Declarative Techniques for NL Queries over Heterogeneous Data

Elham Khabiri, Jeffrey O. Kephart, Fenno F. Heath III, Srideepika Jayaraman
Fateh A. Tipu, Yingjie Li, Dhruv Shah, Achille Fokoue, Anu Bhamidipaty
IBM Research, Yorktown Heights, NY 10598 USA

Abstract

In many industrial settings, users wish to ask questions in natural language, the answers to which require assembling information from diverse structured data sources. With the advent of Large Language Models (LLMs), applications can now translate natural language questions into a set of API calls or database calls, execute them, and combine the results into an appropriate natural language response. However, these applications remain impractical in realistic industrial settings because they do not cope with the data source heterogeneity that typifies such environments. In this work, we simulate the heterogeneity of real industry settings by introducing two extensions of the popular Spider benchmark dataset that require a combination of database and API calls. Then, we introduce a declarative approach to handling such data heterogeneity and demonstrate that it copes with data source heterogeneity significantly better than state-of-the-art LLM-based agentic or imperative code generation systems. Our augmented benchmarks are available to the research community.

1 Introduction

In many industrial settings, users wish to ask questions whose answer may be derived from structured data sources such as a spreadsheets, databases, APIs, or a combination thereof. Often, users don't know whether the answer exists anywhere in the system, or if so how to identify or access the right data source. In recent years, a variety of applications and research efforts have been developed to address this issue. As a result, multiple benchmarks (Yu et al., 2018; Zhong et al., 2017; Li et al., 2023a) and systems (Gao et al., 2023; Deng et al., 2025) have been developed to improve the capability of LLMs to successfully convert natural language questions into SQL queries against a database (the text-to-SQL problem). Many other benchmarks (Patil et al., 2023; Li et al., 2023b) and approaches (Jha et al., 2025; Prabhakar et al., 2025) have targeted the ability of LLMs to sequence and invoke the right APIs to answer a user's question.

Such benchmarks have driven steady progress towards ready and intuitive access to proprietary sources of information via either SQL or API calls, but they fail to test the ability of systems to cope with a combination of the two. In practical applications, one frequently encounters questions like "Which Xylem pumps at Bedford have experienced anomalous temperatures today?", which requires combining a database call to retrieve pumps with the right manufacturer and location with an API call to sense or compute temperature anomalies.[1]

In this paper, we explore the problem of data heterogeneity. While existing agent-based architectures (e.g., ReAct (Yao et al., 2022)) can dynamically orchestrate retrieval and aggregation of information across APIs and databases, they tend to be brittle, expensive to run, and difficult to scale in production. A core limitation of such approaches is that they conflate representation of the user's intent with planning an efficient execution sequence into a single step that is handled directly by LLMs.

We present a more practical architecture that retrieves and aggregates data from databases and APIs by cleanly separating a representation of the user's intent from planning an efficient execution sequence. We rely on SQL as a declarative language that expresses the user's intent and use User Defined Functions (UDFs) to invoke APIs from within SQL queries. By leveraging the UDF capability of modern database systems (e.g., postgres or DB2) to invoke external APIs, we place APIs on the same footing as database tables. In so doing, we leverage decades of research in SQL query optimization and planning for efficient orchestration and aggregation across both database tables and APIs (through their corresponding UDFs). We also

[1] Additional examples of such questions are provided in Appendix A.

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track, pages 1744–1761
November 4-9, 2025

explore an imperative approach that uses an LLM to generate imperative python code that stitches together information from heterogeneous sources and then executes the generated code.

Since we are unaware of any existing benchmark against which we can compare our declarative approach against imperative or agent-based approaches, we have created two new benchmarks consisting of questions whose answers require a combination of database and API calls, both of which are augmentations of the popular Spider dataset and benchmark. Benchmark I replaces a fraction of the real Spider database tables with equivalents that are executed via APIs. This allows us to directly test the mechanism by which database and API calls are combined without having to change the questions or their ground-truth answers from the original Spider benchmark. Benchmark II introduces a new set of scalar APIs that perform simple lexical, numeric, or geo-spatial operations. From a subset of two dozen Spider databases, we transform questions from the original Spider database into new questions that require interleaving database operations with compositions of 1-3 scalar APIs. We establish a set of corresponding ground-truth answers through a semi-automated process that generates over 2300 human-vetted question/answer pairs.

In the remaining sections, we briefly survey related work, detail our implementation of the systems that we are comparing, introduce the benchmark datasets, and summarize experiments that establish that our proposed approach outperforms imperative and agent-based approaches in several key metrics. Our main contributions include:

1. Two new benchmarks for assessing the ability of LLM-based systems to cope with data source heterogeneity;[2]

2. A declarative approach that uses SQL statements to represent user intent and leverages User-Defined Functions (UDFs) to place external APIs on the same footing as database tables, allowing them to be manipulated by standard database execution engines; and

3. Empirical evidence that our approach significantly outperforms an imperative code generation approach and an agent-based approach that combines state-of-the-art Text-to-SQL and API calling tools.

[2]These are available at https://huggingface.co/datasets/ibm-research/SQL-API-Bench.

2 Related work

We segregate previous work into three categories: structured data retrieval, LLM-based tool-calling, and coping with data heterogeneity. While there is much good work on systems that cope with Text-to-SQL and tool/API-calling individually, and some initial efforts to address data heterogeneity, there remain significant gaps in techniques that bridge across heterogeneous data sources and benchmarks that measure their efficacy.

2.1 Structured data retrieval

Popular large-scale Text-to-SQL benchmark datasets consisting of thousands of pairs of questions and their corresponding SQL ground truth include Spider (Yu et al., 2018), Spider-2 (Lei et al., 2025), and BIRD (Li et al., 2024). Enterprise deployment of Text-to-SQL systems faces significant challenges, as they must handle massive schemas containing over 1,000 columns, support multiple SQL dialects, and accommodate complex analytical requirements including data transformations and advanced analytics. Recent work like ReFoRCE (Deng et al., 2025) and DAIL-SQL (Gao et al., 2023) have addressed many of these issues and achieved top performance on the Spider Text-to-SQL benchmark. However, these techniques apply strictly to structured database queries, and cannot handle user requests requiring external API calls.

2.2 LLM-based tool calling

The Berkeley Function-Calling Leaderboard (BFCL) (Patil et al., 2024) maintains a leaderboard for state-of-the-art systems in the tool-calling task. Toolformer (Schick et al., 2023) and ToolLLM (Kojima et al., 2023) use LLMs to select and invoke the right APIs from an available set. (Elder et al., 2025) use the *structure* of SQL to select and orchestrate APIs according to their primary role, e.g. selection or filtering, but the end result is only a set of structured API calls that do not access databases. NESTFUL (Basu et al., 2025) presents a benchmark for evaluating LLMs on nested sequences of API Calls, which provides a harder task than calling individual APIs, but it sticks to the structured nature and singular modularity of APIs. XLAM (Zhang et al., 2024) contains a set

of fine-tuned LLMs that choose and execute API calls. Currently leading the BCFL leaderboard, it is regarded as the state-of-the-art in multi-turn tool calling. However, as it is specifically trained for a structured, function-calling environment, its support for multi-modal reasoning is limited. Moreover, changes to the set of tools or their interfaces requires re-training or fine-tuning (Lin et al., 2024).

2.3 Handling heterogeneous data sources

NL2Code has emerged as an efficient way to handle complex workflows, including data retrieval from multiple sources. CodeAct (Wang et al., 2024) generates executable Python code that serves as a unified action space for combining tool use, control flow, and data handling. However, typically, such systems fail in complex, heterogeneous scenarios that require multi-step reasoning workflow to combine the data.

