. Larger models consistently outperform smaller ones. CoT prompting provides moderate improvements across most difficulty levels.

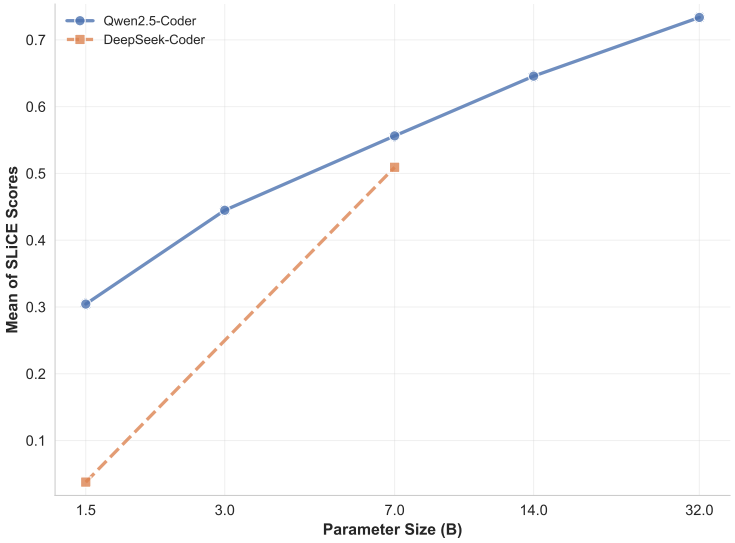


Figure B.2: **Model performance scaling with parameter size for Chain-of-Thought prompting**. The plot shows how mean SLiCE vary with model parameter size (in billions) for Qwen2.5-Coder (1.5B, 3B, 7B, 14B, 32B) and DeepSeek-Coder (1.3B, 6.7B) model families when having one human reasoning trace in the prompt. Both model families demonstrate improved performance with increased parameter size, with Qwen2.5-Coder models consistently outperforming DeepSeek-Coder models across all parameter scales.

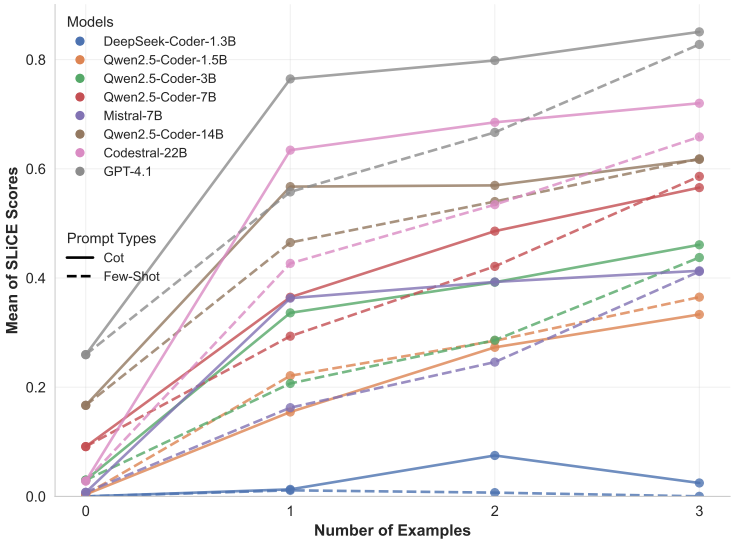


Figure B.3: **Average SLiCE across language models and prompting strategies for hard scripts**. Y-axis the average SLiCE for 8 language models (GPT-4.1, Codestral-22B, Qwen2.5-Coder (7B, 3B, 1.5B), Mistral-7B, and DeepSeek-Coder-1.5B) across different prompting strategies: base (no additional output examples), few-shot (one example), and CoT (one reasoning trace example). The results indicate that larger models generally perform better, with CoT prompting yielding higher metrics than few-shot prompting.

