TQTS-BENCH: A MULTI-SYNTAX BENCHMARK FOR TEXT-TO-QUERY OVER TIME-SERIES DATABASES

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) have significantly advanced natural language querying over relational databases, yet their ability to query time-series databases (TSDBs) remains largely unassessed. Existing benchmarks fail to adequately capture the non-unified query syntaxes, diverse application domains, and unique time-specific query intents inherent to TSDBs. To address this gap, we introduce TQTS-BENCH, a multi-syntax benchmark for evaluating text-to-query capabilities over TSDBs. TQTS-BENCH contains 6,125 high-quality question–answering (QA) pairs spanning 97 TSDBs, 23 distinct query syntaxes, 22 application domains, and 4 types of time-specific query intents. It is constructed through a human-centric AI-assisted workflow, where all QA pairs are carefully reviewed and revised by domain experts to ensure quality and correctness. Extensive evaluations of advanced LLMs and state-of-the-art text-to-query methods reveal challenges in querying TSDBs. Even the best-performing model evaluated, Claude-Opus-5, achieves only 48.98% execution accuracy, while humans reach 87.34%. Error analysis reveals that this performance gap mainly stems from the heterogeneous query syntaxes across different TSDBs, misinterpretation of time-specific intents, and incorrect schema linking. These findings highlight new opportunities to narrow the gap between current LLM capabilities and the requirements of TSDB queries in real-world applications. The benchmark is available at: `https://anonymous.4open.science/r/TQTS-Bench-00CD`.

1 INTRODUCTION

Time-series data are widely used in real-world applications, such as the Internet of Things (IoT), AIOps, and cloud services. This widespread adoption has driven the rapid development of specialized time-series databases (TSDBs) (Jensen et al., 2017). However, analyzing data stored in TSDBs remains challenging because retrieving such data requires learning specialized query syntax (Dranca et al., 2026), particularly for users without expertise in databases. Recently, due to the rapid advancement of large language models (LLMs), text-to-query techniques have progressed significantly, approaching human-level performance on relational database (RDB) benchmarks (e.g., Spider (Yu et al., 2018) and BIRD (Li et al., 2023)). This progress has substantially lowered the barrier to analyzing data stored in databases. More recently, a few studies, such as Zhang et al. (2026) and Dranca et al. (2026), have begun to develop **T**ext-to-**Q**uery techniques specifically for **TS**DBs (**TQTS**). However, it remains unclear how far these methods have advanced TQTS capabilities due to the lack of a comprehensive TQTS benchmark comparable to Spider and BIRD for RDBs.

Despite its importance, constructing a TQTS benchmark is not a straightforward extension of text-to-query benchmarks for RDBs, owing to three key differences (Fig. 1).

- **Syntaxes**: For RDBs, their query languages share the common SQL syntax with differences mostly limited to small dialects. However, for TSDBs, their query syntaxes can be substantially different (Fig. 1(a)). For example, Flux in InfluxDB adopts a pipeline-style syntax, whereas PromQL in Prometheus relies on nested, function-oriented expressions. Moreover, even TSDBs that stay close to SQL, such as QuestDB, extend it with a wide range of time-specific operations and functions, further diverging from basic SQL syntax.
- **Domains**: In domains that require the continuous collection of high-frequency data, such as IoT and cloud services, TSDBs are predominant. Because RDBs are not specifically

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

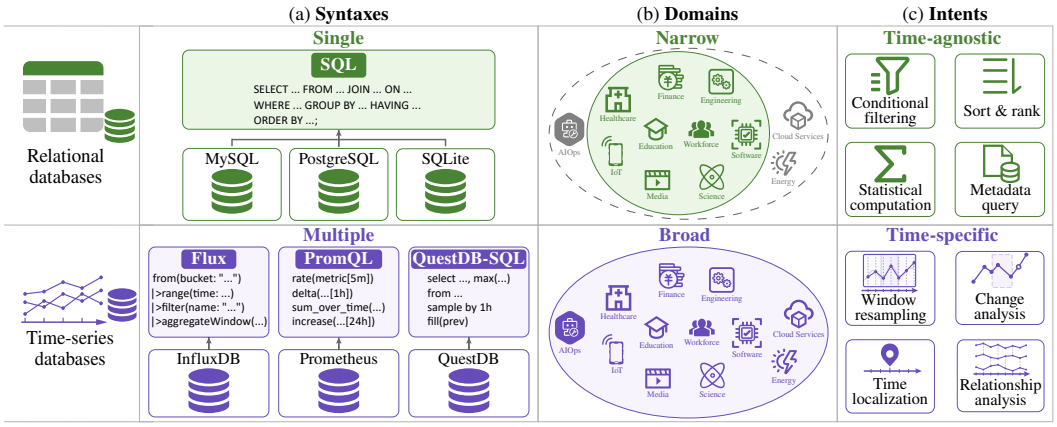


Figure 1: The three key differences between RDBs and TSDBs: (a) **Syntaxes**: unified SQL vs. diverse, non-uniform query syntaxes. (b) **Domains**: narrow domain coverage vs. broad domain coverage. (c) **Intents**: time-agnostic operations vs. time-specific analysis.

designed for such workloads, existing text-to-query benchmarks for RDBs rarely cover these domains, resulting in limited domain coverage (Fig. 1(b)).

- **Intents**: Queries over RDBs typically focus on time-independent intents such as filtering, sorting, and aggregation. In contrast, TSDB queries involve many time-specific intents, such as change analysis and relationship analysis (Fig. 1(c)). This makes text-to-query for RDBs and TSDBs differ in both question and intent understanding.

Due to the three differences above, rather than simply extending existing text-to-query benchmarks for RDBs, we construct TQTS-BENCH, a new benchmark specifically designed for text-to-query over TSDBs. TQTS-BENCH contains 6,125 text-to-query question-answering (QA) pairs, spanning 97 TSDBs with **23 distinct query syntaxes**. These TSDBs span **22 application domains**, while the QA pairs cover **4 time-specific query intents**. To construct this benchmark, we first identified 23 popular and representative TSDB management systems based on the well-known *DB-Engines Ranking of TSDB Management System*[1] and collected 427 datasets and 1,405 seed QAs related to these TSDB management systems from public sources such as TSDB official documentation, forums, and technical posts. These 427 datasets were manually filtered based on data authenticity and schema complexity, resulting in 97 datasets. In collaboration with domain experts, we analyzed the 1,405 seed QAs and identified 9 representative query intents. Based on these query intents and the seed QAs, we developed a human-centric AI-assisted workflow supported by a visual analytics tool to construct new QA pairs. Finally, these QA pairs underwent a rigorous cross-validation process to ensure their correctness and quality, and 6,125 QA pairs were verified.

We conduct a comprehensive evaluation of advanced LLMs and representative text-to-query methods on TQTS-BENCH. The results show that current LLMs still struggle with TQTS tasks. For example, the best-performing model, Claude-Opus-5, achieves an execution accuracy of only 48.98%, which remains far from human performance (87.34%), highlighting the challenges posed by this benchmark. Meanwhile, both existing RDB methods and TSDB methods also achieve limited performance, suggesting that existing methods still face difficulties in TQTS tasks and exhibit limited transferability and generalizability across different settings. We further conduct an error analysis and identify three main factors behind these failures, including heterogeneous query syntaxes across different TSDBs, misinterpretation of time-specific intents, and incorrect schema linking. Based on these findings, we discuss opportunities and promising directions for improving TQTS performance to enable more effective time-series data analysis in real-world applications.

[1] https://db-engines.com/en/ranking/time+series+dbms

2 Related Work

Text-to-query benchmarks over RDBs. RDB benchmarks have established well-defined evaluation paradigms for the Text-to-SQL task. Spider (Yu et al., 2018) evaluates whether models can generalize to unseen databases in different domains. BIRD (Li et al., 2023) extends this setting to larger and real-world databases, requiring models to use database contents and external knowledge. Spider 2.0 (Lei et al., 2025) further targets complex enterprise environments with large schemas and realistic query workflows. Beyond these general-purpose benchmarks, several studies focus on specific applications. For example, ScienceBenchmark (Zhang et al., 2023) targets scientific databases, while SParC (Yu et al., 2019b) and CoSQL (Yu et al., 2019a) focus on contextual and conversational queries. Additionally, many benchmarks (Vo et al., 2022; Chang et al., 2023; Liu et al., 2026) evaluate model capabilities on reasoning and query robustness. Despite covering broad evaluation settings, these benchmarks are grounded in RDBs and ignore time-series query requirements. However, TSDBs inherently contain multiple query syntaxes and are designed for time-series analysis. Therefore, existing RDB benchmarks are insufficient for evaluating text-to-query over TSDBs.

Text-to-query benchmarks over TSDBs. Recent studies have developed several text-to-query benchmarks for specific TSDBs. For example, TEFD (Wu et al., 2026) evaluates text-to-Flux generation for InfluxDB, while another study (Dranca et al., 2026) evaluates the generation of both Flux and InfluxQL queries in an agro-food production scenario. PromCopilot (Zhang et al., 2026) focuses on generating PromQL queries for Prometheus-based system monitoring. However, these studies are often limited to specific TSDBs, query syntaxes, or application domains, preventing a comprehensive evaluation of LLMs' text-to-query capabilities across diverse TSDB environments and scenarios. To address this gap, we develop TQTS-Bench, a multi-syntax benchmark that spans a broad range of TSDBs, query syntaxes, and application domains, enabling a systematic and comprehensive evaluation of LLMs for text-to-query tasks over TSDBs.

3 Dataset Construction

3.1 Task Definition

Given a natural-language question, a database schema, and a target TSDB, the TQTS task aims to generate an executable query that correctly retrieves the requested data from the TSDB. Here, the database schema describes the logical organization of time-series data, including how timestamps, data values, and associated metadata are structured and typed (Bader et al., 2017). Depending on the TSDB management system, these elements may be represented as tables or measurements, columns or fields, and tags or labels. The target TSDB refers to a dataset stored in a specified TSDB management system (e.g., InfluxDB).

3.2 Construction pipeline

As shown in Fig. 2, the construction of TQTS-Bench consists of two main steps: TSDB construction and QA construction and verification. The TSDB construction step collects and curates 97 high-quality TSDBs, and the QA construction and verification step constructs 6,125 QA pairs with a human-centric AI-assisted workflow.

3.2.1 TSDB construction

Based on the well-known *DB-Engines Ranking of TSDB Management System*, we first select 23 popular and representative TSDB management systems depending on the accessibility. With these systems, we then collect real-world TSDB datasets from diverse application scenarios. Although existing studies (Li et al., 2023; Yu et al., 2018; Zhang et al., 2026; Wu et al., 2026) provide datasets for text-to-query tasks, these datasets are typically distributed as standalone files (e.g., CSV files) rather than as operational TSDBs. Therefore, we gather publicly available datasets from multiple real-world sources, including official websites, hosting platforms that export data from specific TSDBs, and datasets documented as being collected or stored using TSDBs. In parallel, we compiled a set of seed QAs to facilitate subsequent identification of the query intent. These QA pairs were sourced from official TSDB documentation, community forum discussions, and technical posts.

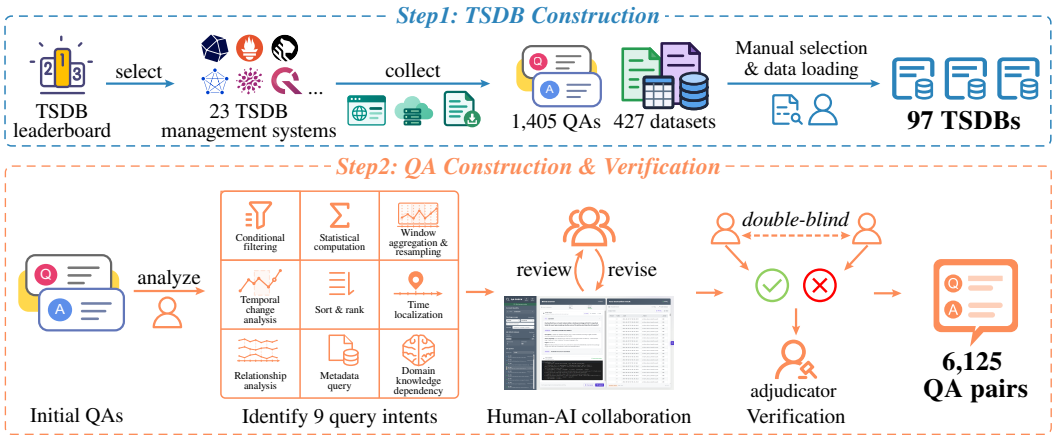


Figure 2: The construction pipeline of TQTS-BENCH.