BLENDSQL (Glenn et al., 2024) is a hybrid dataset that introduces new SQL extensions (LLM functions) that support hybrid question and answering across structured data (tables) and unstructured data. In contrast, we rely on the strength of existing SQL dialects that support user-defined functions. We believe that our approach could achieve the same effect as BLENDSQL without requiring special extensions, simply by positioning the unstructured data retrieval APIs as UDFs.

3 Approaches to handling heterogeneity

Here we describe three approaches for handling NL queries over heterogeneous data sources. The Declarative and Imperative NL query strategies use LLM-based code generation, while the Agentic query strategy is a ReAct agent-based solution that relies on specialized tools to access APIs and databases. For consistency, and due to its proven effectiveness as a component of Text-to-SQL systems (Gao et al., 2023; Deng et al., 2025), all three query strategies employ the Mistral-Large LLM (Albert Jiang, Jul 24, 2024).

3.1 Declarative approach

This section describes the components of our declarative *siwarex* framework, which leverages two data source schema that can be provided by users or derived semi-automatically from domain metadata:

1. The *Abstract Schema*, in the form of an Entity-Relationship Diagram, provides a global view of the data source properties and interrelationships in a format that is agnostic to whether the data source is a database table or an API.

2. The *API Mapping Schema* provides information necessary to invoke an API call, such as the URL, the method (POST, GET, etc.), and details of the input and output parameters.

As illustrated in Figure 1, the Abstract Schema and API Mapping Schema are generated either manually or via an automated process that leverages database schemas, OpenAPI specs, or other metadata. A deterministic offline process automatically converts the Abstract Schema into a relational schema that is used at runtime to generate SQL queries from NL questions. In that relational schema, tables corresponding to APIs (e.g., cascade) are *virtual tables*, each of which is associated with a corresponding User Defined Function (UDF) that invokes the associated API by consulting details provided in the API Mapping. An example UDF and its associated API wrapper are illustrated in Appendix B.

siwarex includes two key components:

- A standard **Text2SQL** module that, given the relational schema generated from the Abstract Graph (i.e., with virtual tables) and a user's NL question, generates a corresponding SQL. An example is provided in Appendix C.1.

- A rule-based **Query Rewriter** that bridges the physical and logical representation of data entities. It rewrites LLM-generated SQL containing *virtual tables* into executable SQL by replacing *virtual tables* with their corresponding UDFs, and (based on static SQL analysis) passes the right arguments.

We also implemented a variant of the Declarative method called **Declarative2** that can be used in situations where the UDFs represent scalar functions. Instead of using a query rewriter and treating UDFs as virtual tables, each scalar API is wrappered as a UDF and called directly in a manner analogous to built-in SQL functions like LENGTH. An example is provided in Appendix C.2. The resultant SQL expression tends to be easier to understand than that produced by **Declarative**, but its applicability is more limited. It is only possible to evaluate it for Benchmark II (described in Section 4.2).

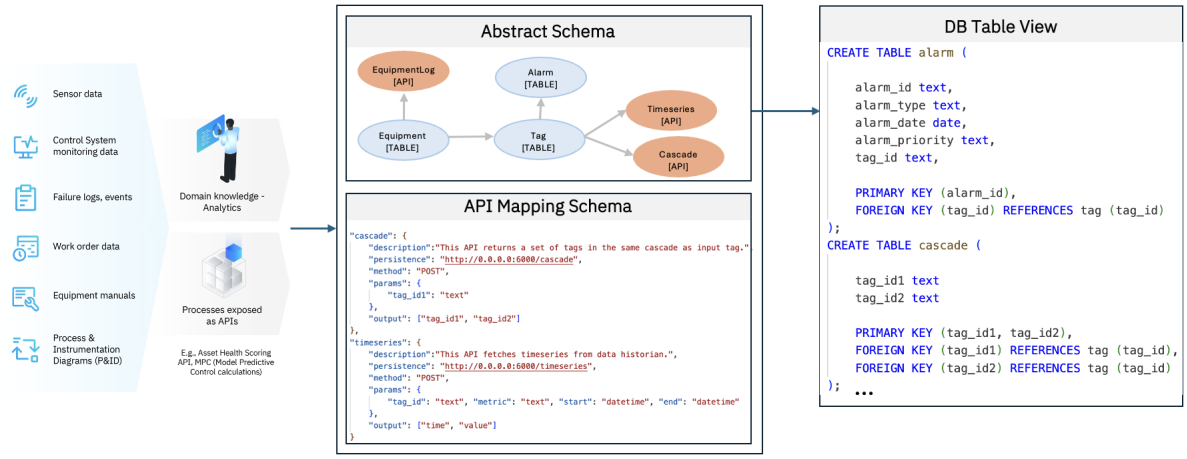

Figure 1: **Example of schemas and table view used by *siwarex*.** The Abstract Schema and API Mapping Schema required by *siwarex* can be provided manually or extracted from domain metadata. If a database schema is provided, the Abstract Schema can be extracted from it automatically; likewise the API Mapping Schema can be extracted automatically from an OpenAPI spec. For systems that mix DB access and API calls, the edges between API and DB nodes in the Abstract Schema may be augmented by a minimal amount of expert knowledge. Once the Abstract Schema is created, a relational schema (DB Table View) is generated from it automatically. The DB Table View represents all entities consistently as tables regardless of whether they are actually tables or APIs.

3.2 Imperative approach

We implemented an imperative approach that generates for each natural language question a Python program that choreographs the various database and API calls that are required to answer it. The relevant database schema are extracted dynamically from the database, and the relevant API specifications obtained. This information, along with the input question and sample rows from each table, is included in an LLM prompt that generates a Python program that is then executed to return the answer. An example of such an auto-generated Python program can be found in Appendix C.3.

3.3 Agentic approach

We implemented an agentic approach that uses ReAct (Yao et al., 2022) reasoning to call various tools to answer a given NL question, including:

1. SQLDatabaseToolkit, a Langchain toolkit that interacts with SQL databases, which we configure to use the Mistral-Large LLM (Albert Jiang, Jul 24, 2024).

2. xLAM (Prabhakar et al., 2025), an API-calling tool at the top of Berkeley API Leader board as of mid-2025.[3] From a set of API metadata provided to this tool, it selects those most likely to help answer the question.

[3] gorilla.cs.berkeley.edu/leaderboard.html

ReAct is widely regarded as a standard for evaluating tool-calling performance of LLMs (Kim et al., 2024; Fu et al., 2024; Basu et al., 2025).

The agent is provided with relevant metadata that includes the names and descriptions of available APIs and table schema. On that basis, it generates the sequence of APIs and database queries that must be executed to answer the given question. An example chain-of-thought trace is provided in Appendix C.4.

4 Two benchmark datasets

To assess the ability of any system to cope with heterogeneous data sources, appropriate benchmarks are needed. Since we are unaware of any that exist, we have created two that we are sharing with the research community. One approach would be to collect questions from real industrial Q&A examples, but sharing such a dataset openly would face practical and political obstacles. Instead, we opted to modify Spider, a popular Text-to-SQL benchmark (Yu et al., 2018) that consists of a collection of databases and tables plus several thousand pairs of natural language queries and their associated ground-truth SQL translations. Our benchmarks augment Spider in two distinct ways.

4.1 Benchmark I

Benchmark I leaves all natural language questions as is, but replaces a fraction of the Spider database

tables with equivalent API calls. Since many natural language queries in Spider require combining information from multiple tables, replacing some tables with API calls necessitates combining database and API calls to answer a NL question. We use a subset of the full Spider benchmark containing 948 questions and associated ground truth SQLs.

