A&A, 711, A302 (2026)
https://doi.org/10.1051/0004-6361/202558221

Astronomy & Astrophysics

Querying an astronomical database using large language models: the ALeRCE text-to-SQL system

P. A. Estévez[1,2,⋆], J. Espejo-Moreira[1,2], S. Sanfeliú-Alvarez[1,5], F. Förster[2,3,4,5], A. M. Muñoz Arancibia[2,4], G. Cabrera-Vives[6,2,7,8], F. E. Bauer[9], A. Bayo[10], M. Catelan[2,11,12], R. Dastidar[2], L. Hernández-García[13,14,2], J. A. Intriago[1], and G. Pignata[9]

[1] Department of Electrical Engineering, University of Chile, Av. Tupper 2007, Santiago, Chile
[2] Millennium Institute of Astrophysics (MAS), Nuncio Monseñor Sótero Sanz 100, Providencia, Santiago, Chile
[3] Data and Artificial Intelligence Initiative (ID&IA), Universidad de Chile, Chile
[4] Center for Mathematical Modeling, Universidad de Chile, Beauchef 851, North building, 7th floor, Santiago 8320000, Chile
[5] Departamento de Astronomía, Universidad de Chile, Casilla 36D, Santiago, Chile
[6] Department of Computer Science, Universidad de Concepción, Edmundo Larenas 219, Concepción, Chile
[7] Center for Data and Artificial Intelligence, Universidad de Concepción, Edmundo Larenas 310, Concepción, Chile
[8] Heidelberg Institute for Theoretical Studies, Heidelberg, Baden-Württemberg, Germany
[9] Instituto de Alta Investigación, Universidad de Tarapacá, Casilla 7D, Arica 1010000, Chile
[10] European Southern Observatory, Karl-Schwarzschild-Strasse 2, 85748 Garching bei München, Germany
[11] Instituto de Astrofísica, Facultad de Física, Pontificia Universidad Católica de Chile, Casilla 306, Santiago 22, Chile
[12] Centro de Astroingeniería, Pontificia Universidad Católica de Chile, Av. Vicuña Mackenna 4860, 7820436 Macul, Santiago, Chile
[13] Instituto de Estudios Astrofísicos, Facultad de Ingeniería y Ciencias, Universidad Diego Portales, Av. Ejército Libertador 441, Santiago, Chile
[14] Centro Interdisciplinario de Data Science, Facultad de Ingeniería y Ciencias, Universidad Diego Portales, Av. Ejército Libertador 441, Santiago, Chile

Received 22 November 2025 / Accepted 11 June 2026

ABSTRACT

We developed a text-to-structured query language (SQL) system based on large language models (LLMs) using in-context learning and applied it to the Automatic Learning for the Rapid Classification of Events (ALeRCE) astronomical database. ALeRCE is a community broker for the Zwicky Transient Facility and the Vera C. Rubin Observatory. The system enables users to query the database in natural language (NL) and generates executable SQL queries. To develop and evaluate the system, we constructed a set of 110 NL-SQL pairs. We propose a step-by-step generation framework comprising four modules: schema linking, query classification, prompt decomposition, and self-correction. The performance of 13 LLMs was evaluated using in-context learning and prompt engineering techniques. Text-to-SQL performance was assessed using the perfect-match (PM) rate for row identifiers (e.g., object identifiers) and column identifiers (i.e., column names). The proposed step-by-step framework consistently outperforms a direct-inference baseline, while the self-correction module consistently reduces execution errors. For Claude Opus 4.6, PM performance on row (column) identifiers is high for simple queries, reaching 0.97 (0.94), and decreases with query complexity to 0.44 (0.72) for medium queries and 0.59 (0.49) for hard queries. Among the 13 models evaluated, the best-performing LLMs for the text-to-SQL task are Claude Opus 4.6, Gemini 2.5 Pro, Gemini 3 Flash, and GPT-5.2-Codex.

Key words. astronomical databases: miscellaneous – catalogs – surveys

1. Introduction

Large language models (LLMs) are revolutionizing the field of natural language processing (NLP) and its applications. These models are pretrained on massive amounts of text data and can be fine-tuned to specific tasks such as language translation or question answering. In the field of astronomy, the main applications up to now have been in the named entity recognition (NER) task (Grezes et al. 2024; Ghosh et al. 2022; Sotnikov & Chaikova 2023; Shao et al. 2024), question-answering (Perkowski et al. 2024), hypothesis generation (Ciuca et al. 2023), text summarization (Coughlin et al. 2023), and artificial intelligence (AI) research assistants (Joseph et al. 2026; Ye et al. 2025). NER aims to identify and classify named entities in text, such as organizations, citations, surveys, telescopes, and celestial objects. For example, AstroBERT (Grezes et al. 2024) identifies organizations in the acknowledgment section of papers from the NASA Astrophysics Data System (ADS) database[1]. Astro-mT5 (Ghosh et al. 2022) identifies 32 named entities from the astrophysics literature data provided by the DEAL SharedTask team. Sotnikov & Chaikova (2023) extracted named entities from the Gamma-ray Coordinates Network (GCN) circulars, including event ID, object name, observed event, observed object, and physical phenomena. AstroLLaMA fine-tunes LlaMA-2 (Nguyen et al. 2023), leveraging a corpus of 300 000 astronomy abstracts. Downstream tasks include text generation (e.g., completing

⋆ Corresponding author: pestevez@cec.uchile.cl

[1] https://ui.adsabs.harvard.edu/

abstracts) and embedding space quality (e.g., where similar abstracts can be retrieved by computing the cosine similarity between the vector embeddings).

AstroLLaMA-chat is an enhanced version of AstroLLaMA, focused on the task of question-answering (Perkowski et al. 2024). Pathfinder (Iyer et al. 2024) is a machine-learning framework for literature review and knowledge discovery in astronomy, focused on semantic searching with natural language. It is an open-source tool that uses a corpus of more than 300 000 abstracts from NASA ADS. Ciuca et al. (2023) focused on hypothesis generation about galactic astronomy using a primal LLM for in-context learning with 1000 papers from NASA ADS, and an adversarial LLM model to critique the idea.

Text summarization is the task of generating a shorter version of a document that preserves the important information and key points while reducing the length. Coughlin et al. (2023) developed Skyportal, an open-source package designed to efficiently discover interesting transients, including multimessenger features, manage follow-up, perform characterization, and visualize results. Skyportal uses ChatGPT to provide human-readable summaries of individual sources, using redshift, classifications, and comments.

Recently, LLMs have been tested as AI agents used as research assistants. ReplicationBench (Ye et al. 2025) is a benchmark for evaluating whether AI agents can replicate entire research papers in astronomy. This includes the experimental setup, derivations, data analysis, and codebase. This is a challenging task, and the best LLM scored under 20%. ASTRO-VISBENCH (Joseph et al. 2026) is a benchmark for scientific computing and visualization in astronomy. An evaluation of state-of-the-art LLMs revealed a significant gap in their ability to serve as effective research assistants.

Another relevant application of LLMs is to convert a natural language (NL) question about a database to its corresponding structured query language (SQL) query, which must be executable. This task is called text-to-SQL (T2S) parsing. ScienceBenchmark (Zhang et al. 2023) is a database containing NL questions and SQL queries for benchmark purposes, which partially includes the Sloan Digital Sky Survey (SDSS) database[2]. However, the highest execution accuracy obtained in SDSS using LLMs reached only 33%. The authors conclude that the proposed benchmark is highly challenging and that the T2S task is far from being solved, especially for complex scientific datasets.

For this study, we developed a T2S system based on LLMs to be applied to the database of the Automatic Learning for the Rapid Classification of Events (ALeRCE). ALeRCE (Förster et al. 2021; Carrasco-Davis et al. 2021; Sánchez-Sáez et al. 2021) is a Chilean-led astronomy broker that is processing the alert stream from the Zwicky Transient Facility (ZTF; Bellm et al. 2019), and has been selected as one of the community brokers for the Vera C. Rubin Observatory and its Legacy Survey of Space and Time (LSST; Ivezić et al. 2019). Currently, ALeRCE has more than 27 000 users from 139 countries[3]. The ALeRCE database[4] can be accessed through the ALeRCE ZTF explorer[5], the SN hunter[6], a ZTF application programming interface (API) with simplified queries[7]. or directly via PostgreSQL.

However, producing SQL queries requires computational expertise and time to learn a new database. The Astronomical Data Query Language (ADQL)[8] includes astronomy-specific geometry functions, but its use has remained highly technical. Under the idea of democratizing access to data in the Rubin era, we aim to develop a system where the input is a question in NL text for the ALeRCE database, and the output is the corresponding executable SQL query. To this end, we propose a step-by-step framework for performing the T2S parsing task using in-context learning with LLMs, and compare its performance with a direct inference system. In Sect. 2, we present a historical context of T2S parsing and related work. In Sect. 3, we describe our methodology, wherein we constructed a dataset of NL questions and their corresponding SQL queries for the ALeRCE database, which we make publicly available. Section 4 presents our results, where we evaluate the performance of 13 LLMs using in-context learning and analyze the errors. In Sect. 5, we discuss possible improvements such as function calling and structured outputs. Finally, we draw conclusions in Sect. 6.

2. Background

2.1. Background on text-to-SQL parsing

The T2S task can be defined as follows (Qin et al. 2022): given a NL question Q and the corresponding database schema S=(T,C), where T stands for tables and C for columns, the goal is to generate an SQL query Y, that when executed will return results that match the user's intent (Katsogiannis-Meimarakis & Koutrika 2023). The question Q is a sequence of |Q| tokens. The database consists of |T| tables and |C| columns. Each table is described by its name, which contains multiple words. Each column within a table is also described by words. Schema linking, which consists of aligning entities in questions to tables or columns is the most important topic of T2S problems (Li et al. 2023). In addition, external knowledge can be used to improve the model's comprehension of the problem. Typically, this refers to domain-specific knowledge, for example, astronomy, as well as mathematical computation. The T2S task presents two kinds of challenges. The first is related to understanding the NL question. The NL is inherently ambiguous (Katsogiannis-Meimarakis & Koutrika 2023), i.e., the formulation of expressions is open to more than one interpretation. The second is that SQL has a strict syntax, which leads to limited expressivity compared to NL. This makes building the syntactically and semantically correct SQL query difficult based on the underlying database schema.

For evaluating T2S systems, several database benchmarks have been created, such as Spider (Yu et al. 2018), BIRD (Li et al. 2023), and ScienceBenchmark (Zhang et al. 2023). The Spider dataset contains 10 181 NL questions and 5693 unique SQL queries over 200 databases belonging to 139 different domains, allowing it to deal with the cross-domain T2S parsing task (Qin et al. 2022). The queries are divided into four levels of difficulty: easy, medium, hard, and extra hard. However, Spider mainly contains simple databases with few tables, columns, and entries, which rarely reflect real-world problems (Zhang et al. 2023). BIRD is a large-scale cross-domain benchmark covering 95 databases from 37 domains. Li et al. (2023) performed an error analysis using GPT-3.5 Turbo in the T2S task on BIRD. The authors found that the most common error is wrong schema linking (41.6%), i.e., erroneous association of tables and columns to

[2] https://skyserver.sdss.org/
[3] Estimation from Google Analytics.
[4] https://science.alerce.online/
[5] https://alerce.online/
[6] https://snhunter.alerce.online/
[7] https://api.alerce.online/ztf/v1
[8] https://www.ivoa.net/documents/ADQL/20180112/

the NL question. The second most common error was misunderstanding database content (40.8%), i.e., failing to recall the correct database structure (e.g., using a column that does not belong to a specific table) or generating fake schema items (e.g., calling a table or column that does not belong to the database). However, the previous two database benchmarks do not contain domain-specific knowledge of astronomy. ScienceBenchmark is a new database benchmark that contains three real-world and domain-specific databases: research policy-making, astrophysics, and cancer research. A subset of the SDSS database is used (5 out of 10 tables) due to limitations in the number of tokens allowed by LLMs. For this database, 200 NL-SQL pairs were manually generated by a team of domain and SQL experts, and data augmentation was carried out. The authors found that state-of-the-art T2S algorithms that obtained over 80% of execution accuracy on Spider, achieved only 21% execution accuracy on SDSS. In addition, GPT-3.5 with prompting achieved only 33% execution accuracy. This poor performance is presumably because SDSS is a domain-specific database that includes many numerical values and requires the use of functions and mathematical operators. In addition, it contains many column names and values labeled with domain-specific abbreviations, e.g., z for redshift. The authors concluded that the current state-of-the-art approaches do not perform well on real-world problems, and that we need domain-specific benchmarks for training and evaluating T2S systems in scientific domains.