The collection process results in 427 candidate TSDB datasets and 1,405 seed QAs. We then manually inspect these candidate datasets based on data authenticity and schema complexity, and finally select 97 representative source datasets. Subsequently, we import these datasets into their corresponding TSDB management systems to instantiate the database environments used in our benchmark. For TSDB details, refer to App. A.

3.2.2 QA CONSTRUCTION AND VERIFICATION

In TQTS-BENCH, each QA pair consists of a natural-language question and its corresponding answer for a specific TSDB. To construct the QA pairs, we first identify representative query intents from the seed QAs. We also developed a visual analytics tool that supports a human-centric AI-assisted workflow to help annotators directly explore and interact with the underlying TSDB.

Query intent identification. With the collected seed QAs, we collaborate with domain experts to analyze the underlying query requirements in these pairs and common TSDB usage scenarios. Based on this analysis, we identify nine representative query intents, comprising four time-specific query intents (**I1**–**I4**) and five time-agnostic query intents (**I5**–**I9**). Specifically, the time-specific intents are: **I1**: Window aggregation & resampling; **I2**: Temporal change analysis; **I3**: Time localization; and **I4**: Relationship analysis. The time-agnostic intents are: **I5**: Conditional filtering; **I6**: Statistical computation; **I7**: Sort & rank; **I8**: Metadata query; and **I9**: Domain knowledge dependency. We find that most query requirements can be represented by a single intent or a combination of multiple intents, indicating that the identified intents cover a broad range of TSDB usage needs. These nine query intents provide a common foundation for QA construction across different TSDB systems, regardless of their underlying query syntax. For details, refer to App. A.3.

Human-AI collaborative construction. In TQTS-BENCH, a qualified QA pair should consist of a natural-language question that faithfully reflects query intents in the target TSDB scenario, together with an executable query that correctly answers it under the given schema and TSDB context. To facilitate QA construction, we develop a visual analytics tool that supports a human-centric AI-assisted workflow. Specifically, we provide LLMs (e.g., GPT-5.6-Sol) with the identified query intents, database schemas, and TSDB context, requiring them to generate the corresponding questions. Annotators then review these questions to verify they match the query intents and reflect meaningful TSDB scenarios, refining their descriptions when necessary to improve clarity and alignment with the query intents. If the questions are determined, the annotators then use LLMs to generate the corresponding executable queries and construct the generated unverified QA pairs.

QA verification. After construction, all generated unverified QA pairs undergo a final verification step to ensure their correctness and quality. To facilitate this process, we use the developed visual analytics tool. For each unverified QA pair, we conduct a cross-validation procedure. Specifically, two annotators independently hand-write queries based on the question, schema, and provided TSDB context. They can execute the hand-written queries to determine whether the query results match the question's requirements. If both annotators obtain results consistent with the unverified answer

and both determine that the question is answerable on the target TSDB, the QA pair is accepted as verified. Otherwise, the case is escalated to an adjudicator. The adjudicator examines the annotators' queries, results, and feedback, and revises the QA pairs accordingly. The revised QA pairs then undergo a further round of verification, or are finalized by the adjudicator if the issue is resolved. For the visual analytics tool and verification details, refer to App. B.

Overall, a total of 6,125 QA pairs are verified. During this process, the annotators collectively spent more than 600 hours and executed 30,774 queries, averaging approximately 6 minutes inspection and 5 executions per QA pair. This rigorous verification process facilitates careful inspection and iterative validation, thereby ensuring the quality and correctness of the constructed QA pairs.

3.3 Data Statistics

TQTS-Bench is a multi-syntax benchmark designed for TQTS tasks. Specifically, we summarize the data statistics of TQTS-Bench from three perspectives: supported query syntaxes, domain coverage, and time-specific query intents.

Multiple query syntaxes. In total, TQTS-Bench contains 6,125 QA pairs spanning 23 distinct query syntaxes across 97 TSDBs, which differ substantially in their structures and usage patterns. As shown in Fig. 3, for the same question, InfluxDB's Flux represents the query as a pipeline of transformations, Prometheus's PromQL uses metric selectors and range-vector functions, and QuestDB SQL extends conventional SQL with dedicated temporal operations such as time sampling. This diversity makes TQTS-Bench challenging, as models must generate queries that are both semantically correct and syntactically compatible with the target TSDB. For more details, refer to App. A.1.

Q What was sensor A's average temperature each hour over the last 24 hours?

InfluxDB

```
from(bucket: "sensors")
  |> range(start: -24h)
  |> filter(fn: (r) =>
      r.sensor == "A" and
      r._field == "temp")
  |> aggregateWindow(
      every: 1h,
      fn: mean,
      createEmpty: true)
```

Prometheus

```
avg_over_time(
    temp{sensor="A"}[1h]
)[24h:1h]
```

QuestDB

```
SELECT
    ts,
    avg(temp)
FROM readings
WHERE sensor = 'A'
SAMPLE BY 1h FROM
dateadd('h', -24, now())
TO now() FILL(NULL)
ALIGN TO CALENDAR;
```

Figure 3: A representative question expressed using Flux, PromQL, and QuestDB SQL.

Broad domain coverage. Fig. 4(a) compares the domain coverage of TQTS-Bench with existing text-to-query benchmarks over RDBs and TSDBs. TQTS-Bench covers all 22 application domains considered in the comparison, while previous benchmarks typically cover only a subset of them. In particular, many RDB benchmarks emphasize general domains such as finance, healthcare and education. In contrast, TSDB benchmarks are often concentrated on scenarios involving continuously generated, high-frequency time-series data, such as IoT, AIOps, and cloud services, where efficient ingestion and temporal analysis are critical. By spanning both general-purpose and time-intensive domains, TQTS-Bench supports a more comprehensive evaluation of text-to-query methods across real-world data settings. For more domain details, refer to App. A.2

More time-specific query intents. Fig. 4(b) reports the query intent statistics across different text-to-query benchmarks. Compared with other benchmarks, TQTS-Bench has the highest average numbers of both query intents and time-specific intents per QA pair, as well as the highest average number of time-specific operations per QA pair. Together, these statistics highlight the temporal complexity of TQTS-Bench, making it a challenging benchmark for translating diverse time-specific intents into executable queries. For more query intent details, refer to App. A.3.

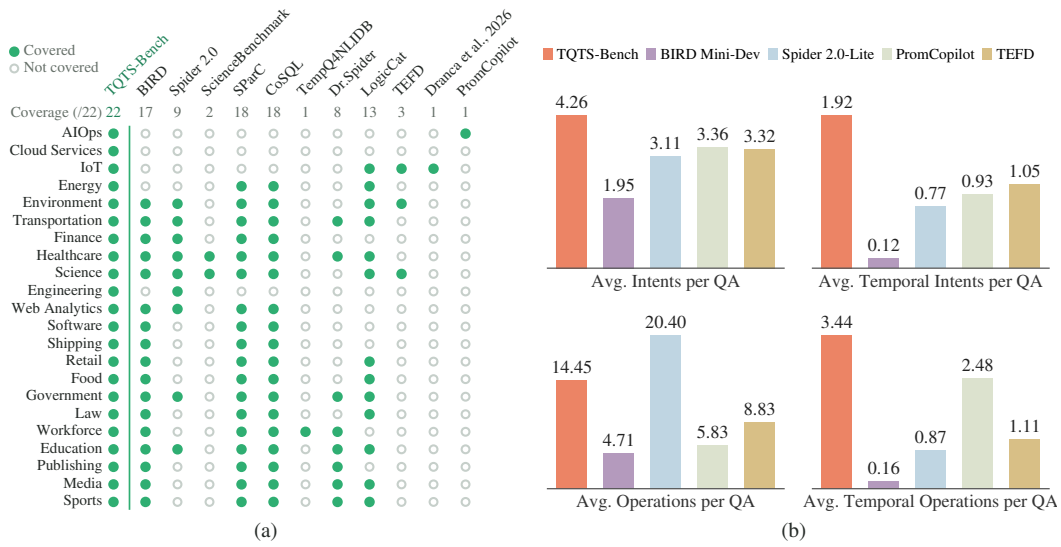


Figure 4: (a) Domain coverage and (b) time-specific query intent statistics of TQTS-BENCH.

4 EXPERIMENT

4.1 EXPERIMENTAL SETUP

Evaluation metric. We evaluate model performance using the widely adopted metric **Execution Accuracy (EX)** (Li et al., 2023; Lei et al., 2025), which measures whether the execution result matches the expected answer and thereby reflects the semantic correctness of the generated query.

Difficulty level. Given the diversity of query syntaxes, token count of the gold query for a question is not a suitable measure of difficulty. Therefore, we classify questions by the number of query intents: 1–3 as easy (31.72%), 4–5 as medium (44.33%), and 6–9 as hard (23.95%). This categorization reflects the increasing compositional complexity required to answer the question.

Human evaluation. Given the substantial time and cost involved, we conduct human evaluation on a randomly sampled 10% subset of TQTS-BENCH to establish a human performance baseline for real-world TSDB application scenarios. For details, refer to App. C.2.

LLMs. We evaluated a broad range of advanced LLMs, including both open-source and closed-source models. These models comprise representative and state-of-the-art offerings from major AI developers, enabling a comprehensive comparison between open-source and proprietary frontier models. The open-source models include Qwen3.8-Flash (Qiu et al., 2026), DeepSeek-V4-Pro (DeepSeek-AI et al., 2026), GLM-5.3-Flash (GLM-5-Team et al., 2026), and Kimi-K3 (Kimi Team et al., 2026), while the closed-source models include GPT-6-Sol (OpenAI, 2026), Claude-Opus-5 (Anthropic, 2026), and Gemini-3.7-Flash (Google DeepMind, 2026). Following the experiment setting of Lei et al. (2025), if the input length exceeds the model's maximum token limit, it will be truncated from the beginning.

Text-to-query methods. We also evaluate current state-of-the-art and representative text-to-query methods for both RDBs and TSDBs to examine their ability to address TSDB-specific challenges and transferability across different TSDB management systems. The RDB methods include DeepEye-SQL (Li et al., 2026), OpenSearch-SQL (Xie et al., 2025), RSL-SQL (Cao et al., 2024), DAIL-SQL (Gao et al., 2024), and DIN-SQL (Pourreza & Rafiei, 2023), while the TSDB methods include PromCopilot (Zhang et al., 2026). Although RDB methods are not specifically designed for TSDBs, they provide strong baselines because they address fundamental text-to-query challenges such as schema linking and query generation, which are also critical in TSDB applications. To ensure a controlled comparison and eliminate the impact of different backbone models, we use GPT-4o-mini as the unified LLM backbone for all methods.

4.2 QUANTITATIVE EVALUATION RESULTS

Table 1: Evaluation results on TQTS-BENCH. The best result (except human) is in **bold**, and the runner-up is underlined. Results with (*) are tested on a randomly sampled 10% subset.