For example, the Spider dataset includes the database *museum_visit*, which contains 3 tables: *museum*, *visitor* and *visit*. The *museum* table has the following SQL definition:

```
CREATE TABLE museum (
    Museum_ID int PRIMARY KEY,
    Name text,
    Num_of_Staff int,
    Open_Year text
);
```

To generate the API equivalent of *museum*, we programmatically convert its SQL definition to an API */museum* that is written in Python using the Flask framework. When executed, the API collects all of the table records from the database into a dataframe. Then, it applies filtering, selection and aggregation operations (written in Python) to that dataframe to produce the same rows and columns that the SQL execution would have produced. Finally, */museum* is wrappered as a a user-defined function (UDF).

We also programmatically convert the SQL definition of *museum* to a Swagger definition for */museum*:

```
/museum:
    post:
      description: The API 'museum'
          handles requests
      regarding 'museum_id, name,
          num_of_staff,
        open_year', in the context of %'
            museum_visit'.
      requestBody:
        required: false
        content:
         application/json:
            schema:
              type: object
              properties:
                museum_id:
                  type: integer
                name:
                  type: string
                num_of_staff:
                  type: integer
                open_year:
                  type: string
      responses:
        '200':
          description: Data returned.
```

Suppose a user asks *"What are the opening year and staff number of the museum named Plaza Museum?"*. In the original Spider, this is converted to the SQL statement *"SELECT Num_of_Staff , Open_Year FROM museum WHERE name = 'Plaza Museum'"*, which is then executed on the database. However, if the database table *museum* is replaced by the API */museum*, the system must call the */museum* API with the parameter *name* ="Plaza Museum". Now suppose that the user asks: *"What are the id, name and membership level of visitors who have spent the most money in total in all museum tickets?"*. In the original Spider, this would entail joining the *museum* and *visit* tables. In the extended benchmark, the system must combine results of a database call to the *visit* table with results of an API call to */museum*.

4.2 Benchmark II

Benchmark II augments Spider by introducing a set of 16 scalar APIs that perform lexical, numeric, or geospatial operations that are generic enough to ensure that they integrate naturally with most of the existing Spider questions. Examples include counting the number of syllables in a string; determining whether an integer is a prime, a square, or a Fibonacci number; determining the latitude, longitude, country, or province of a place; or calculating the distance between two places. The numeric APIs accept float or integer inputs; floats are truncated to integers. All geospatial APIs are wrappers around Google Geospatial APIs.[4]

Given a question/SQL pair from the original Spider benchmark, we prompted an LLM with the original Spider question/SQL pair, a set of schemas for the Spider and scalar APIs, and a request to appropriately blend 1-3 APIs into the original Spider question. We applied this transformation 3-5 times to each question in 26 selected Spider databases, resulting in 5456 candidate augmented questions.

Unlike the original Spider benchmark, Benchmark II does not provide a ground-truth SQL expression for each question, as the APIs that we have introduced have no SQL equivalent. Instead, the ground truth consists of rows that represent the correct answer to the question. To generate these ground-truth rows, we used three different LLM-based techniques. One technique treated the API calls as virtual tables that were joined with the original Spider tables, while the other two executed the APIs directly as User-Defined-Functions (UDFs).

[4] https://developers.google.com/ar/develop/geospatial

Out of the 5456 candidate augmented questions, 1649 produced no rows for any of the techniques. In many of these cases, the ground-truth SQL was legitimate, but no rows were produced due to the extra restrictions imposed by the scalar APIs. Since there are many erroneous ways to generate no rows, such questions would undermine the utility of the benchmark. Therefore, we dropped them from further consideration.[5] Each of the remaining 3807 question/answer pairs was subjected to a human vetting process that consumed over 100 person-hours, resulting in the final set of 2338 question/answer pairs that comprise Benchmark II.

This benchmark creation procedure yielded questions that, while decidedly nerdy in nature, serve our purpose of requiring a question-answering system to perform a mix of database and API calls.[6] For example, the original Spider utterance

> "What are the names and birth dates of people, ordered by their names in alphabetical order?"

becomes QuestionID *poker_player.165*:

> "What are the names and birth dates of people who live in countries where the name has a prime number of syllables, ordered by their names in alphabetical order?"

Answering this question correctly requires calling the *get_country_of_place* API on the nationality field of the people table in the *poker_player* database, and then applying the *count_syllables* and *is_prime* APIs successively to that result.

Further details about the process by which Benchmark II was created are provided in Appendix D, and more examples of transformed questions are provided in Appendix E.

5 Experiments

We conducted two sets of experiments: one using Benchmark I and the other using Benchmark II. In each case, we compared the accuracy of the three methods introduced in Section 3 and conducted further investigations to gain qualitative insights into what characteristics of the methods most contributed to their overall efficacy.

In both experiments, accuracy was based on retrieved data. For Benchmark I, the benchmark rows were generated by executing the original gold standard SQL on the original Spider database. For Benchmark II, the benchmark rows were obtained by the human-vetting process overviewed in Section 4 and detailed in Appendix D. Returned rows were matched against the benchmark rows using the comparison approach introduced by (Zhong et al., 2020), which requires that the correct rows be retrieved but forgives extra columns. The reported accuracy was the ratio of questions for which the comparison was deemed a match.

5.1 Benchmark I experiments

Figure 2 shows the accuracy of the methods introduced in Section 3 as a function of the percentage of database tables that have been replaced by APIs.

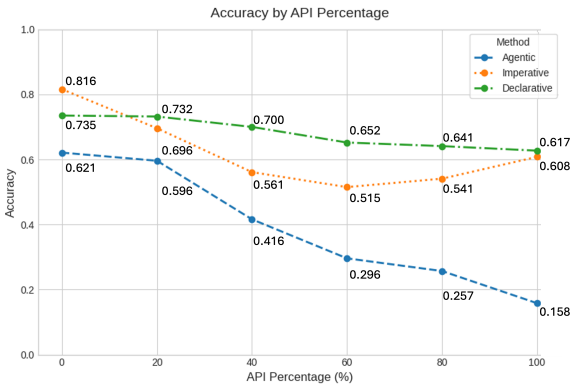

Figure 2: Measured accuracies for **Agentic** (xLAM-based), **Imperative** and **Declarative** methods vs the percentage of DB tables that were replaced by APIs.

Without any APIs (at 0%), the Imperative approach outperforms the other approaches, achieving an accuracy of 0.816, which is comparable to state-of-the-art open-source LLMs on zero-shot evaluation.[7] As the proportion of API calls increases, the accuracies of the Agentic and Declarative systems decrease monotonically, albeit much more steeply for Agentic than for Declarative. In contrast, the accuracy of the Imperative approach is *not* monotonic; it struggles most when the mixture of API and DB calls is roughly even. Error analysis reveals that, under these conditions, the complexity of the python code generated by Imperative is the

[5] An alternative would have been to augment the Spider tables to include rows that match the extra criteria, but we felt that deviating from the well-established Spider content would make our dataset less useful to the research community.

[6] Moreover, piggybacking on the existing Spider benchmark enables us to calibrate our methods against it and generate a dataset an order of magnitude larger than would have been feasible otherwise.

[7] See https://yale-lily.github.io/spider.

greatest, and its accuracy is impaired by programming errors that arise when stitching together API calls and DB queries (typically inconsistent variable use). For API percentages between 20% and 80%, the Declarative approach outshines the other two substantially. When all of the DB queries are replaced with API calls (100%), Declarative is still the best, but only marginally better than Imperative.

Careful error analysis reveals that three factors account for most of the Agentic method's poor performance for mixtures of DB and API calls:

1. **Sequencing**. State-of-the-art tool-calling LLM based systems are not trained, fine-tuned or optimized to perform complex API sequencing, merging, and aggregation tasks. They perform relatively well on questions whose answers require a single API call, but at higher API percentages even the easier Spider queries (e.g., "What is the total number of singers?") typically require invoking multiple APIs and sequencing them properly (e.g. invoking getAllSingers followed by getSize).