2.2. Prompt engineering for the T2S task

Large language models are currently models with billions of parameters, trained on massive amounts of text data. According to Zhao et al. (2026), LLMs show emergent abilities such as in-context learning, instruction following, and step-by-step reasoning. In-context learning allows LLMs to convert a NL question into a SQL query using a prompt text, i.e., it is an inference task (there is no training or fine-tuning). The prompt usually includes four components: a task instruction, a NL question, the database schema, and external knowledge (Chang & Fosler-Lussier 2023). Figure 1 shows the four components of a basic prompt for the T2S task, which is the basic inference task using in-context learning, and serves as a baseline for comparison purposes. We call this scheme direct query generation.

Several studies have shown that the ability of LLMs for complex reasoning can be improved using step-by-step reasoning (Kojima et al. 2022). There are several prompting techniques for multi-step reasoning, such as chain of thought (CoT; Wei et al. 2022), which is able to guide LLMs through a logical reasoning chain, mimicking how humans break down problems into logical intermediate steps (Sahoo et al. 2024). A variant is the logical chain-of-thought (Liu et al. 2023) prompt, which includes effective verification mechanisms to reduce logical errors and hallucinations through a think-verify-revise loop (Zhao et al. 2024). Regarding code generation, structured chain-of-thought (SCoT) prompting incorporates program structures (sequence, branch, and loop structures) into reasoning steps (Li et al. 2025).

The Decomposed In-Context (DIN)-SQL method (Pourreza & Rafiei 2023) decomposes the text-to-SQL task into four modules: (1) schema linking, (2) query classification and decomposition, (3) SQL generation, and (4) self-correction. A prompt-based module is designed for schema linking, which includes 10 random samples from the training set of the Spider dataset. For each mention of a column name in the question, the corresponding column and its table are selected from the given database schema. The second module classifies each query into one of three classes: easy, non-nested complex, and nested complex. The class labels are essential for the query generation module, which uses different prompts for each query class. A simple few-shot prompt is performed without intermediate steps for the questions in the easy class. In addition to class labels, the module also detects the tables to be joined for non-nested and nested queries. The nested complex class is the hardest and requires several intermediate steps before generating the final answer. The prompt for this class is designed in a way that the LLM first solves sub-queries and then uses them to generate the final answer. Finally, a self-correction module is added, where only buggy code is provided to the model, and it is asked to fix the bugs in a zero-shot setting.

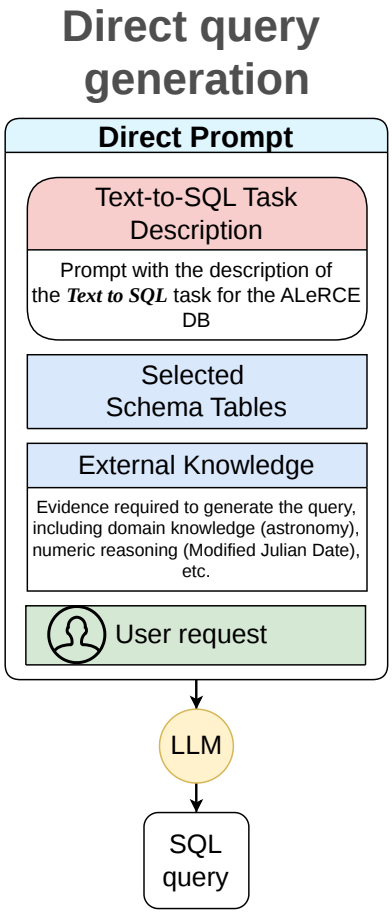


Fig. 1. Four components of a basic prompt for the T2S task. This approach is called direct query generation.

The DAIL-SQL (Gao et al. 2024) proposes a new prompt engineering method focused on question representation and in-context learning for text-to-SQL. Among the question representations studied are a basic prompt (table schema and question in NL), text-representation prompt (add task instruction to the basic prompt), OpenAI demonstration prompt (ODp) (the instruction is more specific, constraining how the model must answer), and code representation prompt (CRp) (it presents the T2S task in SQL syntax, e.g., the NL questions appear inside SQL comments). Regarding in-context learning, DAIL-SQL focuses on example selection and example organization. Among the example selection strategies are random selection, question-similarity selection (select k examples with the most similar questions), query-similarity selection (select k examples similar to the target

SQL query), and DAIL selection (consider both questions and queries to select examples). The experiments were performed with the Spider and Spider-realistic datasets. In the zero-shot scenario, the ODp representation performed best in GPT-3.5 Turbo, but the basic prompt performed best in GPT-4. Adding foreign key information, i.e., relationships between tables in a relational database, significantly improved the execution accuracy of LLMs. Adding the rule to generate SQL queries "with no explanation" consistently improved the performance of all LLMs. DAIL selection generally outperformed other example selection strategies. Similar approaches to DIN-SQL and DAIL-SQL have been presented in MAC-SQL (Wang et al. 2025), and C3 (Dong et al. 2023).

3. Methodology

3.1. ALeRCE database

In early 2026, the ALeRCE database had 25 tables and 304 columns; see the entity-relationship diagram in the Appendix A. It contains information about astrophysical objects, including global and band-dependent statistics, time- and band-dependent flux evolution, cross-matches, and machine-learning probabilities. For example, the ALeRCE light curve classifier (Sánchez-Sáez et al. 2021) computes machine learning probabilities for 15 classes: SNIa, SNIbc, SNII, SLSN, QSO, AGN, Blazar, CV/Nova, YSO, LPV, EB, DSCT, RRL, CEP, and Periodic-Other. The main tables are "object" (22 columns), "probability" (6 columns), "magstat" (28 columns), "detection" (30 columns), "non_detection" (4 columns), "forced_photometry" (42 columns), and "feature" (5 columns). These tables contain information about each object, including time and spectral band statistics. The organization of the database is centered on the "object" and "probability" tables, which contain the attributes and classification of the objects, respectively, and are requested in most of the queries requiring JOIN[9] commands or sub-queries. To collect NL-SQL pairs, we first included 21 examples generated by the ALeRCE team and presented in workshops with the aim of teaching users how to interact with the database. In addition, a group of 10 astronomers was tasked with generating real-world questions, with the only restriction that the information to answer these questions must be obtainable from the ALeRCE database alone. In this way, another 89 examples were generated. Finally, an SQL expert built the target gold queries, resulting in a total of 110 NL-SQL pairs obtained. The listing 1 shows an example of a query, where ranking=1 means taking the highest probability only.

```
User Request:
Give me the objects classified as YSO by their
lightcurves (with a probability higher than 0.7)
Query:
SELECT
    oid, probability
FROM
    probability
WHERE
    classifier_name='lc_classifier' AND
    class_name='YSO' AND
    ranking=1 AND
    probability > 0.7
```

Listing 1: Example of a question in natural language and its corresponding SQL query.

[9] Joining in SQL means retrieving data from two or more tables based on a common field.

Most of the SQL queries included in our gold set represent typical requests found in the literature, which show how the community has already used ALeRCE. These include: requesting light curve detections for given objects (e.g., Magee et al. 2023; Gomez & Gezari 2023; Müller-Bravo et al. 2025; Alexander et al. 2026); radial queries (e.g., Pomeroy & Norris 2024); cross-matches with objects from a given class in one of the ALeRCE classifiers (e.g., López-Navas et al. 2022; Arévalo et al. 2024; Oshikiri et al. 2024). Meanwhile, other queries are highly specific, including, e.g., queries to individual features, or filters through tables other than object and probability (data quality, the Panoramic Survey Telescope & Rapid Response System (PS1)[10] data, Wide-field Infrared Survey Explorer (WISE)[11] data, and solar system data); while these requests are expected to be less frequent, they allow us to clean object samples and/or get a broader understanding of the selected objects, and thus deserve to be included in our gold set.

3.2. Query classification

Queries were classified into three difficulty levels: simple, medium, and hard. This was done first to evaluate how the LLM performance varies with query difficulty and, second, to allow generating a different prompt depending on the level of difficulty, as explained in Sect. 3.5. To this end, we define the tables "object", "probability", and "magstat" as basic tables (the most commonly used) and the remaining tables as general. The definitions of the categories are as follows:

- Simple: queries using up to two basic tables or queries using a single general table.
- Medium: queries that require one general and one basic table, two general tables, or three basic tables. This category also includes queries that contain two simple sub-queries—each comparable in complexity to a simple query—or one sub-query of higher complexity.
- Hard: Any query requiring more than three tables or using more than one non-simple sub-query for optimization.

Based on these, the ALeRCE dataset of 110 NL-SQL pairs was partitioned into a development set of 58 examples and a testing set of 52 examples. Note that when using in-context learning, there is no need for a training set, since there is no parameter adjustment. The development set is used to develop and evaluate various prompt methods. The testing set is used exclusively for the final evaluation. The 58 examples in the development set consisted of 32 simple, 14 medium, and 12 hard queries. The 52 examples of the testing set were composed of 32 simple, 10 medium, and 10 hard queries. Examples of simple, medium and hard queries are given in Appendix B.

3.3. LLM models

Table 1 describes the 13 LLM models used in this work. This includes the full name of the API, the abbreviations of the names used in what follows, and the context length. The latter is the maximum number of tokens (input + output) that an LLM can process in a single forward pass.

3.4. Basic prompt

The basic inference model shown in Fig. 1 is used as a baseline for comparison purposes. In the zero-shot case, the information

[10] https://catalogs.mast.stsci.edu/panstarrs
[11] https://irsa.ipac.caltech.edu/Missions/wise.html

Table 1. Description of the LLM models.

API model	Abbreviation	Context length
gpt-4o-2024-11-20	GPT-4o	128 000
gpt-4.1-2025-04-14	GPT-4.1	1 048 576
gpt-5-2025-08-07	GPT-5	400 000
gpt-5.2-2025-12-11	GPT-5.2	400 000
gpt-5.2-codex	GPT-5.2-Codex	400 000
gpt-5.3-codex	GPT-5.3-Codex	400 000
claude-3-7-sonnet-20250219	Claude 3.7	200 000
claude-sonnet-4-5-20250929	Claude Sonnet 4.5	200 000
claude-opus-4-6	Claude Opus 4.6	200 000
gemini-2.5 flash	Gemini 2.5 Flash	1 048 576
gemini-2.5 pro	Gemini 2.5 Pro	1 048 576
gemini-3-flash-preview	Gemini 3 Flash	1 048 576
gemini-3.1-flash-lite-preview	Gemini 3.1 Flash	1 048 576

given to the model includes the requested T2S task, the database schema, external knowledge (which is query-dependent), and general information about the database structure. The format of the zero-shot prompt is shown in Listing 2. Optionally, a few examples could be added to the prompt, as in the so-called few-shot case. In this work, we mainly deal with the zero-shot case. The few-shot case is presented in Sect. 4.2 for the best LLM models.

```
{General_Task}

# Context:
{General_Context}

# The Database has the following tables that can be
    used to generate the SQL query:
{tables_schema}

{Final_Instructions}

# Important Information for the query
{External_Knowledge}
# User Request: ''{query}''
```

Listing 2: Zero-shot prompt format.

3.5. Proposed framework for the T2S task

Our end goal is to develop a practical virtual assistant for ALeRCE users. With this aim, we decompose the prompt into different steps that run autonomously. Our approach is inspired by previous work such as DIN-SQL (Pourreza & Rafiei 2023) and DAIL-SQL (Gao et al. 2024). The proposed framework for performing the T2S task in the ALeRCE database has four steps: schema linking, classification, decomposition, and self-correction. The first step, schema linking, is shown in Fig. 2. Steps 2 and 3 are shown in Fig. 3. Finally, Fig. 4 shows the self-correction step, the last step of the proposed framework. A detailed description of the four steps of the proposed framework is as follows:

1. Schema linking: The aim is to align the question with the database schema, i.e., to identify the required tables and columns. With this aim, an independent module was designed to extract the required structure from the database; see Fig. 2. This is carried out in two stages. The first identifies the required tables and the second determines the columns. Each stage is performed using zero-shot prompting. For the task of extracting the tables, the model is provided with a question, a description of the tables in the database, and it is instructed to perform the schema linking task. The columns are extracted independently for each of the previously selected tables. A different API call is started for each table, with a prompt describing the columns contained in the table. The outputs of this step are the selected schema tables. The prompt for the schema linking is shown in the Appendix C, Listing C.1.
2. Classification: this step determines the level of difficulty of the query. The prompt for classifying the level of difficulty

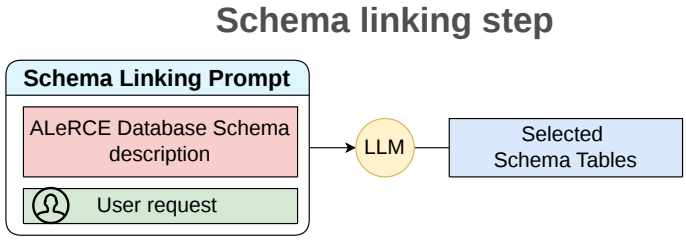


Fig. 2. Diagram of the schema linking, the first step of the proposed framework for the T2S task.