Method	EX(↑)			
	Easy	Medium	Hard	**Overall**
Human performance	*95.71%	*85.52%	*81.94%	*87.34%
Open-Source Models				
Qwen3.8-Flash	43.75%	20.07%	17.79%	27.04%
DeepSeek-V4-Pro	39.89%	15.87%	11.79%	22.51%
GLM-5.3-Flash	54.30%	28.88%	19.15%	34.61%
Kimi-K3	58.26%	37.24%	34.08%	43.15%
Closed-Source Models				
GPT-6-Sol	62.74%	40.63%	35.38%	46.38%
Claude-Opus-5	62.79%	**42.91%**	**41.92%**	**48.98%**
Gemini-3.7-Flash	**63.25%**	37.68%	32.92%	44.65%
RDB Methods				
DeepEye-SQL	14.82%	1.92%	1.16%	5.83%
OpenSearch-SQL	16.37%	1.84%	1.09%	6.27%
RSL-SQL	11.99%	1.33%	0.95%	4.62%
DAIL-SQL	1.34%	0.26%	0.00%	0.54%
DIN-SQL	12.25%	2.17%	1.23%	5.14%
TSDB Methods				
PromCopilot	1.80%	0.22%	0.07%	0.69%

Existing RDB methods struggle with TQTS tasks and exhibit limited transferability to TSDBs. Tab. 1 reports the performance of state-of-the-art RDB methods on TQTS tasks. Specifically, DeepEye-SQL, a high-performing RDB method among those compared on the BIRD benchmark[2], achieves only 5.83% EX on TQTS-BENCH, a notably low score. Moreover, other RDB methods also perform poorly, with most achieving below 10.00% EX and some as low as 0.54%. This performance gap between RDB benchmark and TQTS-BENCH suggests that methods optimized for RDB text-to-query tasks do not necessarily generalize to TSDB scenarios. More detailed analysis of these results is provided in App. C.3.

Although LLMs outperform existing RDB methods, their performance remains unsatisfactory. Table 1 shows that advanced LLMs still achieve limited EX on TQTS tasks. The best-performing model, Claude-Opus-5 achieves the highest EX and consistently outperforms other models at the medium and hard difficulty levels. However, its overall performance remains relatively low, with an EX of only 48.98%, which is far below the human performance of 87.34%. This gap indicates that TQTS tasks remain challenging even for the most advanced models. The gap is more pronounced for some open-source models; for example, DeepSeek-V4-Pro achieves an EX of only 22.51%, leaving substantial room for improvement in practical applications. Compared with explicitly optimized RDB methods for text-to-SQL tasks, LLMs demonstrate stronger generalization ability, but their performance remains far from human-level capability.

Existing TSDB methods exhibit limited generalizability for TQTS tasks. Tab. 1 shows that PromCopilot, a state-of-the-art TSDB method for TQTS tasks, still achieves limited performance, with an EX of only 0.69%. PromCopilot is specifically designed for PromQL, the query syntax of Prometheus, and is primarily targeted in cloud services and AIOps application domains. As reported in Tab. 2, its performance decreases from 3.03% to 0.00% when the domain shifts to others. The result indicates that existing TSDB methods are sensitive to changes in application domains, revealing the limited generalizability of existing TSDB methods across different TQTS task settings. For detailed analysis, refer to App. C.4.

Table 2: Performance of PromCopilot on different domains.

Domain	EX(↑)
Targeted	3.03%
Others	0.00%

[2] https://bird-bench.github.io/

4.3 ERROR ANALYSIS

We perform a systematic error analysis by randomly sampling 600 examples and manually examining the incorrect cases. Given its weak performance, we select DeepSeek-V4-Pro for error analysis, as it may expose more representative failure modes in practical settings. Fig. 5 shows three identified representative error types: **query syntax errors**, **intent understanding errors**, and **schema linking errors**.

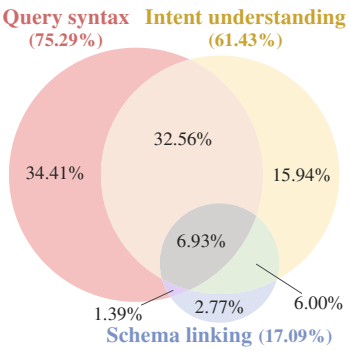


Figure 5: Proportions of incorrect cases by error type.

Query syntax errors (75.29%). Due to the diversity and complexity of TSDB query syntax, the model needs to understand various syntax structural patterns and function invocation rules to generate correct queries. We categorize query syntax errors into two types based on the type of syntax violation: (1) **Query structure errors (42.50%).** The model fails to construct valid query structures such as improper ordering of operators or functions, resulting in incorrect query results (e.g., Fig. 6(a)). (2) **Function/keyword usage errors (32.79%).** The model misuses functions or keywords, including incorrectly applying valid functions (i.e., misunderstanding parameters, or usage constraints) and invoking non-existent functions, thereby causing errors (e.g., Fig. 6(b)). For more error cases, refer to App. C.5.

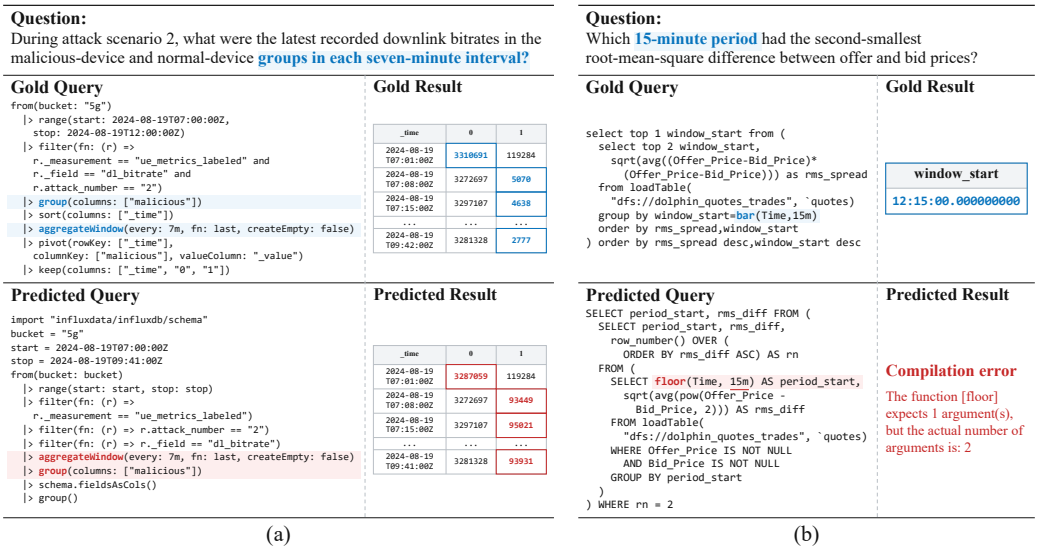


Figure 6: Examples of **query syntax errors**. (a) Query structure error: reversing `group` and `aggregateWindow` selects per-series rather than per-group last readings, producing incorrect results. (b) Function usage error: replacing `bar(Time, 15m)` with `floor(Time, 15m)` violates the single-argument requirement of `floor`, causing a compilation error.

Intent understanding errors (61.43%). For the TQTS task, accurately identifying the requirement of a question is particularly challenging due to the complexity of time-specific query intents. TQTSBENCH includes 9 query intents, 4 of which are time-specific and require models to correctly interpret and execute temporal operations. As shown in Fig. 7(a), errors involving time-specific query intents account for 65.41% of all intent understanding errors, substantially exceeding those associated with time-agnostic query intents (34.59%). Specifically, the error rates for the 4 time-specific intents are 26.31%, 25.19%, 7.52%, and 6.39%, respectively. These results suggest that current models struggle particularly with interpreting time-specific query intents. Fig. 7(b) shows one such failure, caused by missing window aggregation and resampling.

Schema linking errors (17.09%). Schema linking is a critical step in text-to-query tasks (Li et al., 2026). However, existing methods are mainly designed for RDBs and cannot be directly applied

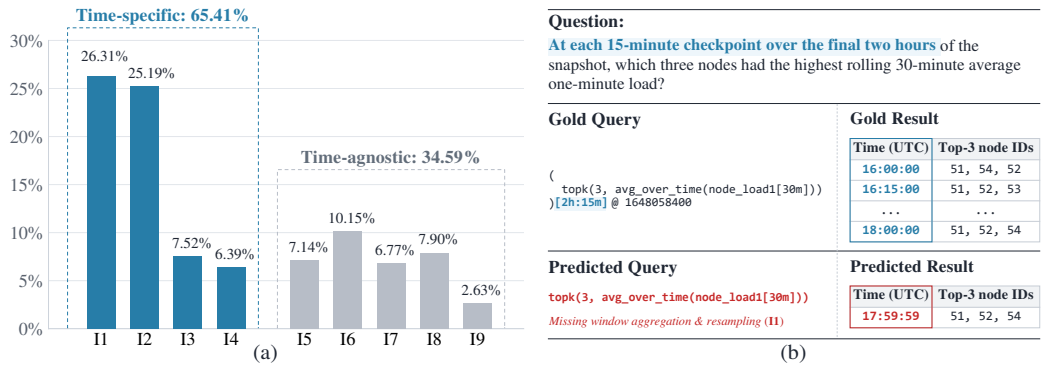


Figure 7: **Intent understanding errors**. (a) Distribution of primary erroneous intents; time-specific intents account for 65.41%. (b) Window aggregation & resampling error: the predicted query lacks `[2h:15m]`, returning a single top-3 result instead of results at each 15-minute checkpoint.

to TSDBs due to their different schema organizations. TSDBs typically organize data using metrics, labels, and tags rather than tables and columns, making traditional schema linking error types unsuitable for TSDBs. Moreover, TSDBs are widely used in IoT and AIOps domains, where high-frequency data collection leads to large-scale schemas. We identify two major challenges for schema linking in TQTS tasks: **(1) Diverse schema structures.** The heterogeneous use of metrics, labels, and tags introduces new schema ambiguities. For schema details, refer to App. A.4. **(2) Large-scale schema.** Cases with schema linking errors have much larger schemas (**381k tokens on average**) than other erroneous cases (115k tokens on average). Large schemas hinder schema comprehension and relevant element retrieval, and long contexts can further degrade LLM performance due to the "lost-in-the-middle" phenomenon (Liu et al., 2024). Therefore, improving schema linking performance in TQTS tasks remains an important direction for future work.

4.4 Ablation Study

To figure out why existing models perform poorly in TQTS-Bench, we conduct an ablation study. Although domains and query intents may also affect TSDB query generation, we focus on syntax because it is an intuitive and explicit difference between TSDBs and RDBs. Specifically, we convert the original TSDBs into RDBs (e.g., SQLite) while keeping the questions of error cases unchanged, and require the model to generate standard SQL queries. We compare the performance with two representative RDB benchmarks, BIRD and Spider 2.0-lite. As shown in Tab. 3, converting TSDBs into RDBs improves EX from 0.00% to 32.21%, which is lower than BIRD (56.91%) but higher than Spider 2.0-lite (22.94%). It suggests that the model's poor performance is due not to the difficulty of the questions but to the diverse query syntaxes. The model is capable of understanding the questions and generating effective SQL queries, but it lacks specific syntax knowledge, which leads to incorrect answers. For implementation details, refer to App. C.6.

Table 3: EX across different query syntaxes and benchmarks.

Setting	Syntax	EX(↑)
Original	Diverse	0.00%
w/ conversion	SQL	32.21%
BIRD	SQL	56.91%
Spider 2.0-lite	SQL	22.94%

5 Conclusion

In this paper, we introduce TQTS-Bench, a comprehensive benchmark for evaluating text-to-query capabilities over TSDBs. Unlike existing RDB benchmarks, TQTS-Bench is designed to capture TSDB-specific challenges by covering diverse query syntaxes, application domains, and time-specific query intents. Extensive evaluations of advanced LLMs and representative text-to-query methods show that existing methods still struggle with TQTS tasks, particularly in query syntax handling, intent understanding, and schema linking in TSDBs. Our analysis reveals key limitations of current methods and highlights opportunities for developing more effective and generalizable text-to-query methods for real-world time-series analysis.