2. **Routing**. Even when the system generates a proper sequence, the master LLM sometimes fails to properly route the decomposed questions to the appropriate database or API tool.

3. **Hallucinated or improperly bound inputs**. Even when the right API is selected (e.g. *is_prime*), the arguments for its invocation are often hallucinated or otherwise incorrect. For example, we have observed cases where the system correctly finds dozens of items with e.g. "count > 3" but then loses track of some of them, resulting in incorrect aggregation.

The Declarative approach avoids all these issues by providing to an LLM a single relational view that removes all the complexity of dealing with multiple heterogeneous sources. To the LLM, everything appears to be relational, and thus it can leverage its Text-to-SQL capability to generate a SQL query for each NL question. The Query Rewriter is then responsible for injecting API invocations through UDFs with the proper arguments (inferred from a deterministic analysis of the SQL query).

5.2 Benchmark II experiments

Table 1 summarizes the accuracy and the average execution time per question for the **Agentic** (xLAM), **Imperative**, **Declarative** and **Declarative2** approaches for the 2338 questions that constitute Benchmark II.

Method	Accuracy	Exec Time (sec)
Agentic	0.357	37.81
Imperative	0.614	15.56
Declarative	0.639	16.50
Declarative2	**0.689**	16.24

Table 1: Accuracy and average execution time per question for the four query strategies on Benchmark II.

Both declarative approaches are somewhat more accurate than **Imperative** and vastly more accurate than **Agentic**. Detailed comparisons of the 204 questions that were answered correctly by **Declarative2** but not by **Declarative** indicate that **Declarative** can suffer from errors that creep in during either the query rewriting or the extra virtual table joins. We believe that improvements to the query rewriter would close this gap somewhat. While **Declarative2** enjoys an accuracy advantage over **Declarative**, its use is limited to scalar APIs, whereas the **Declarative** method described in Section 3 applies broadly to APIs that generate outputs representable as scalars, vectors or tables. The total execution time (including the time to process the NL into SQL or a program and execute the resulting database and API calls) for **Agentic** was notably slower than that of **Imperative** and the two declarative approaches. The trace shown in Appendix C.4 suggests that loops over LLM invocations are one large source of inefficiency for **Agentic**.

6 Conclusion

The ability to answer questions in industrial systems typified by data source heterogeneity is a critical and hitherto unmet need. To help address that gap, we introduced *siwarex*, a declarative system that uses SQL to represent user intent in conjunction with virtual tables and UDFs, thereby enabling external APIs to be treated alongside database tables in a unified framework. We also introduced two new benchmarks that assess a system's ability to cope with data heterogeneity and used them to establish the superiority of the declarative approach over imperative and agentic approaches. We have released these benchmarks to spur further research in this area.[8]

[8]Please visit https://huggingface.co/datasets/ibm-research/SQL-API-Bench.

Limitations

One limitation that we are eager to address in future work is that our benchmark's evaluation metric only considers the execution *accuracy* of the final results. Especially since we wish to produce a benchmark that meaningfully captures practical issues that arise in industrial settings, it is incumbent on us to augment this metric with an execution *performance* metric (i.e. efficiency or speed). While our augmentations of the Spider database had several advantages, in general the table sizes are too small (just dozens of rows) to support credible measurements of the system execution speed, both in terms of the time required to formulate the declarative statement and the time required to execute it. Augmenting a benchmark with larger tables, such as BIRD (Li et al., 2023a), would be far preferable for such a purpose.

Benchmark II only includes APIs that produce scalar outputs (e.g. Boolean outputs for APIs like */is_prime*, or numeric outputs for APIs like */count_syllables*). In many industrial use cases, APIs produce vector or table outputs. For example, the correlation between two sensors can be expressed as a numeric scalar, but correlations between one sensor and several others would naturally be expressed as a vector, and correlations among all sensor pairs would most naturally be represented as a table. As mentioned in Section 3.1, the Declarative method is in principle capable of handling APIs that generate vector or table outputs. In future work, we hope to update Benchmark II to include such APIs and then measure the relative effectiveness of the Declarative, Agentic, Imperative approaches.

Finally, another important future extension is to deploy and test our declarative framework in an industrial system that contains heterogeneous data sources. This being the entire inspiration and motivation for our work, we are confident that our framework can be applied in systems that contain a variety of SQL and noSQL databases as well as APIs that access and/or analyze time series data. However, many practical questions remain to be answered, including the response time as experienced by an end user (which combines the formulation and execution times as mentioned above) and the degree to which creating API mappings for existing APIs can be automated to ensure that the system can be deployed quickly in new environments with minimal configuration.

References

Alexis Tacnet Alok Kothari Antoine Roux Arthur Mensch Audrey Herblin-Stoop Augustin Garreau Austin Birky Bam4d Baptiste Bout Baudouin de Monicault Blanche Savary Carole Rambaud Caroline Feldman Devendra Singh Chaplot Diego de las Casas Diogo Costa Eleonore Arcelin Emma Bou Hanna Etienne Metzger Gaspard Blanchet Gianna Lengyel Guillaume Bour Guillaume Lample Harizo Rajaona Henri Roussez Hichem Sattouf Ian Mack Jean-Malo Delignon Jessica Chudnovsky Justus Murke Kartik Khandelwal Lawrence Stewart Louis Martin Louis Ternon Lucile Saulnier Lélio Renard Lavaud Margaret Jennings Marie Pellat Marie Torelli Marie-Anne Lachaux Marjorie Janiewicz Mickaël Seznec Nicolas Schuhl Niklas Muhs Olivier de Garrigues Patrick von Platen Paul Jacob Pauline Buche Pavan Kumar Reddy Perry Savas Pierre Stock Romain Sauvestre Sagar Vaze Sandeep Subramanian Saurabh Garg Sophia Yang Szymon Antoniak Teven Le Scao Thibault Schueller Thibaut Lavril Thomas Wang Théophile Gervet Timothée Lacroix Valera Nemychnikova Wendy Shang William El Sayed William Marshall Albert Jiang, Alexandre Sablayrolles. Jul 24, 2024. Mistral-large. `https://huggingface.co/mistralai/Mistral-Large-Instruct-2407`. Accessed: [Jul 04, 2025].

Kinjal Basu, Ibrahim Abdelaziz, Kiran Kate, Mayank Agarwal, Maxwell Crouse, Yara Rizk, Kelsey Bradford, Asim Munawar, Sadhana Kumaravel, Saurabh Goyal, Xin Wang, Luis A. Lastras, and Pavan Kapanipathi. 2025. NESTFUL: A benchmark for evaluating LLMs on nested sequences of API calls. *arXiv preprint arXiv:2409.03797*.

Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. 2025. ReFoRCE: A text-to-SQL agent with self-refinement, consensus enforcement, and column exploration. *arXiv preprint arXiv:2502.00675*.

Benjamin Elder, Anupama Murthi, Jungkoo Kang, Ankita Rajaram Naik, Kiran Kate, Kinjal Basu, and Danish Contractor. 2025. Invocable APIs derived from NL2SQL datasets for LLM tool-calling evaluation. *arXiv preprint arXiv:2506.11266*.

Dayuan Fu, Jianzhao Huang, Siyuan Lu, Guanting Dong, Yejie Wang, Keqing He, and Weiran Xu. 2024. PreAct: Prediction enhances agent's planning ability. In *International Conference on Computational Linguistics*.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-SQL empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363*.

Parker Glenn, Parag Dakle, Liang Wang, and Preethi Raghavan. 2024. BlendSQL: A scalable dialect for unifying hybrid question answering in relational algebra. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 453–466,

Bangkok, Thailand. Association for Computational Linguistics.