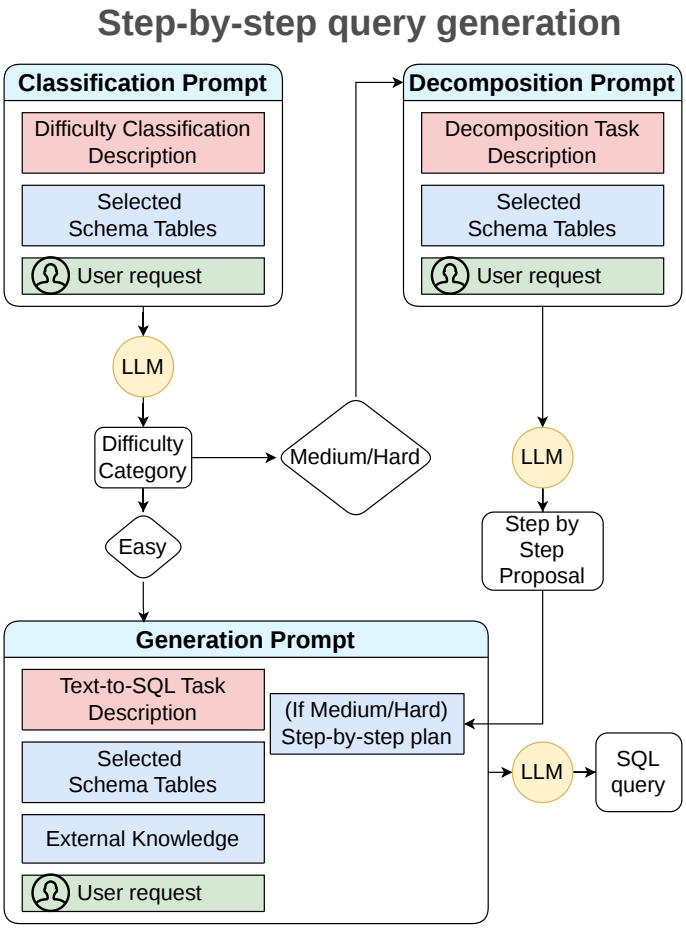


Fig. 3. Step-by-step query generation approach includes a classification prompt and a decomposition prompt. The classification prompt aims to discriminate between simple, medium, and hard queries. The decomposition prompt divides the problem into sub-problems for medium and hard queries. Simple queries skip the decomposition and go directly to the generation prompt.

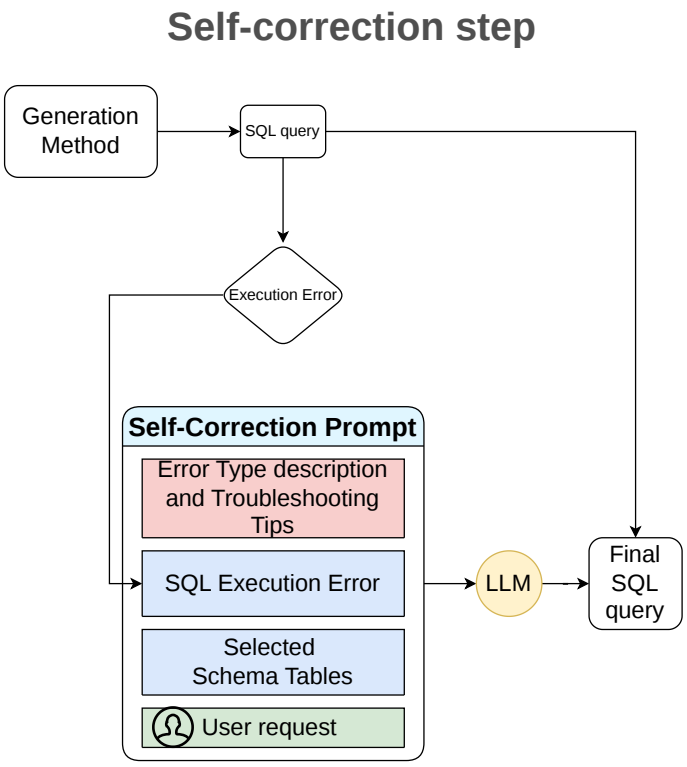


Fig. 4. Self-correction step generates a prompt specific to each type of SQL error: timeout, non-existing structures, or general errors.

uses the structure shown in the Appendix C, Listing C.2. The prompt includes the definition of the query classes by difficulty, the tables and columns obtained in the schema linking stage, and the user query.

3. Decomposition: as shown in Fig. 3, a decomposition prompt is performed for the medium and hard queries. For simple queries, decomposition is not needed; they go directly to the generation prompt block. Otherwise, we decompose the problem into sub-problems. This is useful for complex queries that require different sub-queries. We tested least-to-most prompting (Zhou et al. 2023) and decomposed prompting (Khot et al. 2023). A new module is used to decompose the queries that require complex sub-queries or JOINs. The input to this module is the information extracted in the previous stages.
4. Self-correction: recent LLM models, such as GPT-5.2-Codex, Claude Opus 4.6 or Gemini 3 Flash, are able to correct or improve their original response. A new agent can be used to get feedback about previous responses. The self-correction module addresses the three main types of execution errors found in the generation of queries: timeout; non-existing structures, such as confusion of column and table names, or non-existent columns, tables, functions, etc.; and general errors such as syntax, missing FROM-clauses, and so on. A schema of the self-correction step is shown in Fig. 4. The self-correction step consists of a single iteration and is triggered only when the generated SQL query produces an execution error, so it is not applied to every query. This step is particularly effective at fixing syntax errors and undefined or ambiguous references. Timeout errors can also be corrected when the model identifies the minimal conditions and clauses needed to produce a valid result within the time limit. A different zero-shot prompting is used for each type of error. For the timeout error (more than 2 minutes), it is assumed that there is an optimization problem. The agent is asked to improve or re-order the query according to the error found. In addition, advice on avoiding this type of error is provided. The main structure of the prompt for timeout errors is shown in Appendix C, Listing C.3. A similar structure is used for general errors and non-existent structures, but focusing the prompt on the syntax and the database structure errors.

3.6. Prompt engineering

The prompt is separated into system messages and user messages, as done in OpenAI[12]. The system messages include the general task, the ALeRCE context, and the database schema. The user messages include the user's question and the external knowledge, if needed. For the general task, a personality as a data scientist expert is given:

> "As an SQL expert with a willingness to assist users. . . "

The general context describes the ALeRCE pipeline, the objects and classes used, as well as details on how ALeRCE works (including the probability assigned to objects). The following is an example:

> "ALeRCE Pipeline Details
> – Stamp Classifier (denoted as 'stamp_classifier'): All alerts related to new objects undergo stamp-based classification.
> – Light Curve Classifier (denoted as 'lc_classifier'): A balanced hierarchical random forest classifier employing four models and 15 classes.
> ..."

For the database schema, commands to create the ALeRCE SQL tables are given:

> "CREATE TABLE object (/* this is the most important table. It contains the main statistics of an object, independent of time and band */
> oid VARCHAR PRIMARY KEY
> ...)"

In addition, instructions on the format of the answers are given:

> "Answer ONLY with the SQL query ...", "... Do NOT CHANGE the names of the tables or columns unless the user explicitly asks you to do ..."

For the step-by-step method, different prompts were generated for each step. For the schema linking step, the prompt included a brief description of the task, a description of the tables of the database, and specifications regarding when to select the object, probability, and feature tables. This is because the object and probability tables are designed for frequent use and have been indexed accordingly, while the features table is designed for more advanced use cases and is generally more expensive to query. For the classification step, a prompt with the classification task, a description of the three query categories, the table schema selected from the schema linking step (including column descriptions), instructions associated with the classification, and the user request are used.

[12] https://platform.openai.com/docs/guides/prompt-engineering/#messages-and-roles

For the decomposition step, the query generation was split into two stages. In the first stage, the LLM is asked to generate a step-by-step plan to generate the query, including all requirements, conditions, and details. In the second stage, the LLM is requested to generate the query following the plan generated in the first stage. For medium queries, a prompt example for the first stage is the following:

> "Your task is to DECOMPOSE the user request into a series of steps required to generate a PostgreSQL query ...".

For hard queries, a higher emphasis is put on the details and order of the query structure, as follows:

> "The request is a very difficult and advanced query, so you will need to use JOIN, INTERSECT, and UNION statements, together with Nested queries. It is very important that you give every possible detail in each step, ...".

Following BIRD (Li et al. 2023), we add external knowledge to the prompts in order to provide the information needed to solve the problem, but that is not present in the dataset, such as numeric reasoning knowledge, domain knowledge, and synonym knowledge. This external knowledge is specific to each query. The following is an example of the external knowledge required to solve a simple query:

> Request: "Query all objects that were first classified as SN by the stamp classifier between August 17 and August 21, 2024, with a probability greater than 0.5 or at least two detections."
> External Knowledge:
> – MJD date for August 17 2024 = 60173.0
> – MJD date for August 21 2024 = 60177.0

Notice that we used the same prompts for all LLMs, following established prompt engineering practices. This independence of the LLM provider allows us to make fair comparisons and to use these prompts with new versions of the LLMs, when the old models become deprecated. In addition, it allows the users to select an LLM of their own choice.

3.7. Evaluation metrics

The most common metric used in T2S is the execution accuracy (EX). EX is the proportion of questions in the test set for which the execution results of both the predicted and ground-truth inquiries are identical (Li et al. 2023). In this work, we propose a modified EX, to consider partially correct results. We compute the set of row identifiers, called ids, in the ground truth, O_t, as well as the set of the predicted ids, O_p. In the ALeRCE database, examples of row identifiers are 'object identifier' (oid), candidate identifier (candid), oid_catalog, objectidps1 (object id closest source from PS1 catalog), classifier_name, or count.

For this binary classification problem, precision and recall are calculated. For each predicted query (k, l), where k is the user query identifier and l is the query run identifier (e.g., an index from 0 to 9 for 10 runs of the user query k), the recall metric is calculated by counting the predicted identifiers ($id_i^{k,l}$, where i is the row number among the predicted values) that match the set of true ids (O_t^k), divided by the cardinality of the set of true ids, $|O_t^k|$:

$$r_{k,l} = \frac{\sum_{i \in \text{pred}} score(id_i^{k,l}, O_t^k)}{|O_t^k|}, \tag{1}$$

where

$$score(id, O) = \begin{cases} 1, & id \in O \\ 0, & id \notin O \end{cases}. \tag{2}$$

Likewise, for each user query k and query run l, the precision metric is calculated by counting the matches between the true ids ($id_i^{k,l}$, where i is the row number among the true values) and the set of predicted ids ($O_p^{k,l}$), and dividing this amount by the cardinality of the set of predicted ids, $|O_p^{k,l}|$:

$$p_{k,l} = \frac{\sum_{i \in \text{true}} score(id_i^{k,l}, O_p^{k,l})}{|O_p^{k,l}|}. \tag{3}$$

The same can be done in the column dimension, where, for example, an oid can have associated values such as firstmjd, lastmjd, ndet (number of detections), and so on. Thus, we define $r_{k,l}^{rows}$ and $p_{k,l}^{rows}$ as the recall and precision computed along the row identifiers. Likewise $r_{k,l}^{cols}$ and $p_{k,l}^{cols}$ are the recall and precision computed along the column identifiers.

In this work, we assume that an answer containing all and only the expected rows is correct. On the other hand, we assume that an answer containing all but not necessarily only the expected columns is correct. This is because the latter will not significantly impact the user experience. According to this assumption, we evaluate all prompting methods using a metric of perfect matching rates (PM), defined as follows:

$$PM = \frac{1}{n_{\text{queries}}\, n_{\text{runs}}} \sum_{k \in \text{queries}} \sum_{l \in \text{runs}} N_{\text{perfect}}(k, l), \tag{4}$$

where n_{queries} is the total number of queries, and n_{runs} is the number of runs per query, typically 10. $N_{\text{perfect}}(k, l)$ is defined as:

$$N_{\text{perfect}}^{\text{rows}}(k, l) = \begin{cases} 1, & \text{if } p_{k,l}^{\text{rows}} = r_{k,l}^{\text{rows}} = 1 \\ 0, & \text{otherwise} \end{cases}, \tag{5}$$

$$N_{\text{perfect}}^{\text{cols}}(k, l) = \begin{cases} 1, & \text{if } r_{k,l}^{\text{cols}} = 1 \\ 0, & \text{otherwise} \end{cases}, \tag{6}$$

for the case of rows and columns, respectively. Accordingly, we define PM_{rows} and PM_{cols} as metrics for perfect matches among the rows or columns, respectively.