AI USAGE DISCLOSURE

This work used large language models (LLMs) in three limited capacities: (1) to assess the coverage of manually collected related work; (2) to generate candidate QA pairs during benchmark construction, which were subsequently reviewed and finalized by human experts; and (3) to assist with grammar and phrasing refinement. LLMs were not used to generate experimental results, figures, or references. All cited works and final manuscript were verified and approved by the authors, who take full responsibility for the accuracy and integrity of this paper.

REFERENCES

Anthropic. Introducing claude opus 5. https://www.anthropic.com/news/claude-opus-5, July 2026.

Andreas Bader, Oliver Kopp, and Michael Falkenthal. Survey and comparison of open source time series databases. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband*, pp. 249–268. Gesellschaft für Informatik eV, 2017.

Zhenbiao Cao, Yuanlei Zheng, Zhihao Fan, Xiaojin Zhang, Wei Chen, and Xiang Bai. RSL-SQL: Robust schema linking in text-to-sql generation. *arXiv preprint arXiv:2411.00073*, 2024. doi: 10.48550/arXiv.2411.00073.

Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, Steve Ash, William Yang Wang, Zhiguo Wang, Vittorio Castelli, Patrick Ng, and Bing Xiang. Dr.Spider: A diagnostic evaluation benchmark towards text-to-SQL robustness. In *Proceedings of the International Conference on Learning Representations*, 2023.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, et al. DeepSeek-V4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Lacramioara Dranca, Pablo Donate, Julio A. Sanguesa, Piedad Garrido, Vicente Torres-Sanz, and Francisco J. Martinez. LLMs for industrial databases: An agro-food production plant use case. *Frontiers in Artificial Intelligence*, 9, 2026. doi: 10.3389/frai.2026.1764367.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-SQL empowered by large language models: A benchmark evaluation. *Proceedings of the VLDB Endowment*, 17(5):1132–1145, 2024. doi: 10.14778/3641204.3641221.

GLM-5-Team, Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, et al. GLM-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Google DeepMind. Gemini 3.7 Flash Model Card, August 2026. URL https://deepmind.google/models/model-cards/gemini-3-7-flash/.

Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. Time series management systems: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 29(11):2581–2600, 2017. doi: 10.1109/TKDE.2017.2740932.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y. Charles, et al. Kimi K3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows. In *Proceedings of the International Conference on Learning Representations*, 2025.

Boyan Li, Chong Chen, Zhujun Xue, Yinan Mei, and Yuyu Luo. DeepEye-SQL: A software-engineering-inspired text-to-sql framework. *Proceedings of the ACM on Management of Data*, 4 (3):1–28, 2026. doi: 10.1145/3802035.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can LLM already serve as a database interface? a BIg bench for large-scale database grounded text-to-SQLs. In *Proceedings of the Advances in Neural Information Processing Systems*, volume 36, pp. 42330–42357, 2023. doi: 10.52202/075280-1835.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

Tao Liu, Xutao Mao, Dixuan Zhang, Yifan Li, Haixin Liu, Lulu Kong, Jiaming Hou, Rui Li, Yunlong Li, Aoze Zheng, Zhiqiang Zhang, Zhewei Luo, Hongying Zan, Kunli Zhang, and Min Peng. LogicCat: A chain-of-thought text-to-SQL benchmark for complex reasoning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(36):29958–29966, 2026. doi: 10.1609/aaai.v40i36.40243.

OpenAI. Introducing gpt-6 sol and luna, September 2026. URL `https://openai.com/index/introducing-gpt-6-sol-and-luna/`.

Mohammadreza Pourreza and Davood Rafiei. DIN-SQL: Decomposed in-context learning of text-to-sql with self-correction. In *Proceedings of the Advances in Neural Information Processing Systems*, volume 36, pp. 36339–36348, 2023. doi: 10.52202/075280-1577.

Zihan Qiu, Zekun Wang, Xiao Li, Yanpeng Li, Yang Xu, Yixuan Wang, Huaqing Zhang, Rui Men, Bochao Mao, Chengruidong Zhang, Fan Zhou, Hao Luo, Haofeng Huang, Haoran Lian, Haoyan Huang, Hongqing Chen, Jianwei Zhang, Jing Xu, Junjie Wang, Langshi Chen, Liangyu Wang, Linlang Jiang, Man Yuan, Minmin Sun, Peng Jin, Siqi Zhang, Siyu Wang, Xingzhang Ren, Yakai Wang, Yi Zhang, Yiming Dong, Yizhong Cao, Yubo Ma, Yunfei Mao, Bo Zheng, and Dayiheng Liu. On the design of Qwen3.8-Next architecture: Evaluation, efficiency, and training stability, 2026. URL `https://arxiv.org/abs/2608.30320`.

Ngoc Phuoc An Vo, Octavian Popescu, Irene Manotas, and Vadim Sheinin. Tackling temporal questions in natural language interface to databases. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 179–187, 2022. doi: 10.18653/v1/2022.emnlp-industry.18.

Xuefeng Wu, Yuanfeng Song, Jiawei Wen, Dandan Huang, Raymond Chi-Wing Wong, and Haodi Zhang. TEFD: A benchmark for natural language to Flux query generation in time-series databases. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, pp. 10020–10031, 2026. doi: 10.1145/3770855.3817508.

Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. OpenSearch-SQL: Enhancing text-to-sql with dynamic few-shot and consistency alignment. *Proceedings of the ACM on Management of Data*, 3(3):1–24, 2025. doi: 10.1145/3725331.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pp. 3911–3921, 2018. doi: 10.18653/v1/D18-1425.

Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter Lasecki, and Dragomir Radev. CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing*, pp. 1962–1979, 2019a. doi: 10.18653/v1/D19-1204.

Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit, David Proctor, Sungrok Shim, Jonathan Kraft, Vincent Zhang, Caiming Xiong, Richard Socher, and Dragomir Radev. SParC: Cross-domain

semantic parsing in context. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pp. 4511–4523, 2019b. doi: 10.18653/v1/P19-1443.

Chenxi Zhang, Bicheng Zhang, Dingyu Yang, Xin Peng, Miao Chen, Senyu Xie, Gang Chen, Wei Bi, and Wei Li. PromCopilot: Simplifying Prometheus metric querying in cloud native online service systems via large language models. *ACM Transactions on Software Engineering and Methodology*, 2026. doi: 10.1145/3797910.

Yi Zhang, Jan Deriu, George Katsogiannis-Meimarakis, Catherine Kosten, Georgia Koutrika, and Kurt Stockinger. ScienceBenchmark: A complex real-world benchmark for evaluating natural language to SQL systems. *Proceedings of the VLDB Endowment*, 17(4):685–698, 2023.

A DETAILS OF TQTS-BENCH

In this section, we provide detailed information about TQTS-BENCH, including which TSDB management systems were selected for use, what the selected TSDBs are, and which domain they belong to. We also describe the query intents covered in TQTS-BENCH, including the identified intent types and their proportions.

A.1 IDENTIFYING TSDB MANAGEMENT SYSTEMS AND THEIR QUERY SYNTAX

We identified 23 TSDB management systems based on their popularity rankings on the TSDB leaderboard and their accessibility, and used them to build TQTS-BENCH. Note that several highly ranked TSDB management systems, such as kdb+, were not selected because they require users to apply for and obtain a license before they can be used, which limits their accessibility. As shown in Tab. 4, we present syntax usage examples for two query intents in each TSDB management system, along with the number of TSDBs included in our benchmark. Specifically, the examples illustrate how some systems provide specific functions and operators to help answer questions involving window aggregation & resampling (**I1**) or temporal change analysis (**I2**). The " / " indicates that no suitable example was identified for the query interface considered.

Table 4: Selected TSDB management systems and their temporal functions and operators.

TSDB management system	**Representative functions and operators**		**# TSDB**
	Window aggregation & resampling (I1)	**Temporal change analysis (I2)**	
InfluxDB OSS v2	`aggregateWindow()`	`derivative()`	17
InfluxDB 3 Core	`date_bin_gapfill()`	/	
Prometheus	`avg_over_time()`	`resets()`	13
TimescaleDB	`time_bucket_gapfill()`	`delta()`	8
DolphinDB	`resample()`	`ratios()`	5
Apache Druid	`TIME_FLOOR()`	`TIMESTAMPDIFF()`	4
QuestDB	`SAMPLE BY`	`datediff()`	5
TDengine	`STATE_WINDOW()`	`CSUM()`	4
Apache IoTDB	`GROUP BY SESSION()`	`TIME_DIFFERENCE()`	4
VictoriaMetrics	`rollup()`	`deriv_fast()`	3
Arc	`time_bucket()`	`REGR_SLOPE(v, t)`	2
GridDB	`TIME_SAMPLING()`	`TIMESTAMP_DIFF()`	4
M3DB	`summarize()`	`perSecond()`	1
CrateDB	`date_bin()`	`age()`	4
CnosDB	`date_bin()`	`idelta_right()`	3
ArcadeDB	`ts.timeBucket()`	`ts.delta()`	1
GreptimeDB	`avg_over_time()`	`increase()`	1
IBM Db2 Event Store	`window()`	/	4
Riak TS	/	/	2
BangDB	`ROLLUP 5`	/	3
Machbase Neo	`timewindow()`	`MAP_DIFF()`	5
OpenMLDB	`ROWS_RANGE`	`drawdown()`	3
openGemini	`GROUP BY time()`	`ELAPSED()`	1

A.2 IDENTIFYING TSDBS AND THEIR DOMAINS

In TQTS-BENCH, we manually identify a total of 97 high-quality real-world TSDBs for benchmark construction. Fig. 8 summarizes these TSDBs, along with their application domains, data volumes, and numbers of QA pairs. The inner ring denotes domains, while the outer ring denotes individual TSDBs. Each sector's angle represents its number of QA pairs, and its color intensity reflects the data volume of the corresponding database: darker shades indicate larger volumes, and lighter

shades indicate smaller ones. The inner ring covers domains such as cloud services, energy, and IoT, whereas the outer ring presents different TSDBs.

As shown in Fig. 8, TQTS-BENCH covers a wide range of real-world domain applications, especially some time-series-intensive domains, such as IoT, AIOps and cloud services. These domains typically generate and store high-frequency time-series data continuously, resulting in diverse data characteristics and large data volumes. The total data volume of TQTS-BENCH is **41.68 GB**, reflecting the large-scale and diverse nature of real-world time-series workloads collected from various application domains. To effectively and efficiently analyze these time-series data, many TSDB systems, such as InfluxDB, Prometheus, and TimescaleDB, have been widely adopted. The diversity of application domains, together with the different query syntaxes of TSDBs, highlights the significant challenges to real-world TQTS tasks.