Saurabh Jha, Rohan Arora, Yuji Watanabe, Takumi Yanagawa, Yinfang Chen, Jackson Clark, Bhavya Bhavya, Mudit Verma, Harshit Kumar, Hirokuni Kitahara, Noah Zheutlin, Saki Takano, Divya Pathak, Felix George, Xinbo Wu, Bekir O. Turkkan, Gerard Vanloo, Michael Nidd, Ting Dai, Oishik Chatterjee, Pranjal Gupta, Suranjana Samanta, Pooja Aggarwal, Rong Lee, Pavankumar Murali, Jae wook Ahn, Debanjana Kar, Ameet Rahane, Carlos Fonseca, Amit Paradkar, Yu Deng, Pratibha Moogi, Prateeti Mohapatra, Naoki Abe, Chandrasekhar Narayanaswami, Tianyin Xu, Lav R. Varshney, Ruchi Mahindru, Anca Sailer, Laura Shwartz, Daby Sow, Nicholas C. M. Fuller, and Ruchir Puri. 2025. ITBench: Evaluating AI agents across diverse real-world IT automation tasks. *arXiv preprint arXiv:2502.05352*.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. 2024. An LLM compiler for parallel function calling. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Takeshi Kojima, Shinn Yao, Jinyi Zhao, Xiang Ren, et al. 2023. ToolLLM: Facilitating LLMs to master 16000+ real-world APIs. *arXiv preprint arXiv:2312.11568*.

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. 2025. Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows. *arXiv preprint arXiv:2411.07763*.

Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, et al. 2023a. Can LLM already serve as a database interface. *A big bench for large-scale database grounded text-to-SQLs. CoRR abs/2305.03111*.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024. Can LLM already serve as a database interface? a big bench for large-scale database grounded text-to-SQLs. *Advances in Neural Information Processing Systems*, 36.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023b. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *The 2023 Conference on Empirical Methods in Natural Language Processing*.

Qiqiang Lin, Muning Wen, Qiuying Peng, Guanyu Nie, Junwei Liao, Jun Wang, Xiaoyun Mo, Jiamu Zhou, Cheng Cheng, Yin Zhao, Jun Wang, and Weinan Zhang. 2024. Hammer: Robust function-calling for on-device language models via function masking. *arXiv preprint arXiv:2410.04587*.

Shishir G. Patil, Huanzhi Mao, Charlie Cheng-Jie Ji, Fanjia Yan, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. 2024. The Berkeley Function Calling Leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Advances in Neural Information Processing Systems*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, et al. 2025. APIGen-MT: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. *arXiv preprint arXiv:2504.03601*.

Timo Schick, Kamil Dwivedi-Yu, Nathanael Schärli, Nathan Scales, Le Hou, Daniel Khashabi, Patrick Lewis, Oana-Maria Simig, and Sebastian Riedel. 2023. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Shinn Yao, Jinyi Zhao, Dian Yu, Kangyan Chen, Bill Yuchen Lin, Xiang Ma, Yujie Bang, Qingyun Zhou, and Xiang Ren. 2022. ReAct: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

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

Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Akshara Prabhakar, Haolin Chen, Zhiwei Liu, Yihao Feng, Tulika Awalgaonkar, Rithesh Murthy, Eric Hu, Zeyuan Chen, Ran Xu, Juan Carlos Niebles, Shelby Heinecke, Huan Wang, Silvio Savarese, and Caiming Xiong. 2024. xLAM: A family of large action models to empower AI agent systems.

Ruiqi Zhong, Tao Yu, and Dan Klein. 2020. Semantic evaluation for text-to-SQL with distilled test suites. *CoRR*, abs/2010.02840.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*.

Appendix A Hybrid questions

Table 2 of this appendix provides a few more examples of questions inspired by industry scenarios with which we are familiar, the answers to which require combining database and API calls.

Appendix B UDF and API wrapper example

This appendix details the UDF definition and API wrapper for one example API among the Benchmark II set: $/is\_fibonacci$.

Listing 1 shows the UDF definition for $/is\_fibonacci$ that is added to the database.

```
CREATE FUNCTION is_fibonacci_udf(
    dnf_constraint dnf)
  RETURNS SETOF is_fibonacci
AS $$
  # Note dnf_constraint might be None to
       indicate that the arguments are
      not constrained
  # return tuple containing lists as
      composite types
  plpy.info(f"Input type: {type(
      dnf_constraint)}")
  plpy.info(f"Input: {dnf_constraint}")

  import requests
  if dnf_constraint is None:
    dnf_json = {}
  else:
        dnf_json = dnf_constraint
  results = requests.post("<server_url
      >:5001/
      APIWrapperFor_is_fibonacci_udf",
      json=dnf_json)
  results.close()
  return results.json()
$$ LANGUAGE plpython3u;
```

Listing 1: UDF definition for */is_fibonacci*.

Listing 2 displays the API wrapper code called by the UDF, which calls the actual */is_fibonacci* code and manipulates the result into a form suitable for the database to consume it.

```
class APIWrapperFor_is_fibonacci_udf(
    Resource):

    def post(self):
        data = request.get_json()
        results = []
        failed_functions = []

        url = "http://<server_name>/
            is_fibonacci"
        method = RESTAPIFunction.
            RESTAPI_METHOD.POST
        parameters = [OpenAPIParameter("
            number", False, "float",
            OpenAPIParameter.IN_ENUM.
            QUERY), OpenAPIParameter("
            truth", False, "boolean",
            OpenAPIParameter.IN_ENUM.
            QUERY)]
        output = ["number", "truth"]
        f = RESTAPIFunction(url, method,
             parameters,output_keys =
            output)
        if can_invoke_api(data, f)[0]:
            result = invoke(data, f)
            results.append((output,
                result))
        else:
            failed_functions.append(f)

        if len(results) == 0:
            raise Exception(f"Cannot␣
                invoke␣any␣of␣the␣REST␣
                API␣{[f.url␣for␣f␣in␣
                failed_functions]}")
        else:
            if len(failed_functions) >
                0:
                logger.warning(f"It␣
                    failed␣in␣all␣of␣the
                    ␣urls␣in␣{[f.url␣for
                    ␣f␣in␣
                    failed_functions]}")

        return return_response(merge(
            results))

api.add_resource(
    APIWrapperFor__is_fibonacci_udf, '/
    APIWrapperFor__is_fibonacci_udf')
```

Listing 2: API wrapper for */is_fibonacci*.

Appendix C Query recipes

This appendix illustrates the differences among the various query strategies by comparing the ground-truth recipes they should produce for a given question drawn from the Benchmark II dataset. We use *recipe* as a generic term for an expression that can be evaluated on a database and a set of APIs to produce an answer in the form of rows and columns. For the Declarative and Declarative2 query strategies, the recipe is a SQL statement. For the Imperative query strategy, the recipe is a Python program. There is no recipe for the Agentic query strategy, as it completely interleaves the processes of retrieval and reasoning, so in that case we show a typical chain-of-thought trace.

In what follows, we will show the recipe that each strategy should ideally produce (i.e. the ground truth) for QuestionID poker_player.172 from the Benchmark II dataset:

> Show names of people whose nationality is not 'Russia' and whose name contains a number of syllables that is a Fibonacci number.