4. Results

In this section, we present results comparing the performance of direct versus step-by-step (sbs) query generation methods, with and without self-correction, as a function of query difficulty across the 13 LLMs. In addition, we analyze the errors obtained. Table D.1 in Appendix D shows the PM results. For evaluation, we conducted 10 runs and computed the average PM and standard deviation across these runs. Consistency varies between LLM models, even when using a zero temperature. For earlier models, such as Claude 3.7, GPT-4o and GPT-4.1, SQL outputs vary more noticeably for medium and hard queries, where more tokens are required and thus the probability of divergence increases (see the standard deviation in Table D.1). Newer models have more deterministic behavior, notably Gemini 3 Flash, Claude Sonnet 4.5, and Claude Opus 4.6. It can be observed in

Table 2. General statistical ranking (with self-correction).

Model (method)	RS	RM	RH	CS	CM	CH	R-SUM	C-SUM	SUM
Claude Opus 4.6 (SbS)	1	1	1	1	3	3	3	7	10
Gemini 2.5 Pro (SbS)	3	1	3	2	2	2	7	6	13
Gemini 3 Flash (Direct)	2	2	2	4	2	1	6	7	13
GPT-5.2-Codex (SbS)	3	1	4	2	2	2	8	6	14
Gemini 3 Flash (SbS)	2	2	3	2	2	3	7	7	14
Claude Sonnet 4.5 (SbS)	2	2	4	3	1	3	8	7	15
GPT-5.3-Codex (SbS)	3	2	3	2	3	2	8	7	15
Claude Opus 4.6 (Direct)	2	2	2	4	2	4	6	10	16
Gemini 2.5 Pro (Direct)	3	2	3	4	2	3	8	9	17
GPT-5.2-Codex (Direct)	4	2	4	4	2	2	10	8	18
Claude Sonnet 4.5 (Direct)	3	1	5	5	3	3	9	11	20
GPT-5.3-Codex (Direct)	5	3	4	5	4	2	12	11	23

Notes. An unpaired permutation test on per-run perfect-match scores (Holm-Bonferroni correction, $\alpha = 0.05$) was performed.

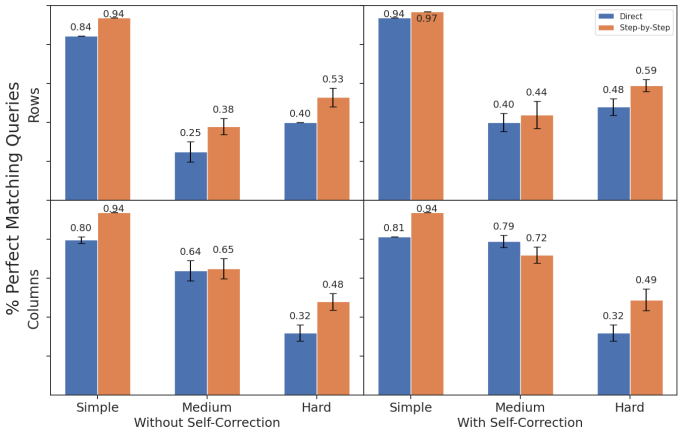


Fig. 5. Claude Opus 4.6 perfect matching performance for rows and columns, with and without self-correction, for the direct and step-by-step query generation methods.

Table D.1 that when comparing the performance of each LLM independently, without (w/o) and with (w/) self-correction, in all cases, the self-correction improves or achieves a similar result to that without self-correction.

Fig. 5 shows the performance of Claude Opus 4.6 w/ and w/o self-correction at different levels of query difficulty. When applying self-correction, the results improve or remain the same for all kinds of queries for both the direct query generation and the sbs method. For example, the performance of row ids improves from 0.84 (w/o self-correction) to 0.94 (w/ self-correction) when using the direct query generation method, and from 0.94 to 0.97 when using the sbs query generation method.

Table 2 shows a general statistical ranking of the top 6 LLMs according to their PM performance, using both sbs and direct query generation methods with self-correction. The top 6 LLMs in our experiments are: Claude Opus 4.6, Claude Sonnet 4.5, Gemini 2.5 Pro, Gemini 3 Flash, GPT-5.2-Codex and GPT-5.3-Codex. We used an unpaired permutation test to rank the LLMs according to their performance on rows (R) and columns (C), for simple (S), medium (M), and hard (H) queries. A rank of 1 in the table denotes the best-performing LLM. This rank may be assigned to multiple models when pairwise performance differences are not statistically significant. Next, we aggregated the results to obtain a final sum, where a lower number indicates a better ranking. It can be observed that the sbs query generation methods obtained a better ranking than the direct query generation methods in all cases, except for Gemini 3 Flash, which direct query generation version ranked third. Claude Opus 4.6 (sbs) and Gemini 2.5 Pro (sbs) are ranked first and second, respectively.

Table 3 shows the average total cost per run of the whole test set in US dollars, as well as its split into different stage components, for the top 6 LLMs. Using the cost of Gemini 2.5 Pro (sbs) as a reference, the cost increases 2.41 times when using Claude Opus 4.6 (sbs), and 1.01 times when using GPT 5.2 Codex (sbs). Gemini 3 Flash has the lowest cost for both the direct and sbs query generation methods. Table E.1 in Appendix E shows the maximum total token consumption, broken down by pipeline stage, LLM model and query generation method. Importantly, all stages remain well within the context window limits of the 13 LLMs evaluated.

Fig. 6 shows the performance of both query generation methods with self-correction using Gemini 2.5 Pro, measured as the rate of PM queries for row and column identifiers, as a function of query difficulty. It can be observed that the proposed sbs framework obtained higher mean values than the direct method in all cases, but there are no statistically significant differences according to the permutation test[13], except for medium queries (rows) and simple queries (columns).

The right side of Fig. 5 shows the same experiment with Claude Opus 4.6. It can be observed that the proposed sbs framework outperforms the direct method for simple and hard queries. The differences are statistically significant according to the permutation test, except for medium queries where there is a draw. The performance of sbs is high for simple queries (e.g., 0.97 for row ids) and decreases with query difficulty, from 0.44 for medium to 0.59 for hard queries. The columns' performance follows a similar pattern, highlighting that medium queries achieve a perfect match of 0.72. Across all queries, the direct query generation method using Claude Opus 4.6 outperformed or performed similarly to the Gemini 2.5 Pro direct query generation method. In addition, when comparing the results of the step-by-step method, Claude Opus 4.6 outperformed or performed similarly to Gemini 2.5 Pro for simple and hard queries.

Figure 7 shows the performance of the top 6 LLMs using the direct method, in terms of the percentage of PM queries for row and column identifiers, as a function of the degree of difficulty

[13] https://rasbt.github.io/mlxtend/user_guide/evaluate/permutation_test/

Table 3. Average total cost per run in US dollars by LLM model and query generation method.

LLM (query generation method)	Total	Schema	Difficulty	Decomp	SQL	Self-correction	Ratio
Claude Opus 4.6 (Step-by-step)	**2.87**	0.23	**0.43**	**1.07**	1.05	0.09	**2.41**
Claude Opus 4.6 (Direct)	1.55	0.23	–	–	**1.19**	**0.13**	1.30
GPT-5.2-Codex (Step-by-step)	1.20	0.15	0.15	0.43	0.38	0.10	1.01
Gemini 2.5 Pro (Step-by-step)	1.19	**0.43**	0.11	0.29	0.30	0.06	1.00
Gemini 2.5 Pro (Direct)	0.85	**0.43**	–	–	0.34	0.08	0.72
GPT-5.2-Codex (Direct)	0.82	0.15	–	–	0.56	0.11	0.69
Claude Sonnet 4.5 (Step-by-step)	0.71	0.14	0.06	0.26	0.14	0.11	0.60
GPT-5.3-Codex (Step-by-step)	0.61	0.07	0.07	0.19	0.21	0.06	0.51
Claude Sonnet 4.5 (Direct)	0.39	0.14	–	–	0.16	0.08	0.33
GPT-5.3-Codex (Direct)	0.34	0.07	–	–	0.20	0.07	0.28
Gemini 3 Flash (Direct)	0.30	0.08	–	–	0.20	0.02	0.25
Gemini 3 Flash (Step-by-step)	0.27	0.08	0.04	0.06	0.08	0.01	0.23

Notes. Values are aggregated over all queries from the test set and averaged over 10 runs. The cost is split into each stage of the query. The ratio compares the total cost against Gemini 2.5 Pro (sbs).

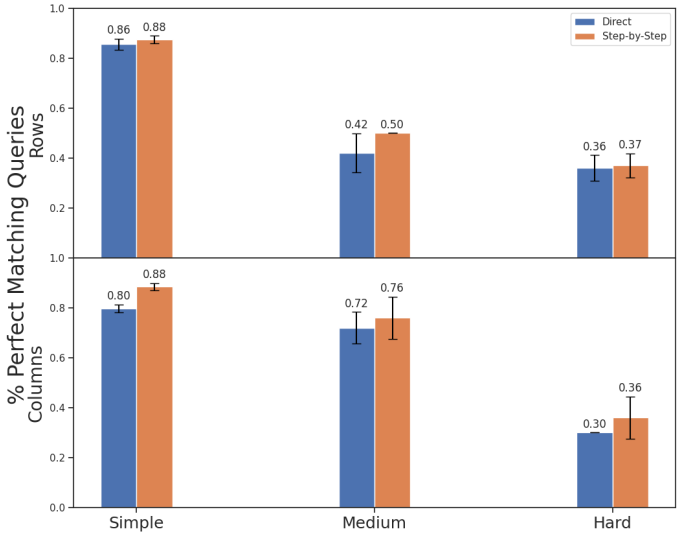


Fig. 6. Comparison of direct versus step-by-step PM performance with self-correction using Gemini 2.5 Pro.

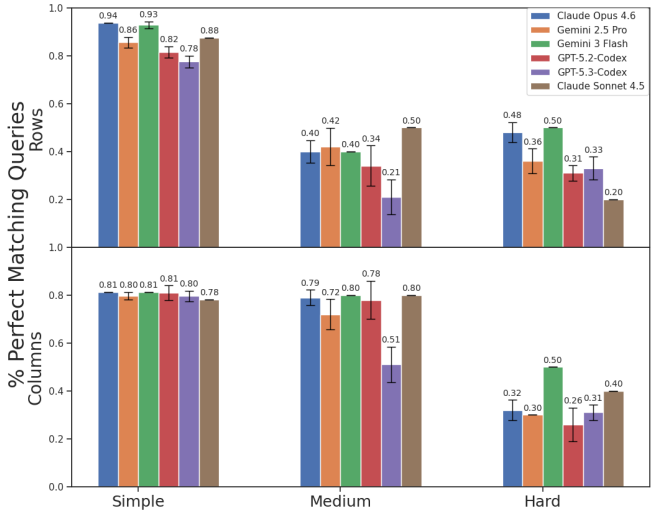


Fig. 7. Performance comparison of the top 6 LLMs when using the direct method, as a function of the level of query difficulty.

of the query. Likewise, Fig. 8 shows the performance of the top 6 LLMs using the step-by-step method. Figs. E.1, E.2, and E.3 in Appendix E show the progression over time of the LLM performance according to each provider for the models evaluated (Claude, Gemini and GPT) for our T2S task. The results of using GPT-5.2-Codex are shown in Fig. E.4.