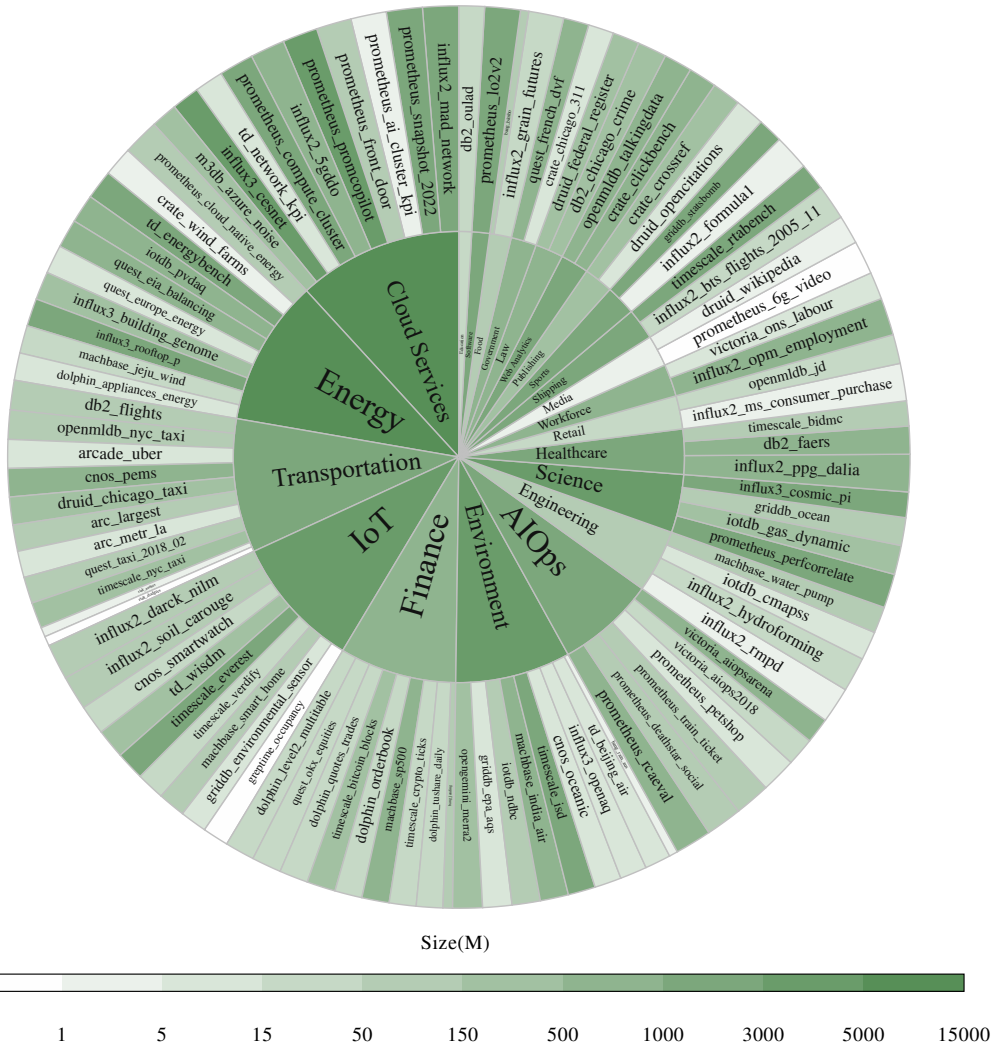


Figure 8: The domain and size of the TSDBs in TQTS-BENCH.

A.3 DETAILS OF QUERY INTENTS

This subsection details the query intents in TQTS-BENCH, including the identified query intent types, corresponding question examples, and the proportion of each query intent.

A.3.1 THE IDENTIFIED QUERY INTENTS

We collaborate with domain experts and identify 9 representative query intents in real-world scenarios, including 4 time-specific intents and 5 time-agnostic intents. The definitions of these query intents, along with their corresponding examples, are presented in Tab. 5.

Table 5: The definitions and examples of the nine query intents used in TQTS-BENCH.

Query intent	Definition	Example
Time-specific		
I1: Window aggregation & resampling	Resample or aggregate a time series over time windows or a new sampling grid, yielding results for each window or grid.	"What was Hankyung's average temperature forecast **for each day** of June 2017?"
I2: Temporal change analysis	Analyzing temporal changes in values within a single time series.	"What was the largest **one-second increase** in whole-home active power?"
I3: Time localization	Locate the time point(s) or interval at which an event or state occurs.	" **When** was the earliest weather observation in the database?"
I4: Relationship analysis	Analyzing relationships between two or more independently identifiable time series.	"On September 12, at which minutes did **Apple's trading volume exceed Amazon's** ?"
Time-agnostic		
I5: Conditional filtering	Filtering samples or events based on specified conditions.	"Return the full records for matches played **after June 15, 2024** ."
I6: Statistical computation	Computing statistical values from samples or events.	"What was the **average fare across all trips** ?"
I7: Sort & rank	Ranking candidates by one or more ordering criteria.	"Show the **first 100** sensor readings **in chronological order** ."
I8: Metadata query	Querying metadata about time-series databases, such as schema information.	"How is **each column** in the flight-record table **defined** in Db2?"
I9: Domain knowledge dependency	Requiring the use of specialized domain knowledge to answer the question.	"What were the 15-minute **OHLC prices** and traded share volume for each equity on March 27, 2025?"

A.3.2 STATISTIC OF QUERY INTENTS

Fig. 9 presents the statistics of query intents. To examine whether the distribution of query intents is unbiased with respect to query difficulty, we analyze the proportion of each intent across different difficulty levels, as shown in Fig. 9(a). Overall, each query intent is relatively evenly distributed across different difficulty levels, suggesting that query difficulty is not dominated by any particular intent type and that the benchmark maintains a balanced intent distribution across difficulty levels. Nevertheless, time-specific intents (**I1**–**I4**) occur more frequently in hard questions, and less in easy questions, indicating that temporal reasoning tends to introduce additional complexity.

Fig. 9(b) further shows that the overall distribution of query intents is consistent with real-world information-seeking demands. For example, **I5:** *Conditional Filtering* and **I6:** *Statistical Computation* account for the largest proportions, reflecting their prevalence in practical querying scenarios. We additionally investigate how the number of query intents varies with query difficulty, as shown in Fig. 9(c). As the difficulty increases, queries tend to involve more intents, with time-specific query intents showing the most pronounced increase.

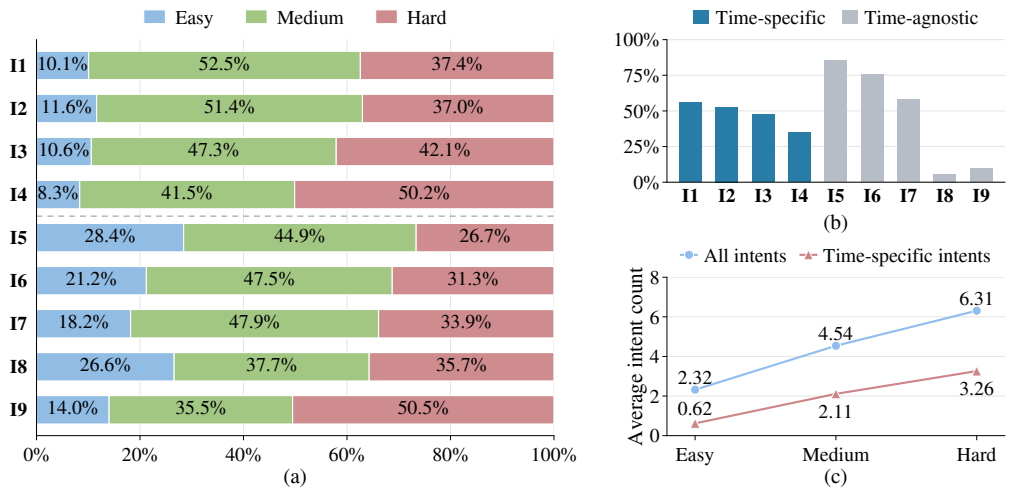


Figure 9: The statistics of query intents. (a) Distribution of query intents across different difficulty levels; (b) Distribution of query intents in TQTS-BENCH; (c) Average query intent count across different difficulty levels.

A.4 NON-UNIFIED SCHEMA STRUCTURES ACROSS TSDBS

Here, we illustrate the non-unified schema structures of TSDBs. TSDB schemas describe the logical organization of time-series data and can be broadly grouped into seven categories: **Relational tables**, **Time–dimension–metric tables**, **Tagged time-series table**, **Measurement-based organization**, **Metric–label series**, **Device tree** and **Property graphs**. Fig. 10 shows how the same time-series data are organized differently across these seven schema structures. The descriptions of these non-unified seven schema structures are as follows:

- **Relational tables** store records in named tables with typed columns. Each row contains attributes such as entity identifiers, timestamps, and values. Each table has a primary key that uniquely identifies records, while some columns may serve as foreign keys to reference records in related tables.
- **Time–dimension–metric tables** store event records, where each row represents an event associated with a datasource. Each row contains a primary timestamp, dimensions describing event attributes, and metrics representing quantitative values for analysis.
- **Tagged time-series tables** store timestamped observations, where each row represents an observation associated with an entity or series. Each row contains tags for identifying the entity or series and data columns recording observed values.
- **Measurement-based organization** stores observation points under different measurements within a data container. Each point contains a timestamp, tags describing the observation context, and fields recording measured values.
- **Metric–label series** represent time-series data as collections of timestamped samples. Each series contains a metric name, a set of labels identifying the series, and samples recording the metric values over time.
- **Device trees** store device and measurement data in a hierarchical structure. Each root-to-leaf path represents a time series, and each point along the path contains a timestamp and an observed value.
- **Property graphs** represent entities and observations as vertices connected by edges. Each vertex and edge contain properties that describe attributes such as timestamps and observed values.

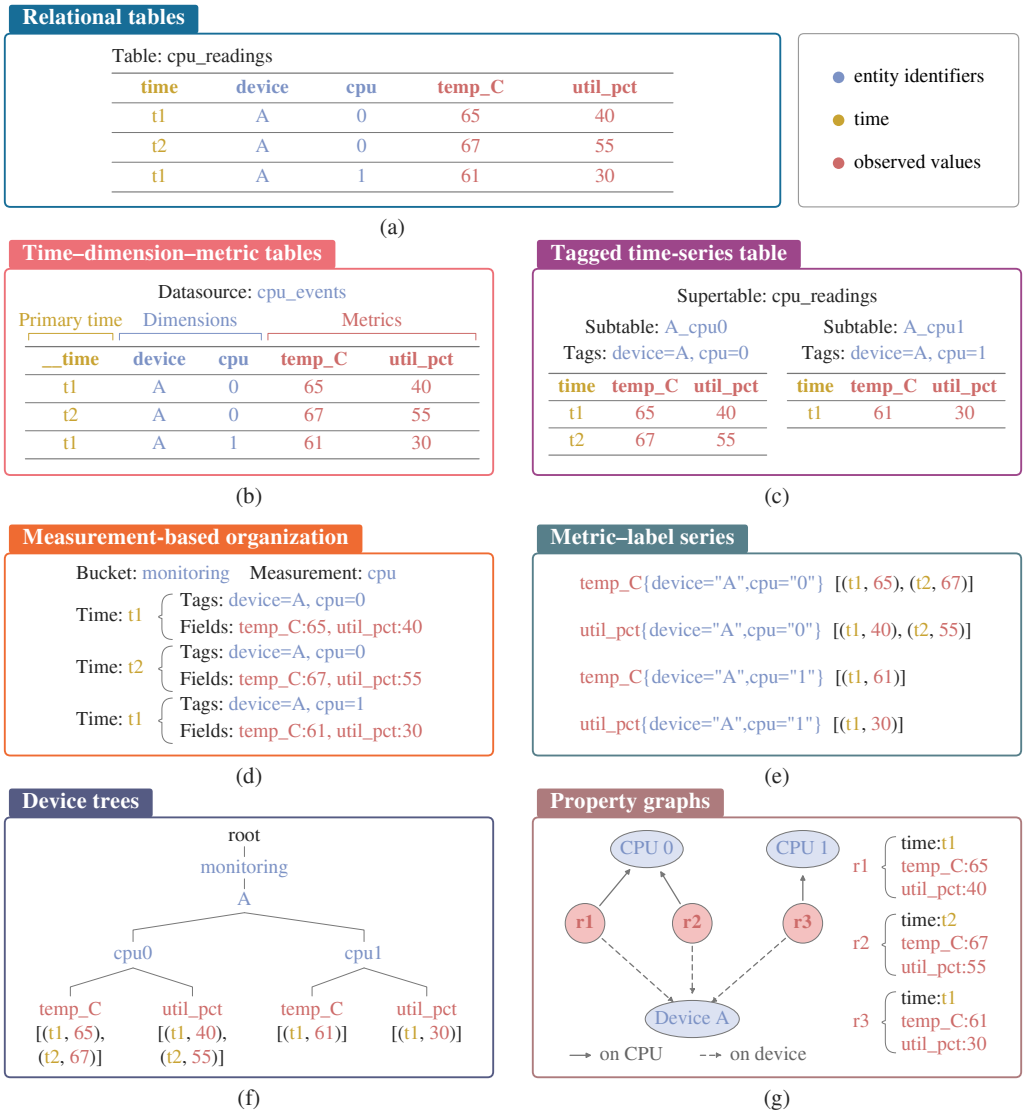


Figure 10: The seven types of schema structures of the same time-series data.