Question	Required DB and API calls
What is the condition of all submersible pumps in my organization?	• DB call to retrieve assets with type = 'submersible pump' and owner = 'me' • API call to condition insight analyzer that either applies textual analysis to recent work order descriptions or analyzes appropriate time series.
Which chillers at the Ft. Worth site are in bad condition?	• DB call to retrieve assets with type = 'chiller' and location = 'Ft. Worth'. • API call to a condition insight analyzer. • API call to condition insight evaluator that determines whether the insight analysis for a given asset qualifies as "bad".
Find near duplicates of open work orders for assets that are at least 10 years old	• DB call to retrieve assets with status = 'open' and TODAY - install_date >= 10 years. • API call to work order similarity scorer.
For workorders pertaining to Elsco transformers in the ERCOT grid, list ones with missing problem codes and try to classify them automatically based on the work order description.	• DB call to retrieve assets with type = 'transformer', manufacturer = 'Elsco', and powergrid = 'ERCOT'. • API call to problem code classifier.
At the Shreveport refinery, identify butterfly valves associated with tanks whose pressure has come within 1kpa of the nominal limit at least twice during the past month, and group them by manufacturer.	• DB call to retrieve assets with location = 'Shreveport', type = 'butterfly valve' OR 'tank'. • API call to time series analytic to find tanks in the above list that satisfy the pressure criterion. • API call to physical asset map to identify adjacent butterfly valves. • DB call to group the valves by manufacturer.

Table 2: Examples of industry-inspired questions whose answers require a combination of database and API calls.

C.1 Declarative

For poker_player.172, the Declarative query strategy should produce a pure SQL statement similar to:

```
SELECT p.Name FROM people as p
JOIN count_syllables AS cs
ON p.Name = cs.string
JOIN is_fibonacci as ifb
ON cs.count = ifb.number
WHERE p.Nationality !='Russia' AND
ifb.truth = true
```

The query rewriter then translates this expression by replacing *is_fibonacci* and *count_syllables* with UDFs with the correct arguments, and the result is executed to produce the output rows.

C.2 Declarative2

The Declarative2 query strategy should produce a SQL statement like:

```
SELECT p.Name FROM people as p
WHERE p.Nationality !="Russia" AND
is_fibonacci(count_syllables(p.Name)) = true
```

This expression is then executed directly on a SQLite database that includes the UDFs *is_fibonacci* and *count_syllables* as application-defined functions.

C.3 Imperative

The Imperative query strategy should produce a Python program equivalent to the following:

```
from siwarex_lite.api_functions import *
import sqlite3
import json

# Connect to the SQLite database
conn = sqlite3.connect('/Users/ffhiii/
    projects/generate-api-benchmark/
    schemas/poker_player/poker_player.
    sqlite')
cursor = conn.cursor()

# Query to get the necessary data
cursor.execute("SELECT Name, Nationality
     FROM people")

# Fetch all rows from the query
rows = cursor.fetchall()

# Initialize the final projection
final_projection = []

# Iterate through each row
for row in rows:

  name = row[0]
  nationality = row[1]

  # Check if the nationality is not '
      Russia'

  if nationality != 'Russia':

    # Count the number of syllables in
        the name
    syllable_count = count_syllables(
        string=name)

    # Check if the number of syllables
        is a Fibonacci number
    is_fibonacci_number = is_fibonacci(
        number=syllable_count)

    # If the number of syllables is a
        Fibonacci number, add to the
        final projection
    if is_fibonacci_number:
      final_projection.append({"Name":
          name, "Nationality":
          nationality})

# Close the database connection
conn.close()

# Print the final projection
print(json.dumps(final_projection,
    indent=2))
```

Listing 3: Auto-generated python program produced for QuestionID poker_player.172 by the imperative strategy.

This program is then executed to produce the output rows.

C.4 Agentic

Here is a somewhat trimmed version of the chain-of-thought trace from an execution of the Agentic query strategy on poker_player.172:

```
Entering new AgentExecutor chain...
 To solve this problem, we need to first
      retrieve the names and
 nationalities from the database using
     Text2SQL. Then, we will
 check if the nationality is not Russia
     and if the number of
 syllables in their name is a Fibonacci
     number using XLAM
 and execute_api.

Action: Text2SQL
Action Input: {"question": "SELECT name,
     nationality FROM people LIMIT 100"}
Thought: We need to execute the API to
    get the number of syllables in the
    name.
Action: execute_api
Action Input: {"api_name":"
    count_syllables","arguments":{"
    string": "Todor Salparaov"}}
Observation: {'result rows:': '[["
    Aleksey Ostapenko", "Russia"], ["
    Todor Salparov", "Bulgaria"], ["
    Roman Bragin", "Russia"], ["Sergey
    Grankin", "Russia"], ["Yevgeni
    Sivozhelez", "Russia"], ["Maksim
    Botin", "Russia"], ["Semen
    Poltavskiy", "Russia"]]'}
```

```
Thought: Now we need to filter out those
     who are not Russian and then check
    if the number of syllables in their
    name is a Fibonacci number.

Action: XLAM
Action Input: {"question": "Is the
    number of syllables in Todor
    Salparov a Fibonacci number?"}
Thought: We need to execute the API to
    get the number of syllables in the
    name.
Action: execute_api
Action Input: {"api_name":"
    count_syllables","arguments":{"
    string": "Todor Salparov"}}
Observation: {'string': 'Todor Salparov
    ', 'count': 5}
Thought: The number of syllables in
    Todor Salparov is 5 which is a
    Fibonacci number, and he is not
    Russian so he meets the criteria.
    Let's continue checking other names.

Action: XLAM
Action Input: {"question": "Is the
    number of syllables in Roman Bragin
    a Fibonacci number?"}
Thought: We need to execute the API to
    get the number of syllables in the
    name.
Action: execute_api
Action Input: {"api_name":"
    count_syllables","arguments":{"
    string": "Sergey Grankin"}}
Observation: {'string': 'Roman Bragin',
    'count': 4}
Thought: The number of syllables in
    Roman Bragin is 4, which is not a
    Fibonacci number. Let's continue
    checking other names.

Action: XLAM
Action Input: {"question": "Is the
    number of syllables in Sergey
    Grankin a Fibonacci number?"}
Thought: We need to execute the API to
    get the number of syllables in the
    name.
Action: execute_api
Action Input: {"api_name":"
    count_syllables","arguments":{"
    string": "Sergey Grankin"}}
Observation: {'string': 'Sergey Grankin
    ', 'count': 4}
Thought: The number of syllables in
    Sergey Grankin is 4 which is not a
    Fibonacci number. Let's continue
    checking other names.

Action: XLAM
Action Input: {"question": "Is the
    number of syllables in Yevgeni
    Sivozhelez a Fibonacci number?"}
Thought: We need to execute the API to
    get the number of syllables in the
    name.
Action: execute_api
Action Input: {"api_name":"
    count_syllables","arguments":{"
    string": "Yevgeni Sivozhelez"}}
Observation: {'string': 'Yevgeni
    Sivozhelez', 'count': 7}
Thought: The number of syllables in
    Yevgeni Sivozhelez is 7 which is not
     a Fibonacci number. Let's continue
    checking other names.

Action: XLAM
Action Input: {"question": "Is the
    number of syllables in Maksim Botin
    a Fibonacci number?"}
Thought: We need to execute the API to
    get the number of syllables in the
    name.
Action: execute_api
Action Input: {"api_name":"
    count_syllables","arguments":{"
    string": "Maksim Botin"}}
Observation: {'string': 'Maksim Botin',
    'count': 4}
Thought: The number of syllables in
    Maksim Botin is 4 which is not a
    Fibonacci number. Let's continue
    checking other names.

Action: XLAM
Action Input: {"question": "Is the
    number of syllables in Semen
    Poltavskiy a Fibonacci number?"}
Thought: We need to execute the API to
    get the number of syllables in the
    name.
Action: execute_api
Action Input: {"api_name":"
    count_syllables","arguments":{"
    string": "Semen Poltavskiy"}}
Observation: {'string': 'Semen
    Poltavskiy', 'count': 5}
Thought: The number of syllables in
    Semen Poltavskiy is 5 which is a
    Fibonacci number. However, he is
    Russian so we skip him. There are no
     more names to check.

Thought: I now know the final answer
Final Answer: [{"names":["Todor Salparov
    "]}]

Finished chain.
  "QuestionID": "poker_player.172",
  "AugmentedQuestion": "Show names of
      people whose nationality is not '
      Russia' and whose name contains a
      number of syllables that is a
      Fibonacci number.",
  "AugmentedProducesRows": true,
  "PredictedQueryRows": [
    {
      "names": ["Todor Salparov"]
    }
  ],
  "time_spent": 32.83591556549072
```

Listing 4: Chain-of-thought trace for QuestionID poker_player.172 generated by **Agentic**.