Table B.3: Benchmark results of language models on schema lineage extraction (Part 1): Qwen2.5-Coder variants. Mean corpus-level SLiCE scores and standard deviations across six random seeds (Mean ± Standard Deviation).

Parameter	Difficulty	Qwen2.5-Coder				
		1.5B	3B	7B	14B	32B
Base	All	0.014 ± 0.002	0.100 ± 0.004	0.167 ± 0.005	0.286 ± 0.004	0.355 ± 0.004
	Easy	0.027 ± 0.004	0.163 ± 0.004	0.271 ± 0.011	0.465 ± 0.010	0.578 ± 0.007
	Medium	0.006 ± 0.002	0.074 ± 0.005	0.108 ± 0.006	0.179 ± 0.006	0.239 ± 0.004
	Hard	0.004 ± 0.001	0.030 ± 0.006	0.091 ± 0.005	0.167 ± 0.009	0.171 ± 0.006
One-shot	All	0.054 ± 0.015	0.391 ± 0.015	0.487 ± 0.018	0.547 ± 0.005	0.623 ± 0.004
	Easy	0.391 ± 0.009	0.517 ± 0.017	0.639 ± 0.013	0.635 ± 0.006	0.692 ± 0.004
	Medium	0.274 ± 0.012	0.358 ± 0.021	0.436 ± 0.031	0.504 ± 0.009	0.601 ± 0.007
	Hard	0.221 ± 0.006	0.207 ± 0.008	0.293 ± 0.029	0.465 ± 0.024	0.531 ± 0.010
Two-shot	Medium	0.345 ± 0.007	0.467 ± 0.012	0.547 ± 0.020	0.586 ± 0.008	0.664 ± 0.009
	Hard	0.285 ± 0.030	0.286 ± 0.025	0.421 ± 0.018	0.540 ± 0.009	0.653 ± 0.007
Three-shot	Hard	0.365 ± 0.026	0.438 ± 0.027	0.586 ± 0.052	0.618 ± 0.016	0.749 ± 0.015
CoT-1	All	0.304 ± 0.017	0.445 ± 0.010	0.556 ± 0.009	0.646 ± 0.007	0.734 ± 0.007
	Easy	0.377 ± 0.021	0.528 ± 0.003	0.664 ± 0.014	0.717 ± 0.014	0.777 ± 0.010
	Medium	0.302 ± 0.014	0.418 ± 0.021	0.540 ± 0.013	0.616 ± 0.014	0.714 ± 0.010
	Hard	0.154 ± 0.036	0.336 ± 0.009	0.365 ± 0.019	0.567 ± 0.012	0.689 ± 0.019
CoT-2	Medium	0.293 ± 0.015	0.437 ± 0.011	0.568 ± 0.015	0.685 ± 0.017	0.748 ± 0.014
	Hard	0.273 ± 0.013	0.392 ± 0.011	0.486 ± 0.015	0.570 ± 0.012	0.727 ± 0.024
CoT-3	Hard	0.333 ± 0.010	0.461 ± 0.012	0.566 ± 0.029	0.617 ± 0.014	0.797 ± 0.019

Table B.3: Benchmark results of language models on schema lineage extraction (Part 2): DeepSeek-Coder models. Mean corpus-level SLiCE scores and standard deviations across six random seeds (Mean ± Standard Deviation).