A.5 Details of the TSDB context

For each TSDB included in TQTS-Bench, we provide a customized context that captures its background information, key characteristics, and concrete usage scenarios. By incorporating this contextual knowledge, the model can gain a deeper understanding of the target TSDB and its application requirements, allowing it to generate questions that better reflect real-world TSDB usage scenarios and business needs.

For example, as shown in Fig. 11, the context of a Prometheus database contains descriptions of its background, including the fixed benchmark time setting, and the real-world cloud-service telemetry scenarios involving Kubernetes, node, application, Loki, PostgreSQL, and service metrics.

Database context used in TQTS-BENCH construction process


```
{
  "name": "Prometheus March 2022 cloud-service snapshot",
  "description": "Seven contiguous native Prometheus blocks
  containing real Kubernetes, node, application, Loki, PostgreSQL,
  and service telemetry from a cloud-service environment.",


  "engine": {
    "name": "prometheus",
    "version": "2.54.1"
  },
  "query_language": "PromQL: Prometheus Query Language for instant
  queries at the fixed benchmark current_time, using declared metric
  and label names. Guidance: The query language is PromQL, and
  Prometheus has no SQL tables.",


  "time": {
    "current_time": "2022-03-23T17:59:59Z",
    "timezone": "UTC",
    "ranges": [
      {
        "start": "2022-03-18T12:00:00Z",
        "end": "2022-03-23T18:00:00Z"
      }


    ]
  }
```


Figure 11: An example of the customized Prometheus database context used in TQTS-BENCH.

B DETAILS OF QA CONSTRUCTION

B.1 DESIGN OF VISUAL ANALYTICS ANNOTATION TOOL

Candidate QA pairs constructed with AI assistance may contain incorrect queries, unnatural wording, or mismatches between the intended request and the database context. To support systematic human verification before benchmark inclusion, we developed a visual analytics tool guided by three quality requirements. These requirements concern answer correctness, question naturalness and clarity, and alignment with query intents and TSDB context.

R1: Ensuring the correctness of executable answers. Reliable answers are essential for evaluating model performance, yet successful execution does not establish whether a query correctly answers its associated question. We therefore require independent human interpretations and execution evidence to assess answer correctness beyond execution success.

R2: Ensuring the naturalness and clarity of questions. AI-assisted construction requires explicit review of whether generated questions express meaningful requests in natural, comprehensible language. Awkward phrasing, ambiguity, and mechanical descriptions can obscure the intended meaning. The system must therefore help reviewers identify these issues and support targeted refinement before benchmark inclusion.

R3: Ensuring alignment with query intents and TSDB context. A fluent question and an executable answer do not necessarily constitute a suitable benchmark example. Each pair must faithfully represent the selected query intents, reflect a meaningful TSDB scenario, and be answerable from

the supplied context. Human review must therefore identify misalignment, missing information, and unstated assumptions.

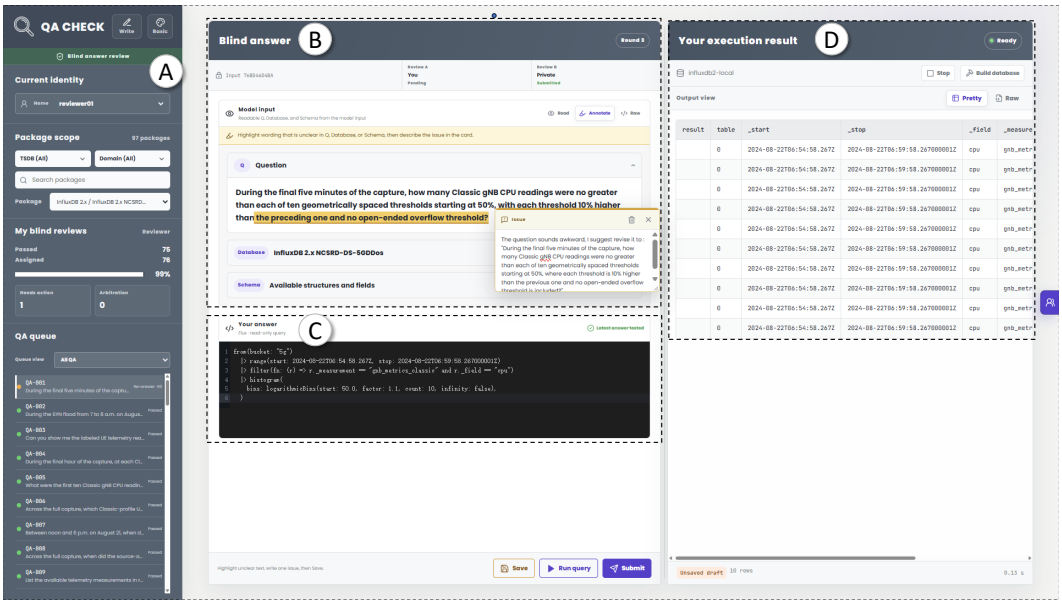


Figure 12: The visual interface for annotators in human-AI collaboration step.

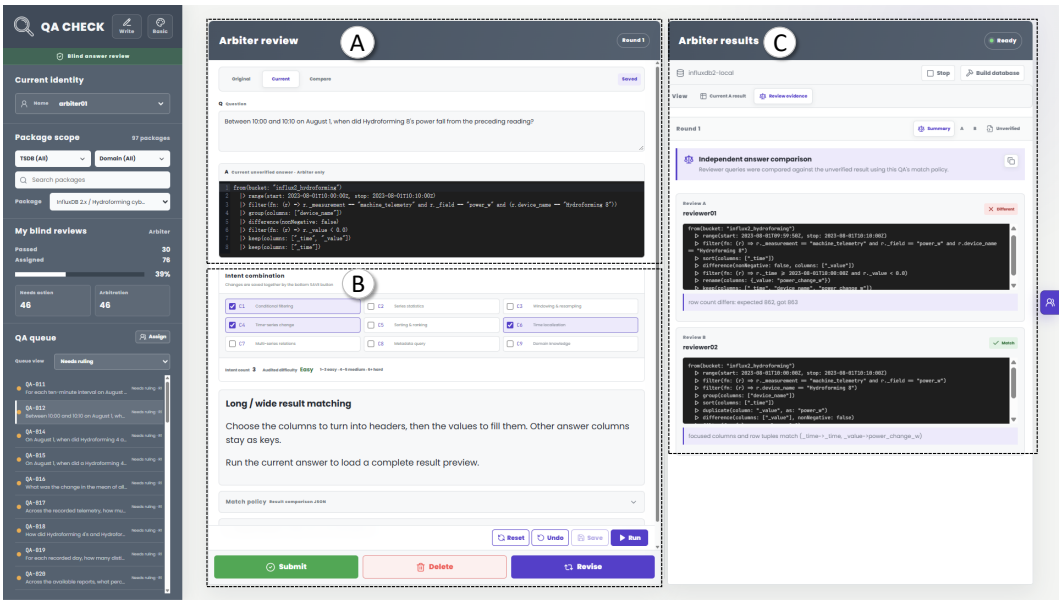


Figure 13: The visual interface for adjudicator in QA verification step.

B.2 Detail of QA verification

The verification workflow begins with two annotators independently reviewing each candidate QA pair. Fig. 12A enables each annotator to select TSDB and inspect current progress. Fig. 12B presents the question, database description, and schema while hiding the candidate answer and the peer annotator's solution. Annotators manually construct their own queries in the editor (Fig. 12C), providing independent answers to the questions and examining whether the available context supports answering it. The annotation interface in Fig. 12B allows them to flag unclear wording or insufficient

information. They can then inspect their queries' execution results and refine their answers before submission (Fig. 12D).

When verification reveals inconsistent results or an unanswerable question, an adjudicator examines the QA pair together with its intent combination, match policy, and annotator feedback. Fig. 13 shows the interface for adjudicator to review and revise the unverified QA pair. The adjudicator can inspect the QA pair (Fig. 13A), its corresponding query intents and can check the query result (Fig. 13B). The Review evidence view brings together both annotator queries and execution results for comparison (Fig. 13C). These views support assessment of query correctness alongside the question's wording, intended meaning, and contextual requirements.

The adjudicator revises the QA pair based on the identified issues, after which it undergoes another verification round. This process continues until both annotators independently obtain results consistent with the revised answer. The tool thus supports human review and correction of AI-assisted construction outputs, keeping interpretation, revision, and verification under human control.

The tool also records active review time and review-related actions, including query executions, annotations, review decisions, and arbitration operations. The Human activity dashboard summarizes these records across database packages and displays review progress. These records allow us to document the human effort involved in QA verification and provide the basis for the verification effort reported in the main text.

C More experiment detail and analysis

C.1 Evaluation criteria for TQTS-Bench

Here, we introduce the detailed criteria for the Execution Accuracy (**EX**) metric used in the experiment for TQTS-Bench. Inspired by Spider 2.0 (Lei et al., 2025), EX evaluates whether a predicted query produces the expected results, primarily by checking whether its output contains all columns returned by the gold query. However, in TSDBs, this situation becomes more complex, since models can often generate different queries that return the same underlying data in different formats, such as queries involving the `pivot` function.

As illustrated in Fig. 14, the absence of the `pivot` function results in a different output format between the predicted and gold results, causing the evaluation to incorrectly mark the prediction as incorrect. To address this issue, inspired by the matching policy in Spider 2.0, we introduce an additional transformation step to establish a more precise and suitable evaluation criterion for TQTS tasks. This policy reduces false negatives in evaluation while avoiding an increase in false positives.

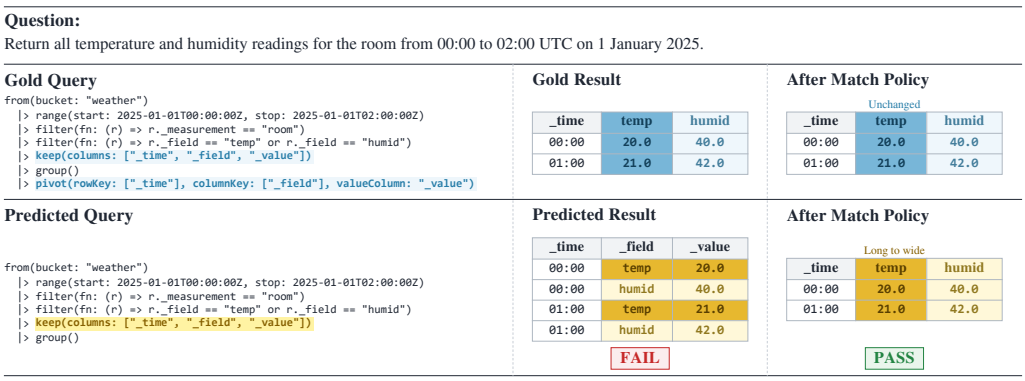


Figure 14: The evaluation criteria and an example of the match policy used in the evaluation.

C.2 Detailed Information for Human Evaluation

In the human evaluation, we utilize the developed visual analytics tool as the evaluation interface. Specifically, human experts are provided with the question, database schema, and TSDB context,

and are asked to manually write the corresponding queries. After completion, we evaluate the EX of the queries provided by human experts.

C.3 WHY THE RDB METHODS STRUGGLE WITH THE TQTS TASK

The experimental results demonstrate that existing RDB methods generally perform poorly on the TQTS task. To investigate the reasons behind their limitations, we select DeepEye-SQL, a representative and high-performing text-to-query method on BIRD, for further analysis.

For text-to-query tasks, DeepEye-SQL consists of four main stages: schema linking, query generation, query verification and refinement, and confidence-based selection. However, these stages are primarily designed for generating SQL queries. Specifically, as shown in Fig. 15, the prompts used in the query generation stage are heavily constrained by SQL syntax. Such SQL-specific guidance limits the model's ability to generate queries required by TSDBs, which often adopt different query syntaxes and syntactic structures.