From this chain-of-thought trace, it is clear that the correct answer was obtained, but in a rather cir-

cuitous manner. Rather than using the database to filter the list down to all non-Russians, the first call retrieved all players regardless of their nationality. Then, it proceeded to call */count_syllables* on each of the retrieved players. Apparently, it did not use the function */is_fibonacci* to determine whether the number of syllables retrieved by */count_syllables* was a Fibonacci number — the model evidently performs this computation itself. Inefficiencies such as these likely account for why Agentic takes approximately twice as long as the other methods (according to Table 1).

Appendix D Benchmark II details

This appendix provides details about the process by which the Benchmark II dataset was produced.

First, a subset consisting of 26 databases in the Spider dev dataset were chosen, namely:

- phone_1
- phone_market
- pilot_record
- poker_player
- produce_catalog
- farm
- activity_1
- allergy_1
- wrestler
- workshop_paper
- wedding
- tvshow
- train_show
- tracking_software_problems
- tracking_share_transactions
- tracking_orders
- tracking_grants_for_research
- theme_gallery
- swimming
- apartment_rentals
- architecture
- assets_maintenance
- battle_death
- body_builder
- car_1
- behavior_monitoring

For each original Spider question/SQL pair, Mistral-Large was prompted to produce 3-5 augmented questions that incorporated between 1 and 3 of the numeric, lexical and geospatial APIs introduced in Section 4. The prompt included the original question Spider question, the Spider table schemas, and the schemas for the virtual table equivalents of the APIs, along with a direction to "generate an augmented question that is based on the input question but also requires information from one or more tables in the set of auxiliary SQL Schema provided below". This yielded a set of 5456 candidate augmented questions.

Then, we used three different LLM-based techniques to generate candidate ground-truth rows for each of the 5456 candidate augmented questions. To minimize the likelihood of subtly biasing our findings to favor the Declarative, Agentic, or Imperative NL query approaches, we deliberately chose the ground-truth generators to be as diverse as possible in terms of methodology. The diversity was further improved by executing some of the ground-truth generators on a postgres database and others on a SQLite database.

The first technique, which we call **QR**, prompted a Mistral-Large LLM to function as a Text-to-SQL system with some extra inputs. In addition to the augmented question and the standard Spider schema, the prompt included the original Spider question, its corresponding ground-truth SQL, and additional schema to represent each API as a virtual table. For example, the schema for the $is\_fibonacci$ virtual table was:

````
## is_fibonacci
Description: Determines whether an input
number is a Fibonacci number. If the input
number is not an integer, it will be
truncated to an integer first before
it is evaluated.
```sql
CREATE TABLE is_fibonacci (
````
````

```
    number INTEGER,
    truth Boolean
);
```

We included the original Spider question/SQL pair because we found experimentally that doing so improved the quality of the augmented SQL generated by the LLM. This output was a pure SQL statement that treated any API calls as virtual tables. Examples of this SQL statement are shown in the column labeled "QR Query" in Figure 3. Then, the query-rewriter described in Section 3.1 was applied to this SQL, in effect replacing all virtual table references with their UDF equivalents. Finally, the candidate ground-truth rows were produced by executing the transformed SQL on a postgres database loaded with the Spider data, the Spider database schema, and the APIs' UDF definitions.

A second technique, which we called **SL**, used an approach similar to the Declarative2 approach described in Section 5.2. The APIs were expressed as SQLite application-defined functions[9], which are a type of UDF that can be executed directly by a SQLite database system rather than being treated as virtual tables. In essence, they act like built-in functions such as LENGTH(), LOWER(), or SUBSTRING(). Mistral-Large was prompted to generate a SQL statement for each augmented question. The prompt included the augmented question plus direct descriptions of the API functions, inputs and outputs (as opposed to the virtual table descriptions used for **QR**). The LLM output an SQL expression in which the APIs were represented as composable functions, examples of which appear in the "SL Query" column of Figure 3. This SQL expression was executed on a SQLite database into which the Spider tables and schema had been loaded (along with the application-defined functions) to produce the **SL** candidate ground-truth rows.

It is instructive to compare the **QR** and **SL** queries for QuestionID poker_player.172, for which the augmented question is:

> Show names of people whose nationality is not 'Russia' and whose name contains a number of syllables that is a Fibonacci number.

The two queries are semantically equivalent and produce the same set of rows, but the **SL** version is much easier to understand. Whereas **QR** contains two joins among the *people*, *count_syllables* and *is_fibonacci* tables, the **SL** query contains the simple functional composition *is_fibonacci(count_syllables(people.Name))*.

A third technique, which we call **IMP**, uses an approach similar in many respects to the Imperative approach of Section 3.2. Rather than generating an SQL expression for a given input question, it generates and then executes (on a SQLite database) a Python program that mixes database and API calls. The Python program is produced by prompting Mistral-Large with the original and augmented Spider questions plus the original Spider schema and the API definitions. The ground-truth program produced for QuestionID poker_player.172 is shown in Appendix C.3.

We applied **QR** and **SL** to each of the 5456 candidate augmented questions to generate **QR**, **SL**, and **IMP** candidates for each. As described in the main body of the paper, 1649 questions failed to produce ground-truth rows for any of the three techniques. Often, the produced expression appeared legitimate, and only failed to produce rows because of the extra restrictions imposed by the scalar APIs. Questions for which two or more generators produced results were analyzed automatically using the comparison technique mentioned in Section 5 to determine whether the results were compatible (i.e. equivalent in the loose sense that the rows were required to be the same but the projections (selected columns) were not).

The remaining 3807 questions fell into several categories.

1. 1783 questions had one or more rows produced by both **QR** and **SL**. Of these,
   (a) 1311 were deemed compatible (Case 1)
   (b) 472 were deemed incompatible (Case 2)
2. 150 had rows produced by **QR** but not **SL** (Case 3)
3. 1760 had rows produced by **SL** but not **QR** (Case 4). These were further subdivided into
   (a) 480 had rows produced by **SL** but not **IMP** (Case 4a)
   (b) 866 had rows produced by both **SL** and **IMP** and were deemed compatible (Case 4b)
   (c) 434 had rows produced by both **SL** and **IMP** but were deemed incompatible (Case 4c)

[9] See https://www.sqlite.org/appfunc.html

(d) 229 had rows produced by **IMP** but not **SL** (Case 4d)

An evaluation UI was designed to help human evaluators with each of these cases. Figure 3 is a screenshot of the UI used for Case 1, which is the situation in which **QR** and **SL** produce one or more rows and are deemed compatible. Typically, the evaluator used the evaluation UI in conjunction with the pgadmin postgres UI, allowing them to experiment with the queries and the data if they wished. The user indicated whether they wished to select the **QR** ground-truth candidate, the **SL** candidate, or neither by clicking on the appropriate button. Even when two candidates were deemed compatible, they often had different numbers of columns or different column labels. In such cases, evaluators were asked to prefer the minimal set of columns that would be required to answer the question fully.