Parameter	Difficulty	DeepSeek-Coder	
		1.3B	6.7B
Base	All	0.000 ± 0.000	0.003 ± 0.003
	Easy	0.000 ± 0.000	0.006 ± 0.004
	Medium	0.000 ± 0.000	0.000 ± 0.001
	Hard	0.000 ± 0.000	0.005 ± 0.007
One-shot	All	0.054 ± 0.015	0.084 ± 0.018
	Easy	0.085 ± 0.023	0.127 ± 0.024
	Medium	0.042 ± 0.016	0.071 ± 0.025
	Hard	0.011 ± 0.008	0.015 ± 0.011
Two-shot	Medium	0.007 ± 0.006	0.143 ± 0.044
	Hard	0.007 ± 0.005	0.133 ± 0.037
Three-shot	Hard	0.000 ± 0.001	0.205 ± 0.051
CoT-1	All	0.038 ± 0.017	0.509 ± 0.007
	Easy	0.070 ± 0.016	0.545 ± 0.010
	Medium	0.019 ± 0.019	0.486 ± 0.013
	Hard	0.013 ± 0.021	0.489 ± 0.018
CoT-2	Medium	0.043 ± 0.028	0.501 ± 0.009
	Hard	0.075 ± 0.019	0.485 ± 0.008
CoT-3	Hard	0.025 ± 0.009	0.573 ± 0.023

Table B.3: Benchmark results of language models on schema lineage extraction (Part 3): GPT models and other language models. Mean corpus-level SLiCE scores and standard deviations across six random seeds (Mean ± Standard Deviation).

Parameter	Difficulty	GPT-4.1	GPT-4o	Phi-4	Codestral-22B	Mistral-7B
Base	All	0.418 ± 0.005	0.284 ± 0.003	0.016 ± 0.003	0.126 ± 0.004	0.026 ± 0.003
	Easy	0.544 ± 0.006	0.379 ± 0.005	0.017 ± 0.005	0.230 ± 0.005	0.057 ± 0.004
	Medium	0.373 ± 0.007	0.241 ± 0.004	0.018 ± 0.005	0.077 ± 0.009	0.008 ± 0.003
	Hard	0.260 ± 0.007	0.186 ± 0.006	0.010 ± 0.005	0.028 ± 0.003	0.007 ± 0.001
One-shot	All	0.673 ± 0.008	0.654 ± 0.007	0.511 ± 0.005	0.511 ± 0.005	0.331 ± 0.005
	Easy	0.734 ± 0.004	0.714 ± 0.003	0.617 ± 0.007	0.561 ± 0.005	0.416 ± 0.007
	Medium	0.668 ± 0.016	0.653 ± 0.017	0.461 ± 0.006	0.503 ± 0.007	0.327 ± 0.009
	Hard	0.558 ± 0.017	0.527 ± 0.007	0.397 ± 0.012	0.427 ± 0.003	0.163 ± 0.005
Two-shot	Medium	0.751 ± 0.008	0.685 ± 0.010	0.548 ± 0.011	0.571 ± 0.008	0.427 ± 0.008
	Hard	0.667 ± 0.007	0.645 ± 0.009	0.428 ± 0.038	0.534 ± 0.009	0.246 ± 0.012
Three-shot	Hard	0.828 ± 0.008	0.768 ± 0.010	0.590 ± 0.029	0.658 ± 0.010	0.412 ± 0.010
CoT-1	All	0.767 ± 0.007	0.759 ± 0.008	0.648 ± 0.005	0.662 ± 0.008	0.227 ± 0.009
	Easy	0.755 ± 0.006	0.795 ± 0.007	0.661 ± 0.002	0.716 ± 0.009	0.059 ± 0.011
	Medium	0.778 ± 0.009	0.718 ± 0.009	0.632 ± 0.013	0.627 ± 0.017	0.319 ± 0.013
	Hard	0.765 ± 0.015	0.782 ± 0.016	0.660 ± 0.027	0.634 ± 0.021	0.363 ± 0.017
CoT-2	Medium	0.844 ± 0.006	0.767 ± 0.009	0.670 ± 0.008	0.650 ± 0.014	0.361 ± 0.012
	Hard	0.798 ± 0.011	0.841 ± 0.015	0.714 ± 0.012	0.685 ± 0.012	0.394 ± 0.034
CoT-3	Hard	0.851 ± 0.014	0.881 ± 0.010	0.689 ± 0.008	0.720 ± 0.020	0.413 ± 0.015