As shown in Fig. 16, the generated queries may not conform to the syntax requirements of TSDB query syntax, leading to execution failures. Therefore, the SQL-specific design of existing RDB methods is a key reason why they struggle with the TQTS task, as it prevents them from effectively adapting to the diverse query syntaxes of TSDBs.

DeepEye-SQL: the original skeleton-based generation prompt

 # Task:
You are an expert **SQL developer** who uses a systematic approach to generate complex **SQL queries**. Your task is to analyze the given question and database schema, then generate a **SQL query** using a three-step process:
[...]
Step 1: Plan (SQL Components Analysis)
Analyze the question and identify:
- **SELECT clause**: What data needs to be retrieved? (columns, aggregations, calculations)
- **FROM clause**: Which tables are needed?
- **JOIN clauses**: What relationships need to be established?
- **WHERE clause**: What filtering conditions are required?
- **GROUP BY clause**: What grouping is needed for aggregations?
[...]
- **ORDER BY clause**: What sorting is required?
[...]
 # Important Rules:
[...]
2. **SQLite Compatibility: Use only SQLite-compatible functions and syntax**
[...]
 # Output Format:
Please respond with XML code structured as follows:
[...]
`<result>`
The final **SQL query** that answers the target question and **can be executed on the target SQLite database**, ensure there is not any SQLite comment and not any other explanation text in the SQL query.
[...]
`</result>`
[...]

Figure 15: An excerpt from the SQL generation prompt of DeepEye-SQL.

Failure case: PromQL required, SQL generated

Question

Which streamer containers are currently in the running state during the 200 MiB three-streamer test?

Gold query (PromQL)

```
kube_pod_container_status_running{
  dataset="prom_6g_video",
  bandwidth_mib="200",
  streamer_count="3"
} > 0
```

DeepEye predicted query (SQL)

```
SELECT DISTINCT k.streamer
FROM kube_pod_container_status_running AS k
WHERE k.bandwidth_mib = '200'
  AND k.streamer_count = '3' AND k._value = 1
```

Figure 16: An example where DeepEye-SQL generates SQL instead of the required PromQL.

C.4 Why the TSDB methods still struggle with TQTS task

In the experiment, TSDB methods still achieve limited EX on TQTS-Bench. To understand the reasons behind such poor performance, we conduct a further analysis of the state-of-the-art method, PromCopilot, a text-to-query method designed for Prometheus-based monitoring scenarios.

PromCopilot addresses the TQTS task through two stages. First, it constructs a knowledge graph to describe system contexts and facilitate schema linking. However, this knowledge graph is tightly coupled with Kubernetes cloud-native environments, where entities and relations are defined based on components such as pods, nodes, and services. Such a domain-specific schema makes it difficult to directly transfer the method to other domains. Second, PromCopilot generates queries using LLMs with carefully designed prompts.

As illustrated in Fig. 17(a), the system prompts are primarily tailored for cloud services and AIOps scenarios, further limiting their applicability beyond the original domain. These domain-specific designs hinder the generalization of PromCopilot to broader TSDB application domains, which may explain its limited performance on TQTS-Bench, where it achieves only 0.69% EX. This observation is also consistent with the authors' acknowledgment that PromCopilot relies on Kubernetes- and Prometheus-specific knowledge (Fig. 17(b)).

(a) The system prompt used for query generation

[System]

Task Description:

Given a question, please extract the descriptions of metrics mentioned in the sentence as well as the system component type it belongs to. System component types include **container**, **deployment**, **namespace**, **node**, **pod**, **replicaset**, **statefulset**, **service**, and **API**.

Answer Formats:

Your answer should be an array of tuples. Each tuple should contain two elements: the first is the description of the metric, and the second is the type of component it relates to.

[User]

Question: Show the total memory allocation for the pods of the admin user service on node k8s-node3 or k8s-node4 over the last hour.

[Assistant]

```
[("total memory allocation", "pod")]
```

[User]

Question: {query}

(b) The original statement in PromCopilot (Zhang et al., 2026)

5.2.2 *External Validity.* The threat to external validity mainly lies in the generalizability of our approach and the representativeness of the dataset.

The generalizability of our approach is under threat of dependency on Kubernetes and Prometheus. On the one hand, the knowledge graph schema in this paper is designed for Kubernetes-based cloud native online service systems and may not be suitable for all systems. This makes Prom-Copilot difficult to use directly in other types of system. On the other hand, the prompts, cots, and examples in PromCopilot are all designed for Prometheus's data model and PromQL. This makes it difficult to directly apply Prom-Copilot to other metric monitoring platforms. To integrate PromCopilot with other monitoring platforms, the prompt, cots, and examples need to be modified according to the platform's data model and DSL. If the data model is significantly different from Prometheus's, further modifications may be required to entities and relationships related to metrics in the knowledge graph, as well as the logic for metric knowledge retrieval.

Figure 17: The description of tightly coupled domains as reflected in (a) the prompt for query generation and (b) the statement of external validity in the original paper.

C.5 More Examples of Error Analysis

We provide additional examples for each error type identified in error analysis. Tab. 6 summarizes the error taxonomy and links each subcategory to representative error cases. The following figures present detailed examples, illustrating how these errors manifest in generated queries and how they affect the correctness of the final results. Here, for intent understanding errors, we only consider the errors with time-specific query intents.

Table 6: Overview of the error types and their representative examples.

Error Type	Subcategory	Representative Example(s)
Query syntax errors	Query structure errors	Refer to Fig. 6(a) and Fig. 18(a)
	Function/keyword usage errors	Refer to Fig. 6(b) and Fig. 18(b)
Intent understanding errors	**I1**: Window aggregation & resampling	Refer to Fig. 7(b)
	I2: Temporal change analysis	Refer to Fig. 19(a)
	I3: Time localization	Refer to Fig. 19(b)
	I4: Relationship analysis	Refer to Fig. 19(c)
Schema linking errors	/	Refer to Fig. 20(a) and Fig. 20(b)

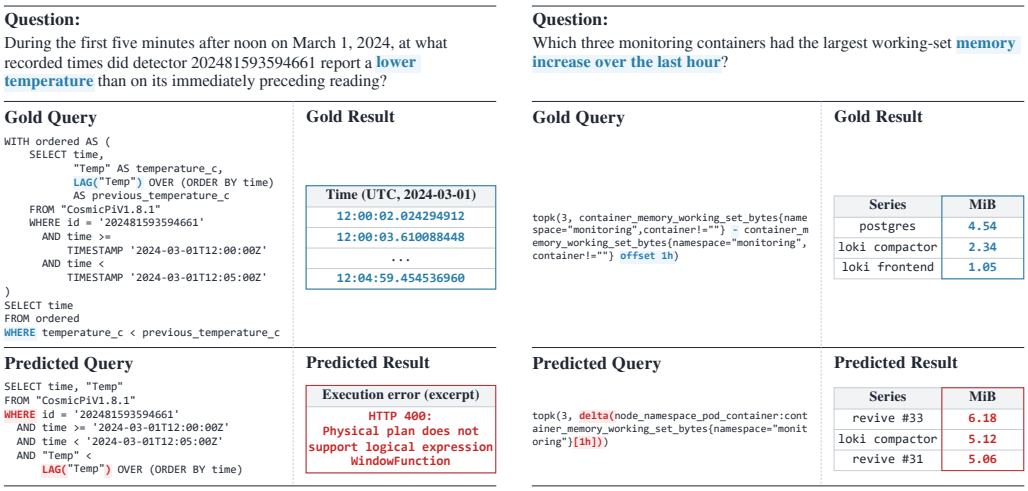


Figure 18: Examples of query syntax errors. (a) Query structure error: placing `LAG()` directly in the `WHERE` clause, rather than computing the preceding reading in a subquery, causes a window-function execution error. (b) Operator usage error: replacing subtraction with `offset 1h` by `delta(...[1h])` substitutes an extrapolated change for the difference between two time points; together with a changed metric, this produces incorrect values and rankings.

C.6 Details of the Ablation Study

Here, we introduce the implementation details of the ablation study in Sec. 4.4. Although query syntax errors account for 75.29% of all error cases, most of these cases involve two or more types of errors simultaneously. Therefore, to eliminate the interference from other error sources, we only select cases with syntax errors as the sole error type for the ablation study. For QA pairs containing only syntax errors, we map the TSDBs to RDBs and organize the schemas following the same format

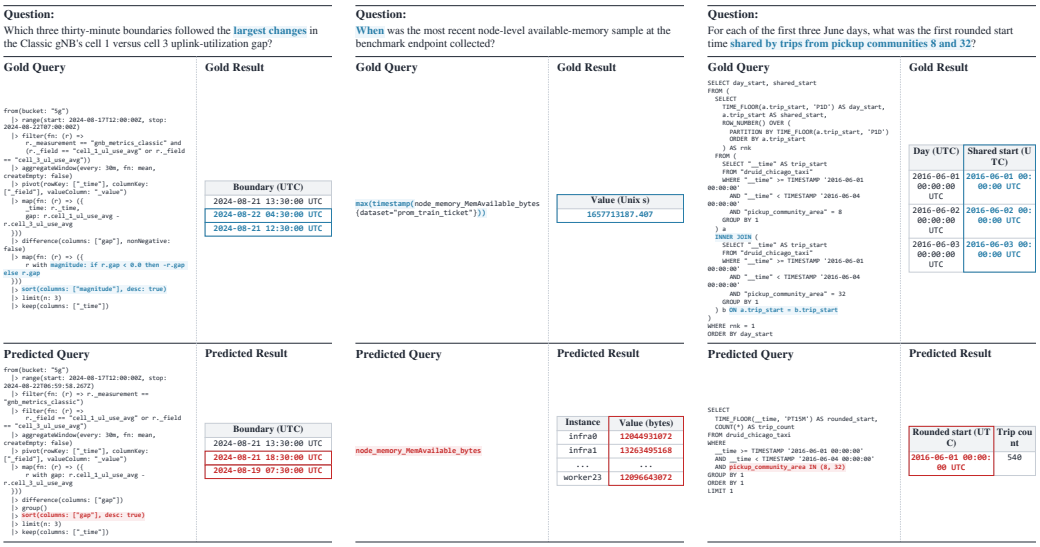


Figure 19: Examples of intent understanding errors. (a) Temporal change analysis error: ranking signed differences rather than absolute magnitudes interprets the largest changes as the largest increases, selecting incorrect time boundaries. (b) Time localization error: returning available-memory values instead of applying `max(timestamp(...))` reports memory quantities rather than the latest sample time. (c) Relationship analysis error: using `IN (8, 32)` instead of matching timestamps across both communities confuses union with intersection; a global `LIMIT 1` further returns a single time rather than the earliest shared time for each day.