4.1. Analysis of errors

Three main kinds of execution errors can be distinguished. A timeout error occurs when a query exceeds ALeRCE's 2-minute execution limit, e.g., when the model omits a specific condition needed to limit the volume of the data requested. Alternatively, the model may misinterpret the request, producing an incorrect condition, or place a condition in the wrong scope (e.g., inside a subquery when it should be applied at the outer level). The second category includes undefined objects, tables, relations, columns, and so on – e.g., names of tables or columns that are not in the database. The third category includes syntax and other errors, such as: datatype mismatch (e.g., UNION types and double precision cannot be matched); cardinality violation (e.g.,

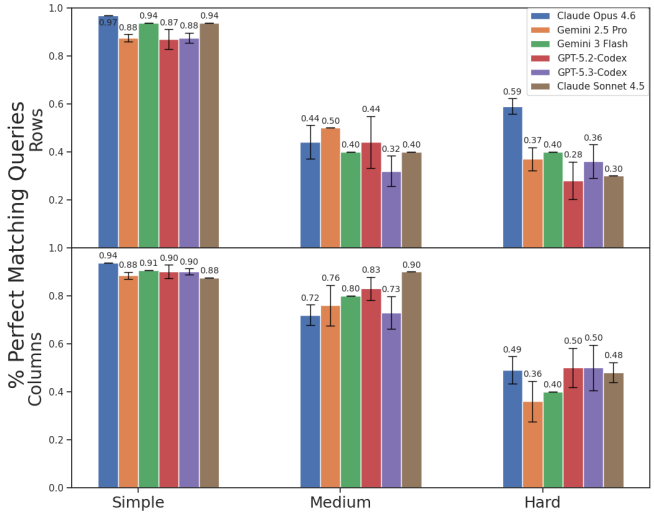


Fig. 8. Performance comparison of the top 6 LLMs when using the step-by-step method, as a function of the level of query difficulty.

Table 4. Example of parsing columns.

Predicted SQL-Query	`SELECT object.oid AS ztf_oid, allwise.oid_catalog AS allwise_oid_catalog, xmatch.dist AS distance ...`
Extracted aliased columns	'ztf_oid', 'allwise_oid_catalog', 'distance', ...
Extracted real columns	'oid', 'oid_catalog', 'dist', ...
Gold columns	'oid', 'oid_catalog', 'dist', ...

more than one row returned by a subquery used as an expression); and ambiguous column (e.g., selecting the same column name from two tables without specifying the table reference).

An error that we found during our system's development was the LLMs' tendency to rename columns. For example, in Table 4, the golden column names 'dist' and 'oid_catalog' were changed to the predicted columns 'distance' and 'allwise_oid_catalog', respectively. The first error is semantically and logically correct, but it does not match the real name. We solved this problem by parsing the queries after execution using the Python library "sqlparse".

Figure 9 compares the execution errors obtained w/ and w/o self-correction when using Claude Opus 4.6. The vast majority of errors are due to timeout. Without self-correction, there are errors associated with undefined columns and tables, cardinality violation, and invalid column references. In Figure 9 all non-timeout errors are corrected by self-correction, except one undefined column.

In addition to execution errors, there are semantic errors: the query is executable but does not, or does not always, produce the intended results. Semantic errors encompass a broad range of issues, among them logical errors (e.g., confusing similarly named columns), language comprehension, ambiguity issues and cases where the model lacks specialized domain knowledge. These kinds of errors are not corrected by self-correction. Fig. 10 shows that when using the step-by-step method with Claude Opus 4.6, semantic errors are generally reduced or kept similar to those of the direct method. It can be observed that for medium queries the percentage of semantic errors is much higher for rows than for columns. This discrepancy appears because column evaluation is much simpler than row evaluation. Row precision and recall require the model to satisfy multiple conditions simultaneously, including correctly identifying join paths, applying filters, and constructing appropriate WHERE clauses. Column selection, on the other hand, can be viewed as a classification or association task; the LLM must identify which attributes are relevant to the query and ensure that the generated SQL executes successfully against the database schema. At the medium level, models can still perform the column association

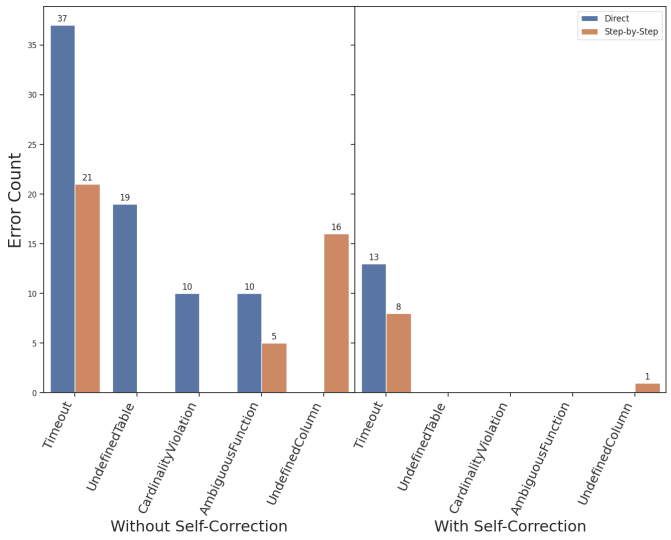


Fig. 9. Number and type of execution errors using Claude Opus 4.6, a) without self-correction and b) with self-correction, for the direct and step-by-step query generation methods.

reliably but begin to struggle with the more compositional reasoning that row-level accuracy demands. The analysis of errors for GPT-5.2-Codex and Gemini 3 Flash are shown in Figs. E.5, and E.6 in Appendix E.

4.2. Few-shot prompting

A few-shot prompt for text-to-SQL is a prompting strategy where an LLM is given a small number of NL-SQL pairs before asking it to generate SQL for a new question. We tested Claude Opus 4.6 and Gemini 2.5 Pro with few-shot prompting. There are three main strategies for example selection in the literature: random, it randomly samples k examples from the available candidates; question similarity selection (QTS; Liu et al. 2022), it chooses k examples with the most similar questions; masked question similarity selection (MQS; Guo et al. 2023), it replaces table names, column names, and values in all questions with a mask token, and computes the similarities of the embeddings with the k-Nearest Neighbor (kNN) algorithm. We made a few changes in the QTS and MQS strategies, in order to adapt them to our dataset. For both strategies, we used a similarity search through embeddings, cosine similarity and top-k selection. For MQS we masked the following elements: ZTF oids, numerical values, month names, and ALeRCE classes. Table 5 shows the results of 1-shot and 3-shot prompting when using Gemini 2.5 Pro (direct). The best results are obtained with the MQS strategy using 3-shot, which achieves a statistically significant increase for all types of queries, as well as row and column ids, with respect to the 0-shot. Notably the row (column) id performance increased 18 (14) points. QTS 3-shot also achieves statistically significant differences with 0-shot, except for a draw in hard rows. Tables F.1 and F.2 in Appendix F show the results of few-shot prompting for the Gemini 2.5 Pro (sbs) and Claude Opus 4.6 (sbs), respectively. Listing F.1 shows an example of few-shot prompt.

5. Discussion

In this section, we discuss two possible ways to improve our proposed framework, especially from the viewpoint of putting in production our T2S system in the near future: structured outputs and function (tool) calling. Regarding specialized tools,

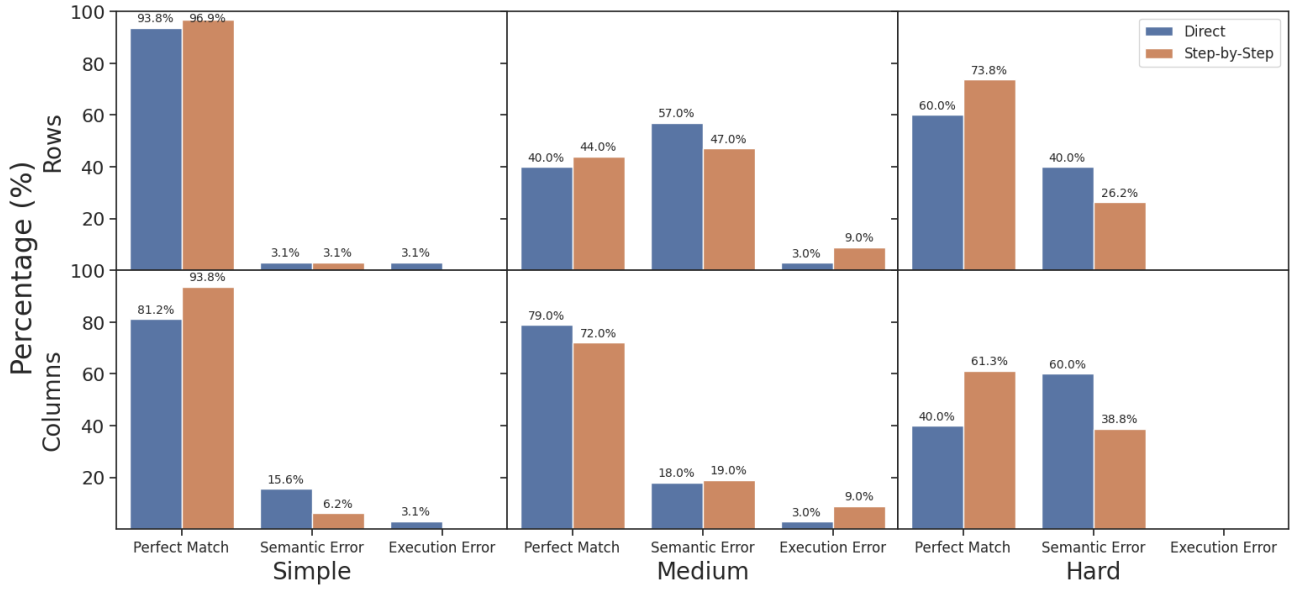


Fig. 10. Percentage of perfect queries, semantic errors, and execution errors for the direct and step-by-step query generation methods using Claude Opus 4.6.

Table 5. Few-shot PM results by query difficulty when using Gemini 2.5 Pro (direct).

Few-shot	Selection	Gemini 2.5 Pro					
		Simple		Medium		Hard	
		Rows	Columns	Rows	Columns	Rows	Columns
0-shot	–	0.86 ± 0.022	0.80 ± 0.016	0.42 ± 0.079	0.72 ± 0.063	0.36 ± 0.052	0.30 ± 0.000
1-shot	Random	0.86 ± 0.017 ∼	0.82 ± 0.026 ∼	0.50 ± 0.000 ↑	0.84 ± 0.055 ↑	0.34 ± 0.055 ∼	0.34 ± 0.114 ∼
	QTS	0.90 ± 0.015 ↑	0.85 ± 0.023 ↑	0.42 ± 0.042 ∼	0.79 ± 0.057 ↑	0.42 ± 0.063 ∼	0.27 ± 0.048 ∼
	MQS	0.94 ± 0.022 ↑	0.89 ± 0.028 ↑	0.38 ± 0.045 ∼	0.76 ± 0.055 ∼	**0.58 ± 0.045** ↑	0.36 ± 0.055 ∼
3-shot	Random	0.92 ± 0.017 ↑	0.87 ± 0.014 ↑	0.44 ± 0.055 ∼	0.80 ± 0.071 ↑	0.34 ± 0.055 ∼	0.38 ± 0.045 ↑
	QTS	**0.97 ± 0.000** ↑	**0.93 ± 0.017** ↑	0.58 ± 0.045 ↑	0.80 ± 0.000 ↑	0.40 ± 0.100 ∼	**0.40 ± 0.000** ↑
	MQS	0.94 ± 0.000 ↑	0.86 ± 0.017 ↑	**0.60 ± 0.000** ↑	**0.86 ± 0.055** ↑	0.48 ± 0.045 ↑	**0.40 ± 0.000** ↑

Notes. The best value per table column is bolded. Arrows compare the few-shot setting against the 0-shot baseline for each query difficulty using two-sided permutation tests on perfect-match scores grouped by run; ↑/↓ indicate significant higher/lower results at $\alpha = 0.05$ with no multiple-testing correction, and ∼ indicates not significant.

function calling can considerably expand the model's capabilities. In our NL-SQL dataset, the external knowledge associated with some queries, which is mainly composed of MJD dates and celestial coordinates (e.g., RA and Dec), was added by hand. However, this kind of information can be obtained automatically through function or tool calling. These tools leverage the model's ability to interpret the user's request and format the appropriate input for each function. Listing G.1 shows an example of a tool that converts dates found in the user request to MJD format. Table G.1 (see Appendix G) shows the PM results when using Claude Opus 4.6, Gemini 2.5 Pro, and Gemini 3 Flash w/ and w/o the function calling for MJD. It can be observed that the performance increases or remains the same for the sbs query generation method, except for hard rows with Claude Opus 4.6. The main point here is that the function calling simplifies the operation from the user viewpoint, and therefore it is highly recommended to include it in production.