In more complex cases in which the candidate ground-truth rows were deemed incompatible, such as Case 2, the UI included an additional hint column, as illustrated in Figure 4. The hint was produced by an LLM that was prompted to analyze the question and the pseudo-SQL expression and produce an assessment of whether the expression was correct and if so why or why not. While the recommendations were not sufficiently reliable to be trusted by the human evaluator, the insights were generally quite helpful.

We identified several categories of reasons why we found it necessary to eliminate certain questions, which are not necessarily mutually exclusive:

1. The question was inherently nonsensical, e.g. apartment_rentals.187: "...whose building's manager's names are Fibonacci numbers when converted to integers".

2. The question was inherently ambiguous. For example, activity_1.218 was rejected because the question has multiple interpretations: "How many students are advised by each rank of faculty who participate in activities that have names with a number of syllables divisible by 3? List the rank and the number of students." In this case, there is a legitimate question whether "names with a number of syllables" applies to students, faculty, or activities.

3. No ground-truth generator provided a correct set of rows. A common variant was failure to include a DISTINCT, resulting in multiple identical rows.

4. No ground-truth generator provided an appropriate minimal but sufficient set of columns. A common issue was that all generators provided extraneous columns.

5. There was an incorrect cast (e.g. from text or float to integer).

6. The question asked for an extremum value but there were ties, so multiple answers were legitimate.

7. The geospatial API was applied to a field that was not a place (this happened particularly when countries or cities were described by integer IDs or abbreviations like BAL for Baltimore). In rare cases, the system attempted to extract a location from an event or a date.

8. Failure to handle dates appropriately. There were several variants; in one case "occurring after" a specific date was interpreted as occurring after midnight on that date, so that events occurring on the day itself were counted as occurring after the date.

9. Failure to use the correct codes (e.g. "Male" vs "M" or "Yes" vs 1 vs true)

10. Mistaken interpretation of a column e.g. phone_market.29, where *num_of_employees* was misinterpreted as being equivalent to population density. Another example is farm.180, where the question asked to filter on cities with prime census ranking but the ranking was a non-numeric string like "1442 of 5,000".

11. Semantic confusion, e.g. using a geospatial API to extract a province or state from a country.

The end result of this human vetting process was a benchmark dataset consisting of 2338 pairs of questions and accompanying ground-truth rows, which we are sharing with the research community (see footnotes in the introduction and conclusion).

Appendix E Benchmark II questions

Table 3 provides a few more examples of transformed Spider questions for Benchmark II.

poker_player.197	Count the number of different nationalities where the average height of players is above 190 cm.	SELECT COUNT(DISTINCT p.Nationality) FROM people p JOIN poker_player pp ON p.People_ID = pp.People_ID GROUP BY p.Nationality HAVING AVG(p.Height) > 190;	[{ "_col_0": 1 }]	SELECT COUNT(DISTINCT people.Nationality) FROM people WHERE people.Height > 190;	[{ "COUNT(DISTINCT people.Nationality)": 1 }]	Choose QR Choose SL Reject Both Defer
poker_player.172	Show names of people whose nationality is not 'Russia' and whose name contains a number of syllables that is a Fibonacci number.	SELECT p.Name FROM people AS p JOIN count_syllables AS cs ON p.Name = cs.string JOIN is_fibonacci AS ifb ON cs.count = ifb.number WHERE p.Nationality != 'Russia' AND ifb.truth = TRUE	[{ "name": "Teodor Salparov" }]	SELECT people.Name FROM people WHERE people.Nationality != 'Russia' AND is_fibonacci (count_syllables (people.Name)) = 1;	[{ "Name": "Teodor Salparov" }]	Choose QR Choose SL Reject Both Defer

Figure 3: Closeup screenshot of UI used to compare ground truth candidates generated by **QR** and **SL** methods, in the case where both produce rows and those rows are deemed mutually compatible.

Benchmark Dataset

QuestionID	Question	QR Query	QR Rows	SL Query	SL Rows	Hint	Jeff's Actions
farm.182	Find the official names of cities with population bigger than 1500 or smaller than 500, and check if their Area_km_2 is a perfect square.	SELECT c.Official_Name FROM city AS c JOIN is_perfect_square AS psq ON CAST(c.Area_km_2 AS INT) = psq.number WHERE c.Population > 1500 OR c.Population < 500 AND psq.truth = TRUE;	[{ "official_name": "Perth-Andover" }]	SELECT city.Official_Name, city.Population, city.Area_km_2, is_perfect_square (city.Area_km_2) AS is_perfect_square FROM city WHERE city.Population > 1500 OR city.Population < 500;	[{ "Official_Name": "Grand Falls/Grand-Sault", "Population": 5706, "Area_km_2": 18.06, "is_perfect_square": 0 }, { "Official_Name": "Perth-Andover", "Population": 1778, "Area_km_2": 8.89, "is_perfect_square": 0 }, { "Official_Name": "Aroostook", "Population": 351, "Area_km_2": 2.24, "is_perfect_square": 0 }]	1. **Difference between the two queries:** - **Query 1** joins the `city` table with an `is_perfect_square` table on the condition that the city's area (cast to integer) matches a number in the `is_perfect_square` table where the corresponding truth value is true. It filters cities based on population before the join. - **Query 2** selects from the `city` table directly and includes a function call `is_perfect_square(city.Area_km_2)` to determine if the area is a perfect square. It filters cities based on population but does not explicitly filter based on whether the area is a perfect square in the WHERE clause. Instead, it seems to intend to display this information as a separate column. 2. **Which one is more likely to accurately represent the question:** - The question asks to find cities with specific population criteria and then check if their area is a perfect square. **Query 2** aligns better with this intention because it explicitly calculates whether each city's area is a perfect square using the `is_perfect_square()` function. Although it doesn't filter out non-perfect square areas in the WHERE clause, it provides the necessary information to do so post-query. - **Query 1**, while attempting to filter by perfect squares through a join, might not capture the exact logic intended by the question due to its structure and could potentially miss some valid results depending on how the `is_perfect_square` table is populated. Therefore, **Query 2** is more likely to accurately represent the question, assuming that further processing can be done to filter out rows where `is_perfect_square` is	Choose QR Choose SL Reject Both Defer

Figure 4: Screenshot of UI used to compare ground truth candidates.

QuestionID	Original Spider	Benchmark II
aircraft.20	What is the average number of international passengers of all airports?	What is the average number of international passengers of all airports whose names have exactly three syllables?
aircraft.21	What is the average number of international passengers of all airports?	What is the average number of international passengers of all airports located in countries where the name length is a prime number?
aircraft.27	What is the average number of international passengers for an airport?	What is the average number of international passengers for airports within a 100 km radius of London?
body_builder.4	How many body builders are there?	How many body builders are there who were born in places within 100 km of Port Huron, Michigan?
body_builder.16	What is the average snatch score of body builders?	What is the average snatch score of body builders whose total weight lifted is a prime number?
company_office.185	How many companies are in either 'Banking' industry or 'Conglomerate' industry?	How many companies in either 'Banking' industry or 'Conglomerate' industry have headquarters in countries where the name has a prime number of syllables?
farm.10	List the total number of horses on farms in ascending order.	List the total number of horses on farms in ascending order, but only include those farms whose ID is a prime number.
phone_1.1	The names of models that launched between 2002 and 2004.	Which models launched between 2002 and 2004 have a ROM size that is divisible by 32?

Table 3: Examples of transformed versions of questions from the original Spider dev set. The Spider dev set had no QuestionIDs; we introduced them for bookkeeping purposes. Note that the first three examples (for the aircraft database) derive from essentially the same original Spider question, but were transformed into very different Benchmark II questions. Most of the augmented Benchmark II questions require 1 or 2 API calls. While the LLM tried to generate questions requiring 3 API calls in some cases, most of them failed because either the third constraint was logically nonsensical or it was so restrictive that it resulted in no rows being generated.