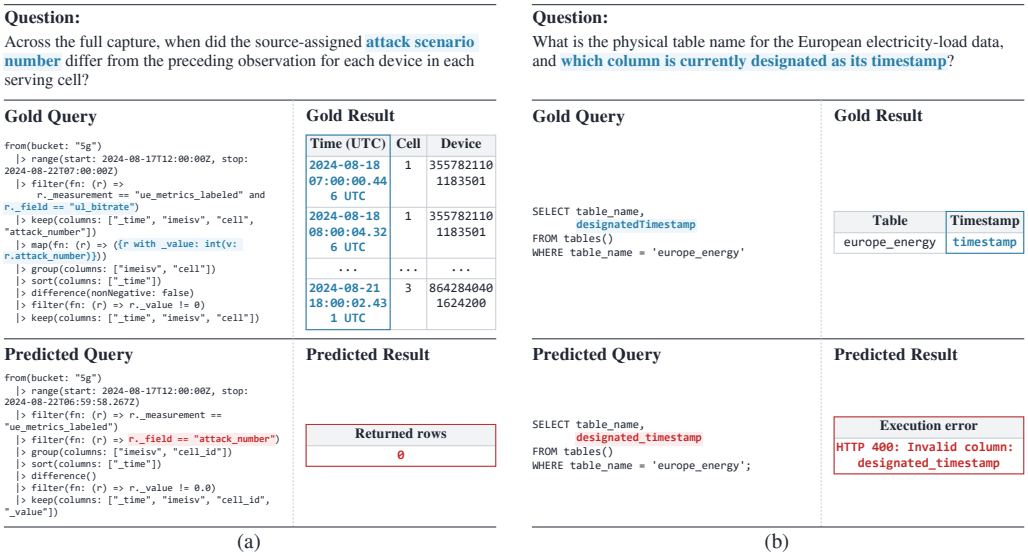


Figure 20: Two examples of schema linking errors. (a) The model treats the `attack_number` tag as a field and filters on `_field == "attack_number"`, which removes all relevant records and produces an empty result. (b) The model uses `designated_timestamp` instead of the metadata column `designatedTimestamp`, causing an invalid-column error and preventing retrieval of the designated timestamp column.

as BIRD. We then keep the original questions unchanged, manually annotate each question with a corresponding SQL gold query, and verify that the execution results are consistent with the original gold answers. Based on this converted setting, we evaluate the model and observe that the execution accuracy on these error cases improves from 0% to 32.21%, which falls between the performance levels on BIRD and Spider 2.0-lite. This suggests that the poor performance of the model is primarily due to the model's inability to handle TSDB query syntax rather than the difficulty of the questions.

To further demonstrate the difference in the model's proficiency between SQL and TSDB query syntax, we present a case study. As shown in Fig. 21, the model-generated Flux query fails due to a syntax error, whereas the generated SQL query successfully produces the correct result, despite the SQL gold query, containing 766 tokens, requiring a much more complex implementation than its Flux counterpart, containing 255 tokens. This suggests that the model still lacks sufficient capability in understanding and generating TSDB query syntax, such as Flux.

Question:

From March 5 through March 15, 2025, on which days did the number of distinct watt-valued channels with at least one positive reading differ from the preceding day, and by how many? Include metering inaccuracy and derived aggregate loads, and use only readings within this period.

Predicted Flux Query

```
from(bucket: "influx2_darck_nilm")
  |> range(start: 2025-03-05T00:30:00Z, stop: 2025-03-16T00:00:00Z)
  |> filter(fn: (r) => r._measurement == "apartment_power")
  |> filter(fn: (r) => r._field != "record_present")
  |> filter(fn: (r) => r._value > 0.0)
  |> keep(columns: ["_time", "_field"])
  |> group(columns: ["_field"])
  |> aggregateWindow(every: 1d, fn: count, createEmpty: false)
  |> group()
  |> keep(columns: ["_time", "_field"])
  |> group(columns: ["_time"])
  |> distinct(column: "_field")
  |> count(column: "_field")
  |> duplicate(column: "_stop", as: "_time")
  |> window(every: 1d)
  |> difference(nonNegative: false, columns: ["_value"])
  |> filter(fn: (r) => r._value != 0)
  |> keep(columns: ["_time", "_value"])
```

Predicted Flux Result

Wrong execution result

Execution error
Error: failed to execute query: 400 Bad Request: runtime error: column "_value" does not exist

Predicted SQL Query

```
WITH daily_channels AS (
    SELECT
        _time::date AS day,
        unnest(ARRAY[
            'main', 'lightbathroom', 'lighthallway', 'lightsherlock', 'lightkitchen', 'lightwatson', 'lightstoreroom',
            'fcob', 'fridge', 'sherlockalarmclocklight', 'sherlockcharger', 'sherlockfloorlamphue', 'sherlockguitaramp',
            'sherlockhairdryer', 'sherlockledstrip', 'sherlockmonitor', 'sherlockpc', 'sherlockserver', 'sherlocktv',
            'kettle', 'blender', 'kitchensoundsystem', 'microwave', 'watsoncharger', 'watsondesklamp', 'watsonfan',
            'watsonfloorlamp', 'watsonipadcharger', 'watsonlaptop', 'watsonledmap', 'watsonmonitor', 'watsonpiano',
            'watsonvinylplayer', 'printerscanner', 'router', 'vacuum', 'washingmachine', 'wheymixer', 'stoveoven13',
            'stoveplatel1', 'stoveplatel2', 'sherlocklaptop', 'lightlivingroom', 'livingfloorlamphue', 'sherlockglobe',
            'htpc', 'drill', 'airmattress', 'sherlockdesk', 'kitchencharger', 'solderingiron', 'mixer',
            'aggr_chargers', 'aggr_stoveplates', 'aggr_lights', 'inaccuracy'
        ]) AS channel,
        unnest(ARRAY[
            main, lightbathroom, lighthallway, lightsherlock, lightkitchen, lightwatson, lightstoreroom,
            fcob, fridge, sherlockalarmclocklight, sherlockcharger, sherlockfloorlamphue, sherlockguitaramp,
            sherlockhairdryer, sherlockledstrip, sherlockmonitor, sherlockpc, sherlockserver, sherlocktv,
            kettle, blender, kitchensoundsystem, microwave, watsoncharger, watsondesklamp, watsonfan,
            watsonfloorlamp, watsonipadcharger, watsonlaptop, watsonledmap, watsonmonitor, watsonpiano,
            watsonvinylplayer, printerscanner, router, vacuum, washingmachine, wheymixer, stoveoven13,
            stoveplatel1, stoveplatel2, sherlocklaptop, lightlivingroom, livingfloorlamphue, sherlockglobe,
            htpc, drill, airmattress, sherlockdesk, kitchencharger, solderingiron, mixer,
            aggr_chargers, aggr_stoveplates, aggr_lights, inaccuracy
        ]) AS watts
    FROM apartment_power
    WHERE _time >= '2025-03-05' AND _time < '2025-03-16'
),
positive_channels AS (
    SELECT day, channel
    FROM daily_channels
    WHERE watts > 0
    GROUP BY day, channel
),
daily_counts AS (
    SELECT day, COUNT(DISTINCT channel) AS distinct_channels
    FROM positive_channels
    GROUP BY day
),
with_prev AS (
    SELECT
        day,
        distinct_channels,
        LAG(distinct_channels) OVER (ORDER BY day) AS prev_count
    FROM daily_counts
)
SELECT
    day,
    distinct_channels - prev_count AS difference
FROM with_prev
WHERE prev_count IS NOT NULL
  AND distinct_channels <> prev_count
ORDER BY day;
```

Predicted SQL Result

Correct execution result

day	difference
2025-03-06	-6
2025-03-07	4
...	...
2025-03-15	-1

Figure 21: A case study on the differences in LLM capabilities between generating Flux for TSDBs and SQL for RDBs.

D PROMPTS USED IN TQTS-BENCH CONSTRUCTION PROCESS

The system prompt used for question generation in TQTS-BENCH construction process

[System]

Task:

You are an assistant for constructing natural-language questions over time-series databases. Using the query intents, target intent combination, database schema, and database context below, generate one natural-language question (Q) for the target time-series database. In this stage, generate Q only; do not generate a query statement (A) or query results.

Inputs:

Query intents:

A question may involve one or more intents. I1–I4 are time-specific intents, and I5–I9 are time-agnostic intents.

I1 – Window aggregation and resampling: Resample or aggregate a time series over time windows or a new sampling grid, yielding results for each window or grid.

I2 – Temporal change analysis: Analyze temporal changes in values within a single time series.

I3 – Time localization: Locate the time point(s) or interval at which an event or state occurs.

I4 – Relationship analysis: Analyze relationships between two or more independently identifiable time series.

I5 – Conditional filtering: Filter samples or events based on specified conditions.

I6 – Statistical computation: Compute statistical values from samples or events.

I7 – Sort & rank: Rank candidates by one or more ordering criteria.

I8 – Metadata query: Query metadata about time-series databases, such as schema information.

I9 – Domain knowledge dependency: Require the use of specialized domain knowledge to answer the question.

Target intent combination for this generation:

{`TARGET_INTENT_COMBINATION`}

Database schema:

{`DATABASE_SCHEMA`}

Database context:

{`DATABASE_CONTEXT`}

Generation constraints:

1. The question must express a real, meaningful query need for the target database and accurately reflect the specified target intent combination. Include an intent only when it is genuinely required by the question.

2. Use only entities, data meanings, and domain conventions supported by the schema and database context, so that the question can be answered using the target database.

3. Express the user's information need in natural, clear, and unambiguous English. Do not mechanically turn physical field names or a query statement into a question, and do not describe query implementation steps.

4. Do not combine unrelated objectives merely to satisfy the target intent combination. Do not assume schema elements, data meanings, domain knowledge, or database capabilities that were not provided.

5. Do not generate tasks such as forecasting, anomaly detection, or pattern clustering that require separate algorithmic models and cannot be completed by querying the target TSDB alone.

Output format:

Return exactly one JSON object, with no explanation or Markdown code fence. The `intents` array must list the intents actually used by Q and must match the target intent combination:

[User]

Target intent combination: {`I1, I5`}

Database schema: {schema}

Database context: {context}

[Assistant]

```
{
  "question": "What were the available-memory readings for farm140105,
  farm140108, and farm140109 at each fifteen-minute checkpoint
  during the final hour?",
  "intents": ["I1", "I5"]
}
```

[User]

Target intent combination: {TARGET_INTENT_COMBINATION}

Database schema: {DATABASE_SCHEMA}

Database context: {DATABASE_CONTEXT}

The system prompt used for query generation in TQTS-BENCH construction process

[System]

Task:

You are an assistant for constructing time-series database queries. The natural-language question (Q) and its intent labels below have been finalized. Using Q, the target database type, database schema, and database context, generate a query statement (A) that correctly answers Q. Do not change the meaning of Q.

Inputs:

Finalized Q:

{REVIEWED_QUESTION}

Confirmed intent labels:

{REVIEWED_INTENTS}

Target database type:

{TARGET_TSDB}

Database schema:

{DATABASE_SCHEMA}

Database context:

{DATABASE_CONTEXT}

Generation constraints:

1. Generate one query that can actually be executed in {TARGET_TSDB}. Use the query syntax supported by that database type; do not default to the syntax of another database system. The query must fully and accurately express the goal, conditions, and temporal semantics of Q, and its result must answer Q.

2. Use only objects that actually exist in the schema and data meanings and domain conventions provided in the database context. Do not invent fields, data, or query results.

3. Do not add analysis goals, conditions, or result content that Q does not request, and do not change Q to accommodate a particular query formulation.

4. If the target database is accessible, execute A, check that execution succeeds, that the result is non-empty and valid, and that the result answers Q; revise A if the check fails. Unless Q explicitly asks for a nonexistent entity or an empty result, an empty result does not count as a verified answer. If the database is inaccessible, do not claim that A was executed or verified.

5. If Q conflicts with the schema or database context, or if information needed for a correct query is missing, do not guess a query statement.

Output format:

Return exactly one JSON object, with no explanation or Markdown code fence. `question` must reproduce Q exactly, `intents` must match the confirmed intent labels, and `answer` must contain a query for the target database type. Record only evidence obtained from an actual execution in `execution_evidence`; if the query was not run, set `executed` to `false` and do not invent results.

[User]

Finalized Q: {`question`}

Confirmed intent labels: {`I1, I5`}

Target database type: {`TSDB`}

Database schema: {`schema`}

Database context: {`context`}

[Assistant]

```
{
  "target_tsdb": "Prometheus",
  "question": "Show the total memory allocation for the pods of
  the admin user service on node k8s-node3 or k8s-node4
  over the last hour.",
  "intents": ["I1", "I5"],
  "answer": "<executable query for the target database>",
  "execution_evidence": {
    "executed": false,
    "non_empty": null,
    "result_summary": null
  }
}
```

[User]

Finalized Q: {`REVIEWED_QUESTION`}

Confirmed intent labels: {`REVIEWED_INTENTS`}

Target database type: {`TARGET_TSDB`}

Database schema: {`DATABASE_SCHEMA`}

Database context: {`DATABASE_CONTEXT`}