Using SQL as a native structured output means the LLM generates executable database queries in a formally constrained syntax. This is structured output because SQL grammar is formal, tables and columns have schema constraints, and syntax validity is useful. For text-to-SQL tasks, Claude, Gemini, and GPT models all support structured output, but they implement it differently at the API and decoding levels. To obtain structured outputs, we used the integrated structured outputs feature of the API providers. OpenAI[14], Anthropic[15], and Google[16] have a specific variable named "text_format", "output_format", and "response_format", respectively, where a JSON schema structure can be passed. Another option is to define a Pydantic BaseModel structure, as shown in all the model guide pages, and selecting the "extra = '*forbid*'" configuration to comply with strict JSON schema requirements and matching. Some benefits of structured outputs include reliable type-safety, seamless system integration, reduced hallucinations, and simpler prompting to obtain consistent formatting. We tested a type of structured

[14] https://developers.openai.com/api/docs/guides/structured-outputs
[15] https://platform.claude.com/docs/en/build-with-claude/structured-outputs
[16] https://ai.google.dev/gemini-api/docs/structured-output

output for the decomposition stage of the step-by-step method for medium and hard queries, defined in Pydantic as shown in Listing G.2 (see Appendix G) and that can be transformed to a JSON schema. Listing G.3 shows the Pydantic structure transformed into a JSON dictionary schema that is the format given to the model, forcing the latter to return a list of steps to build the required SQL, and then joining all these steps into one large query in the SQL generation stage. We tested this format with the medium and hard queries. Table G.2 shows the performance with the normal output and the structured step-by-step output for Gemini 3 Flash, Gemini 2.5 Pro, and Claude Opus 4.6. The models did not show a significant performance improvement. Gemini 3 Flash obtained better results for columns for medium queries, while for rows in medium queries, a degraded performance is obtained for Gemini 3 Flash and Claude Opus 4.6. Nevertheless, some cases showed a more consistent output with more successful runs, reducing execution errors. This approach is worth to be explored in more detail in future work.

A natural extension of the current self-correction module would be to constrain its output via a typed schema, for example, a Pydantic-validated correction object carrying identifier-replacement records restricted to the database vocabulary, applied deterministically on the SQL abstract syntax tree (AST), with a free-form fallback for syntax, type, and structural errors. This would prevent hallucinated table and column names in the most common error class while preserving the flexibility required for user-requested aliases and complex query structures. This is a valuable direction for future work and we plan to explore it, particularly the post-generation validation approach, which is the most compatible with our current API-based pipeline.

6. Conclusions

In this work, we proposed a framework for performing the text-to-SQL parsing task in astronomy using in-context learning with LLMs. With this aim, we built a dataset of NL-SQL queries for the ALeRCE database. We show that the proposed framework outperforms the direct inference method and that self-correction, in general, improves results. In the performance comparison of 13 LLMs, we found that Claude Opus 4.6, Gemini 2.5 Pro, Gemini 3 Flash and GPT 5.2 Codex are the best LLMs for the task at hand. In future work, we plan to explore technologies beyond classic prompt engineering, such as retrieval-augmented generation (RAG) and fine-tuning LLMs on generating SQL outputs. Future work will evaluate the proposed framework against both specialized text-to-SQL systems, e.g., SQLCoder[17], and modern open-weight reasoning models such as DeepSeek, Qwen, and LLaMA variants. In addition, developing a more comprehensive self-review process that evaluates semantic correctness, not only execution errors, is a promising direction for future work. Our ultimate goal is to implement an online virtual assistant for ALeRCE users in the Rubin era.

Acknowledgements. This work gratefully acknowledges funding from ANID: Millennium Science Initiative, AIM23-0001 (P.A.E., J.E.-M., F.F., G.C.-V. A.M.M.A., G.P., F.E.B., M.C., R.D.); ANID/Basal (CATA) grant FB210003 (F.F., M.C., F.E.B.); and FONDECYT Regular grants 1220829 (P.A.E.), 1231877 (G.C.V), 1241005 (F.E.B.), and 1231637 (M.C.). This work was supported by Centro de Modelamiento Matemático (CMM) BASAL fund FB210005 from ANID-Chile. A.B. acknowledges support from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2094 – 390783311. We acknowledge Zhen Guo and Arti Joshi for their contribution in providing samples for the dataset used in this paper.

[17] https://github.com/defog-ai/sqlcoder

References

Alexander, K. D., Margutti, R., Gomez, S., et al. 2026, ApJ, 1000, 139
Arévalo, P., López-Navas, E., Martínez-Aldama, M., et al. 2024, A&A, 683, L8
Bellm, E. C., Kulkarni, S. R., Graham, M. J., et al. 2019, PASP, 131, 018002
Carrasco-Davis, R., Reyes, E., Valenzuela, C., et al. 2021, AJ, 162, 231
Chang, S., & Fosler-Lussier, E. 2023, in NeurIPS 2023 Second Table Representation Learning Workshop
Ciuca, I., Ting, Y.-S., Kruk, S., & Iyer, K. 2023, arXiv e-prints [arXiv:2306.11648]
Coughlin, M. W., Bloom, J. S., Nir, G., et al. 2023, ApJS, 267, 31
Dong, X., Zhang, C., Ge, Y., et al. 2023, arXiv e-prints [arXiv:2307.07306]
Förster, F., Cabrera-Vives, G., Castillo-Navarrete, E., et al. 2021, AJ, 161, 242
Gao, D., Wang, H., Li, Y., et al. 2024, Proc. VLDB Endow., 17, 1132
Ghosh, M., Santra, P., Iqbal, S. A., & Basuchowdhuri, P. 2022, in Proceedings of the first Workshop on Information Extraction from Scientific Publications, eds. T. Ghosal, S. Blanco-Cuaresma, A. Accomazzi, R. M. Patton, F. Grezes, & T. Allen (Association for Computational Linguistics), 100
Gomez, S., & Gezari, S. 2023, ApJ, 955, 46
Grezes, F., Blanco-Cuaresma, S., Accomazzi, A., et al. 2024, Astron. Data Anal. Softw. Syst. XXXI, 535, 119
Guo, C., Tian, Z., Tang, J., et al. 2023, in 20th Pacific Rim International Conference on Artificial Intelligence, PRICAI, Proceedings, Part II, 262
Ivezić, Ž., Kahn, S. M., Tyson, J. A., et al. 2019, ApJ, 873, 111
Iyer, K. G., Yunus, M., O'Neill, C., et al. 2024, ApJS, 275, 38
Joseph, S., Husain, S. M., Offner, S., et al. 2026, in The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track, 38
Katsogiannis-Meimarakis, G., & Koutrika, G. 2023, VLDB J., 32, 905
Khot, T., Trivedi, H., Finlayson, M., et al. 2023, in The Eleventh International Conference on Learning Representations
Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y. 2022, in Proceedings of the 36th International Conference on Neural Information Processing Systems, 35, 22199
Li, J., Hui, B., Qu, G., et al. 2023, Adv. Neural Inform. Process. Syst., 36
Li, J., Li, G., Li, Y., & Jin, Z. 2025, ACM Trans. Softw. Eng. Methodol., 34, 1
Liu, J., Shen, D., Zhang, Y., et al. 2022, in Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures, 100
Liu, H., Teng, Z., Cui, L., et al. 2023, in Findings of the Association for Computational Linguistics: EMNLP 2023, 2908
López-Navas, E., Martínez-Aldama, M., Bernal, S., et al. 2022, MNRAS, 513, L57
Magee, M., Sainz de Murieta, A., Collett, T., & Enzi, W. 2023, MNRAS, 525, 542
Müller-Bravo, T. E., Galbany, L., Stritzinger, M., et al. 2025, A&A, 702, A134
Nguyen, T. D., Ting, Y.-S., Ciuca, I., et al. 2023, in Proceedings of the Second Workshop on Information Extraction from Scientific Publications, 49
Oshikiri, K., Tanaka, M., Tominaga, N., et al. 2024, MNRAS, 527, 334
Perkowski, E., Pan, R., Nguyen, T. D., et al. 2024, RNAAS, 8, 7
Pomeroy, R. T., & Norris, M. A. 2024, MNRAS, 530, 3043
Pourreza, M., & Rafiei, D. 2023, in Proceedings of the 37th International Conference on Neural Information Processing Systems, 36, 36339
Qin, B., Hui, B., Wang, L., et al. 2022, arXiv e-prints [arXiv:2208.13629]
Sahoo, P., Singh, A. K., Saha, S., et al. 2024, arXiv e-prints [arXiv:2402.07927]
Sánchez-Sáez, P., Reyes, I., Valenzuela, C., et al. 2021, AJ, 161, 141
Shao, W., Zhang, R., Ji, P., et al. 2024, Res. Astron. Astrophys., 24, 065012
Sotnikov, V., & Chaikova, A. 2023, Galaxies, 11, 63
Wang, B., Ren, C., Yang, J., et al. 2025, in Proceedings of the 31st International Conference on Computational Linguistics, 540
Wei, J., Wang, X., Schuurmans, D., et al. 2022, in Advances in Neural Information Processing Systems, 35 (Curran Associates, Inc.), 24824
Ye, C., Yuan, S., Cooray, S., et al. 2025, arXiv e-prints [arXiv:2510.24591]
Yu, T., Zhang, R., Yang, K., et al. 2018, in Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, 3911
Zhang, Y., Deriu, J., Katsogiannis-Meimarakis, G., et al. 2023, Proc. VLDB Endow., 17, 685
Zhao, X., Li, M., Lu, W., et al. 2024, in Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024), 6144
Zhao, W. X., Zhou, K., Li, J., et al. 2026, Front. Comput. Sci., 20, 2012627
Zhou, D., Schärli, N., Hou, L., et al. 2023, in The Eleventh International Conference on Learning Representations

Appendix A: ALeRCE database

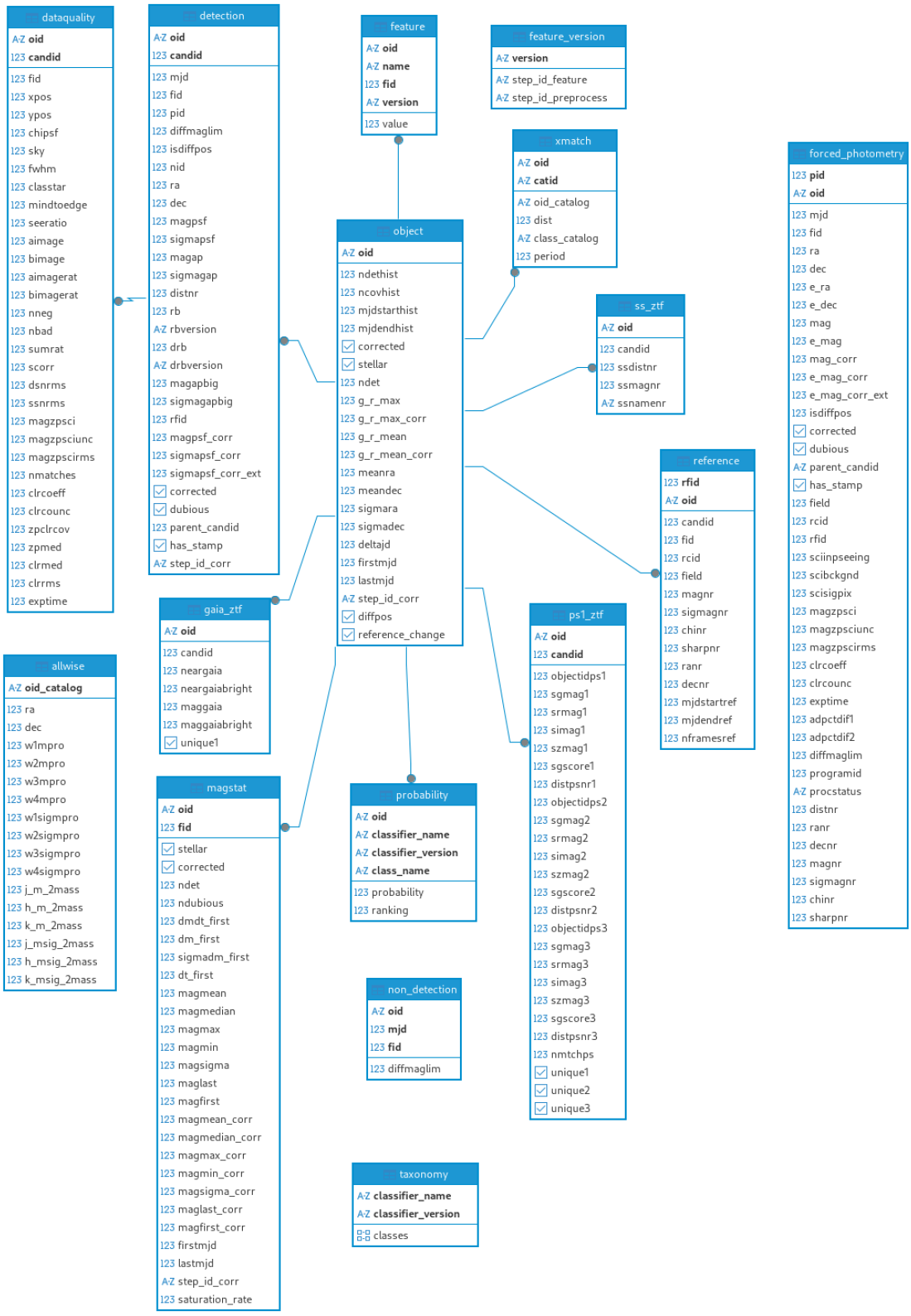


Fig. A.1: Entity-relationship diagram of the ALeRCE database.

Appendix B: Examples of queries

Table B.1: Example of a simple query.

Request	For objects with ZTF identifiers 'ZTF21aaobkmg', 'ZTF21aaomuka', find all rows in the 'probability' table and light curve classifier that have ranking 1 or 2. Return all columns from such table, sort by ranking
Gold SQL Query	`SELECT * FROM probability WHERE oid IN ('ZTF21aaobkmg','ZTF21aaomuka') AND classifier_name = 'lc_classifier' AND ranking <= 2 ORDER BY ranking`

Table B.2: Example of a medium query.

Request	For Solar System identifiers '2003FP134' and '2009UK56', get all detections for all ZTF objects that lie within 2 arcsec from any of them. Return the following columns, sort by MPC name and detection date: all columns from the 'ss_ztf' table; and detection date, filter identifier, isdiffpos flag, RA Dec coordinates, and difference magnitude (and its uncertainty)
Gold SQL Query	`SELECT ss_ztf.*, detection.mjd, detection.fid, detection.isdiffpos, detection.ra, detection.dec, detection.magpsf, detection.sigmapsf FROM ss_ztf INNER JOIN detection ON ss_ztf.oid = detection.oid AND ss_ztf.candid = detection.candid WHERE ssnamenr IN ('2003FP134','2009UK56') AND ssdistnr <2 ORDER BY ssnamenr, mjd`

Table B.3: Example of a hard query.

Request	Find all detections, non-detections and forced photometry points for the ZTF object 'ZTF24aamtvxb'. Return all epochs in the same output table, including the following columns: ZTF identifier, epoch date, filter identifier, isdiffpos flag, detection difference magnitude and its uncertainty, 5-sigma magnitude limit, forced difference magnitude and its uncertainty, and a column named 'table' (that contains the name of the table of origin for each epoch)
Gold SQL Query	`SELECT oid, mjd, fid, isdiffpos, magpsf, sigmapsf, NULL as diffmaglim, CAST(NULL AS bigint) as mag, CAST(NULL AS bigint) as e_mag, 'detection' as table FROM detection WHERE oid = 'ZTF24aamtvxb' UNION ALL SELECT oid, mjd, fid, NULL as isdiffpos, NULL as magpsf, NULL as sigmapsf, diffmaglim, CAST(NULL AS bigint) as mag, CAST(NULL AS bigint) as e_mag, 'non_detection' as table FROM non_detection WHERE oid = 'ZTF24aamtvxb' UNION ALL SELECT oid, mjd, fid, isdiffpos, NULL as magpsf, NULL as sigmapsf, NULL as diffmaglim, mag, e_mag, 'forced_photometry' as table FROM forced_photometry WHERE oid = 'ZTF24aamtvxb'`

Appendix C: Prompt formats

```
# Given the user request, select the tables needed to generate a SQL query
# The Database has the following tables:

# TABLE "object": contains basic filter and time-aggregated statistics such as location, number of
    observations, and the times of first and last detection.
...

# Give ONLY the TABLES that are needed to generate the SQL query.
# Give the answer in the following format: ['table1', 'table2', 'table3', ...]. For example, if the TABLES
    needed for the user request are TABLE object and TABLE taxonomy, then you should type: ['object', '
    taxonomy']
# Just give the tables and ignore any other task given in the request given as "request".
# Remember to use the exact name of the TABLES, as they are written in the DATABASE SCHEMA. Do NOT create
    table names.
# If you think that no table mentioned above is needed, then type: ""
# Request:
{User Request}
```

Listing C.1: Schema linking prompt.

```
# For the given request, classify it by difficulty as "simple", "medium", or "hard" based on the next
    description.

# Simple label description
# Medium label description
# Hard label description

# Database schema. It is not necessary to use all the tables in the schema, but you can use them if you need
    them.
{Table_Schema}
{Final instructions}
# Request:
{User Request}
```

Listing C.2: Difficulty classification prompt.

```
{self_correction_task}
# Correct a SQL query given the next user request:
{user_request}
# These are the table schemas. Assume that only the following tables are required for the query:
{table_schemas}
# The following query is not working due to a timeout error, correct the query using the correct database
    schema or nested queries to optimize.
# SQL Query
{sql_pred}
# Error returned when executing the query in the ALeRCE database
{sql_error}

# Follow these guidelines to correct the query:
# - Check if the SQL code includes the necessary conditions to optimize the query, and if the query is using
    the correct database schema or nested queries to optimize.
#     - It is possible that the query is too complex and that nested queries are necessary to optimize it.
#     - If there is a JOIN or a sub-query between some table and probability, check if the condition 'ranking
    =1' is set in the probability table, unless the request said otherwise.
# - If there are conditions involving dates or times, check if the dates are not too far away, or are in a
    reasonable range.
# - If the probability table is used, always use the next conditions, unless the user explicitly specifies
    different probability conditions.
# - Ensure there are at least 3 conditions on the probability table, because  if  not, the query is too
    general. Add more conditions if necessary.

# Check the query and correct it by modifying the SQL code where the error is found.
# Add COMMENTS IN PostgreSQL format so that the user can understand.
# Answer ONLY with the SQL query
# SQL:
```

Listing C.3: Example of timeout self-correction prompt.

Appendix D: Perfect match rates by LLM

Table D.1: Perfect match rates by LLM, query generation method, query difficulty, and with or without self-correction.

	Simple		Medium		Hard	
Model	w/o Self-Corr	w/ Self-Corr	w/o Self-Corr	w/ Self-Corr	w/o Self-Corr	w/ Self-Corr
ID Match (Rows) – Direct						
Claude 3.7	0.844 ± 0.0	0.866 ± 0.021	0.160 ± 0.084	0.160 ± 0.084	0.280 ± 0.079	0.380 ± 0.079
Claude Opus 4.6	0.844 ± 0.0	**0.938 ± 0.0**	0.250 ± 0.053	0.400 ± 0.047	**0.400 ± 0.0**	**0.480 ± 0.042**
Claude Sonnet 4.5	0.844 ± 0.0	0.875 ± 0.0	**0.400 ± 0.0**	**0.500 ± 0.0**	0.200 ± 0.0	0.200 ± 0.0
Gemini 2.5 Flash	0.775 ± 0.020	0.819 ± 0.025	**0.410 ± 0.057**	0.410 ± 0.057	0.240 ± 0.052	0.300 ± 5.9e-17
Gemini 2.5 Pro	0.809 ± 0.023	0.856 ± 0.022	**0.360 ± 0.117**	0.420 ± 0.079	0.160 ± 0.070	0.360 ± 0.052
Gemini 3 Flash	**0.866 ± 0.015**	**0.928 ± 0.015**	0.300 ± 5.9e-17	0.400 ± 0.0	**0.400 ± 0.0**	**0.500 ± 0.0**
Gemini 3.1 Flash	0.844 ± 0.0	0.906 ± 0.0	**0.400 ± 0.0**	0.400 ± 0.0	0.200 ± 0.0	0.300 ± 5.9e-17
GPT-4.1	0.753 ± 0.023	0.806 ± 0.029	0.210 ± 0.088	0.260 ± 0.070	0.270 ± 0.048	0.340 ± 0.052
GPT-4o	**0.856 ± 0.016**	0.891 ± 0.022	0.130 ± 0.048	0.260 ± 0.084	0.0 ± 0.0	0.110 ± 0.032
GPT-5	0.787 ± 0.032	0.822 ± 0.033	**0.350 ± 0.085**	0.380 ± 0.063	0.230 ± 0.048	0.290 ± 0.032
GPT-5.2	0.784 ± 0.023	0.800 ± 0.026	0.280 ± 0.063	0.300 ± 0.067	**0.370 ± 0.116**	0.380 ± 0.132
GPT-5.2-Codex	0.731 ± 0.030	0.816 ± 0.023	**0.320 ± 0.092**	0.340 ± 0.084	0.280 ± 0.063	0.310 ± 0.032
GPT-5.3-Codex	0.753 ± 0.018	0.775 ± 0.025	0.200 ± 0.067	0.210 ± 0.074	0.190 ± 0.032	0.330 ± 0.048
ID Match (Rows) – Step-by-Step						
Claude 3.7	0.794 ± 0.016	0.828 ± 0.030	0.230 ± 0.177	0.250 ± 0.165	0.150 ± 0.085	0.250 ± 0.085
Claude Opus 4.6	**0.938 ± 0.0**	**0.969 ± 0.0**	0.380 ± 0.042	**0.440 ± 0.070**	**0.530 ± 0.048**	**0.590 ± 0.032**
Claude Sonnet 4.5	0.906 ± 0.0	0.938 ± 0.0	0.400 ± 0.0	0.400 ± 0.0	0.200 ± 0.0	0.300 ± 5.9e-17
Gemini 2.5 Flash	0.853 ± 0.015	0.853 ± 0.015	0.200 ± 0.0	0.250 ± 0.053	0.190 ± 0.074	0.230 ± 0.048
Gemini 2.5 Pro	0.847 ± 0.023	0.875 ± 0.015	**0.490 ± 0.032**	**0.500 ± 0.0**	0.250 ± 0.053	0.370 ± 0.048
Gemini 3 Flash	**0.938 ± 0.0**	0.938 ± 0.0	0.390 ± 0.032	0.400 ± 0.0	0.300 ± 5.9e-17	0.400 ± 0.0
Gemini 3.1 Flash	0.812 ± 0.0	0.844 ± 0.0	0.300 ± 5.9e-17	0.300 ± 5.9e-17	0.100 ± 0.0	0.200 ± 0.0
GPT-4.1	0.797 ± 0.022	0.809 ± 0.034	0.270 ± 0.082	0.300 ± 0.115	0.210 ± 0.074	0.280 ± 0.079
GPT-4o	0.875 ± 0.033	0.900 ± 0.029	0.180 ± 0.042	0.270 ± 0.048	0.130 ± 0.067	0.220 ± 0.063
GPT-5	0.856 ± 0.030	0.884 ± 0.039	0.370 ± 0.048	0.390 ± 0.032	0.190 ± 0.032	0.260 ± 0.070
GPT-5.2	0.806 ± 0.029	0.809 ± 0.031	0.280 ± 0.042	0.290 ± 0.032	0.290 ± 0.074	0.290 ± 0.074
GPT-5.2-Codex	0.847 ± 0.045	0.869 ± 0.041	**0.420 ± 0.092**	**0.440 ± 0.107**	0.250 ± 0.053	0.280 ± 0.079
GPT-5.3-Codex	0.838 ± 0.013	0.875 ± 0.021	0.310 ± 0.074	0.320 ± 0.063	0.350 ± 0.053	0.360 ± 0.070
Column Match – Direct						
Claude 3.7	0.738 ± 0.030	0.762 ± 0.034	0.680 ± 0.042	0.790 ± 0.057	0.310 ± 0.088	0.280 ± 0.092
Claude Opus 4.6	0.797 ± 0.016	**0.812 ± 0.0**	0.640 ± 0.052	0.790 ± 0.032	0.220 ± 0.042	0.220 ± 0.042
Claude Sonnet 4.5	0.781 ± 0.0	0.781 ± 0.0	0.500 ± 0.0	0.700 ± 0.0	0.300 ± 5.9e-17	0.300 ± 5.9e-17
Gemini 2.5 Flash	0.775 ± 0.025	**0.800 ± 0.022**	0.760 ± 0.052	0.820 ± 0.079	0.240 ± 0.052	0.350 ± 0.071
Gemini 2.5 Pro	**0.828 ± 0.016**	**0.828 ± 0.016**	0.760 ± 0.097	0.820 ± 0.063	0.200 ± 0.067	0.280 ± 0.042
Gemini 3 Flash	0.803 ± 0.015	**0.812 ± 0.0**	0.800 ± 0.0	0.800 ± 0.0	**0.400 ± 0.0**	**0.500 ± 0.0**
Gemini 3.1 Flash	0.812 ± 0.0	**0.812 ± 0.0**	**0.900 ± 0.0**	**0.900 ± 0.0**	0.300 ± 5.9e-17	0.300 ± 5.9e-17
GPT-4.1	0.709 ± 0.015	0.741 ± 0.015	0.600 ± 0.047	0.650 ± 0.053	0.230 ± 0.048	0.230 ± 0.048
GPT-4o	0.728 ± 0.026	0.738 ± 0.016	0.270 ± 0.082	0.450 ± 0.085	0.100 ± 0.0	0.120 ± 0.042
GPT-5	0.750 ± 0.026	0.775 ± 0.025	0.690 ± 0.110	0.760 ± 0.107	0.250 ± 0.071	0.280 ± 0.063
GPT-5.2	0.762 ± 0.034	0.778 ± 0.031	0.750 ± 0.071	0.770 ± 0.048	0.270 ± 0.067	0.300 ± 0.067
GPT-5.2-Codex	0.728 ± 0.033	**0.809 ± 0.031**	0.760 ± 0.084	0.790 ± 0.088	0.280 ± 0.042	0.320 ± 0.042
GPT-5.3-Codex	0.759 ± 0.030	0.797 ± 0.022	0.560 ± 0.052	0.560 ± 0.052	0.190 ± 0.057	0.330 ± 0.048
Column Match – Step-by-Step						
Claude 3.7	0.734 ± 0.034	0.778 ± 0.027	0.530 ± 0.149	0.660 ± 0.143	**0.280 ± 0.103**	0.260 ± 0.052
Claude Opus 4.6	**0.938 ± 0.0**	**0.938 ± 0.0**	0.650 ± 0.053	0.720 ± 0.042	0.300 ± 5.9e-17	**0.310 ± 0.032**
Claude Sonnet 4.5	0.875 ± 0.0	0.875 ± 0.0	**0.800 ± 0.0**	**0.900 ± 0.0**	0.250 ± 0.053	0.300 ± 5.9e-17
Gemini 2.5 Flash	0.906 ± 0.0	0.906 ± 0.0	**0.850 ± 0.053**	**0.880 ± 0.042**	0.200 ± 0.0	0.240 ± 0.052
Gemini 2.5 Pro	0.906 ± 0.0	0.906 ± 0.0	**0.790 ± 0.032**	0.800 ± 0.047	0.240 ± 0.052	**0.310 ± 0.057**
Gemini 3 Flash	0.906 ± 0.0	0.906 ± 0.0	**0.800 ± 0.0**	0.810 ± 0.032	0.300 ± 5.9e-17	0.300 ± 5.9e-17
Gemini 3.1 Flash	0.875 ± 0.0	0.906 ± 0.0	**0.800 ± 0.0**	0.800 ± 0.0	0.300 ± 5.9e-17	0.300 ± 5.9e-17
GPT-4.1	0.903 ± 0.010	0.903 ± 0.010	0.680 ± 0.063	0.750 ± 0.053	0.250 ± 0.053	0.290 ± 0.032
GPT-4o	0.822 ± 0.021	0.822 ± 0.021	0.540 ± 0.097	0.700 ± 0.094	**0.360 ± 0.070**	**0.390 ± 0.074**
GPT-5	0.856 ± 0.037	0.872 ± 0.031	**0.750 ± 0.097**	0.830 ± 0.067	0.050 ± 0.053	0.180 ± 0.092
GPT-5.2	0.863 ± 0.016	0.863 ± 0.016	0.760 ± 0.052	0.770 ± 0.067	**0.390 ± 0.074**	**0.390 ± 0.074**
GPT-5.2-Codex	0.887 ± 0.037	0.900 ± 0.029	**0.810 ± 0.057**	0.830 ± 0.048	**0.330 ± 0.067**	**0.380 ± 0.042**
GPT-5.3-Codex	0.875 ± 0.0	0.906 ± 0.0	0.690 ± 0.057	0.730 ± 0.067	**0.370 ± 0.067**	**0.380 ± 0.079**

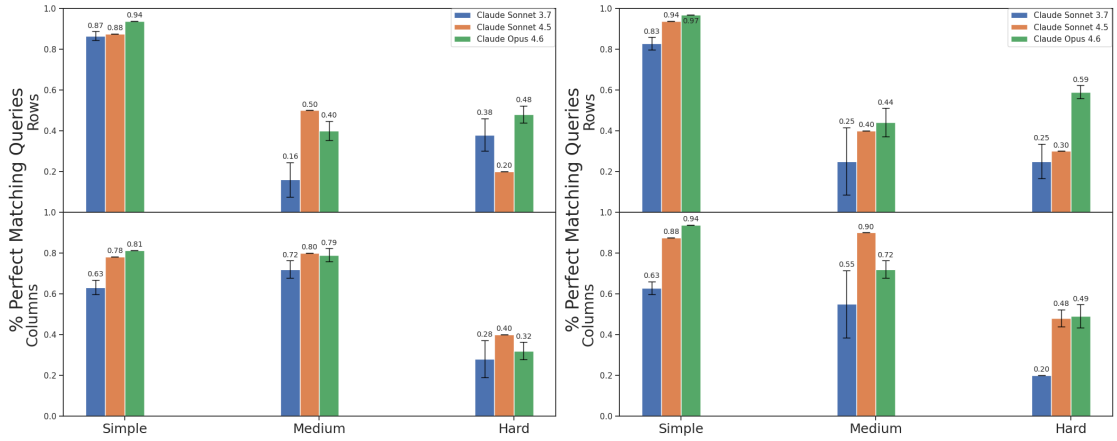
Notes. In each of the four sections of the table, the best mean per column is shown in boldface type, together with every LLM model whose run-level scores are not significantly worse than the best under a two-sided unpaired permutation test (Holm-corrected, $\alpha = 0.05$).

Appendix E: Additional results

Table E.1: Total and stagewise maximum per-query token counts by LLM model and query generation method.

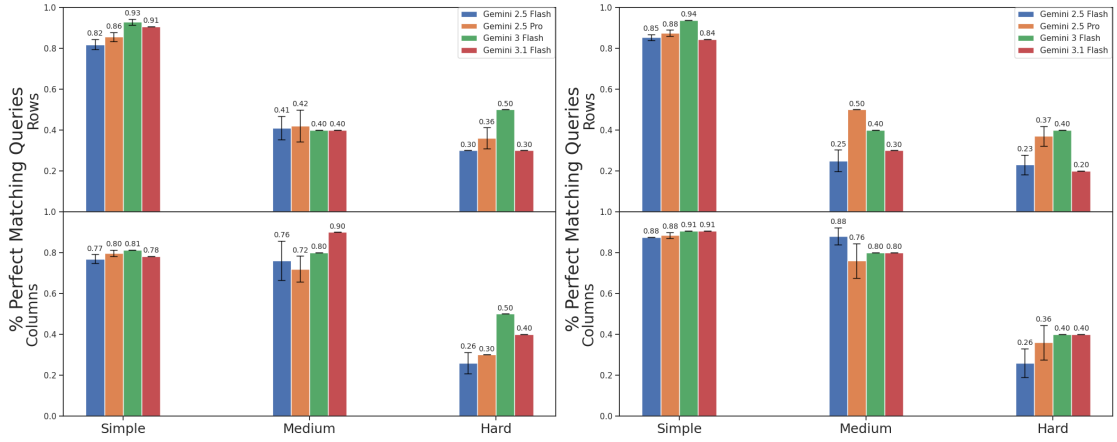
Model	Direct						Step-by-Step					
	Total	Schema	Difficulty	Plan	SQL	Self-Correction	Total	Schema	Difficulty	Plan	SQL	Self-Correction
GPT-5.2	**42 671**	802	**0**	**0**	**15 363**	**28 877**	19 357	802	4 328	8 554	7 991	5 303
GPT-5.3-Codex	39 907	801	**0**	**0**	12 261	27 044	22 978	801	4 263	7 039	6 574	6 712
GPT-5.2-Codex	22 106	1 539	**0**	**0**	13 740	9 029	35 236	1 539	5 085	12 228	10 697	9 252
GPT-5	26 655	3 919	**0**	**0**	13 312	11 232	**58 071**	3 919	**6 609**	**20 935**	**16 627**	**17 157**
Claude Opus 4.6	8 193	1 122	**0**	**0**	4 716	4 135	13 439	1 122	3 648	5 524	4 415	8 098
GPT-4.1	20 738	795	**0**	**0**	8 659	11 351	20 279	951	4 201	7 557	6 878	5 160
Claude Sonnet 3.7	11 303	2 091	**0**	**0**	1 211	8 138	8 243	2 091	1 190	1 162	1 153	4 543
Gemini 2.5 Pro	20 741	**7 936**	**0**	**0**	6 826	6 579	34 344	**7 936**	4 606	8 128	7 499	6 795
GPT-4o	9 509	795	**0**	**0**	6 632	4 127	18 677	795	4 809	7 622	4 647	5 010
Claude Sonnet 4.5	5 737	1 729	**0**	**0**	947	3 692	12 701	1 729	1 194	2 088	1 101	7 872
Gemini 3 Flash	8 557	1 924	**0**	**0**	6 084	3 437	18 789	1 924	4 486	7 458	6 696	6 595
Gemini 3.1 Flash	8 206	913	**0**	**0**	7 067	3 397	27 134	913	5 144	7 835	7 982	5 349

Notes. Values represent the highest token count observed across all evaluated queries for each stage. The largest value per column is bolded to indicate the upper bound.



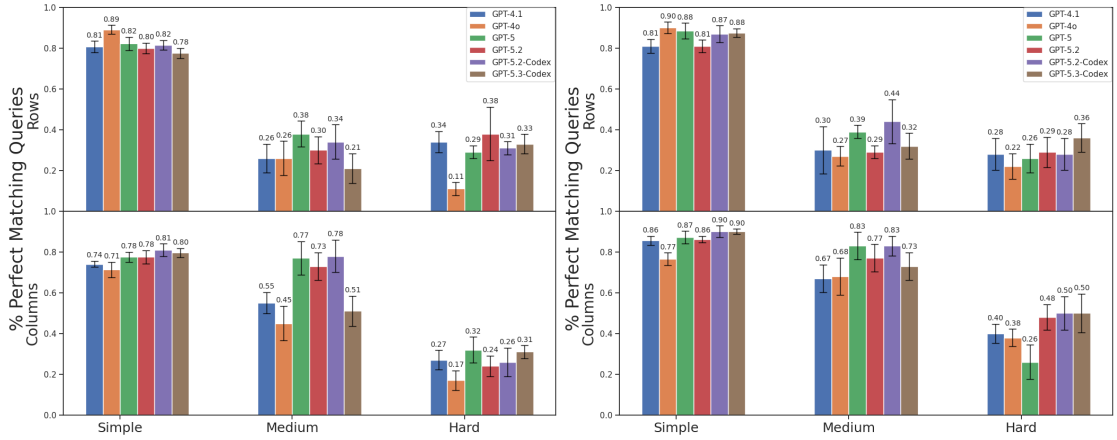
(a) Direct generation method (b) Step-by-step generation method

Fig. E.1: Performance comparison of Claude models, as a function of the level of query difficulty.



(a) Direct generation method (b) Step-by-step generation method

Fig. E.2: Performance comparison of Gemini models, as a function of the level of query difficulty.



(a) Direct generation method

(b) Step-by-step generation method

Fig. E.3: Performance comparison of GPT models, as a function of the level of query difficulty.

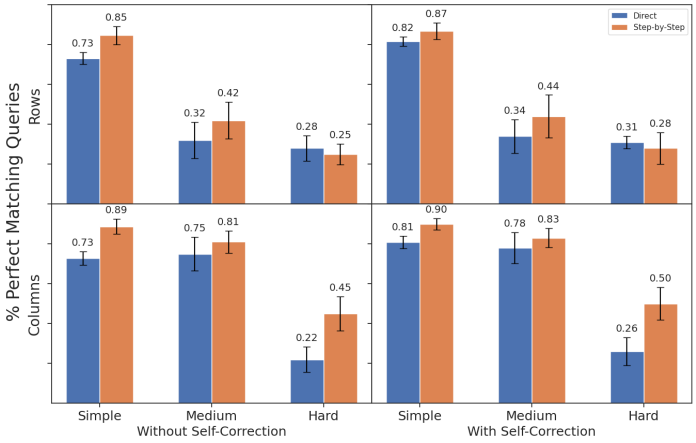


Fig. E.4: GPT-5.2-Codex performance without versus with self correction for direct and step-by-step generation methods.

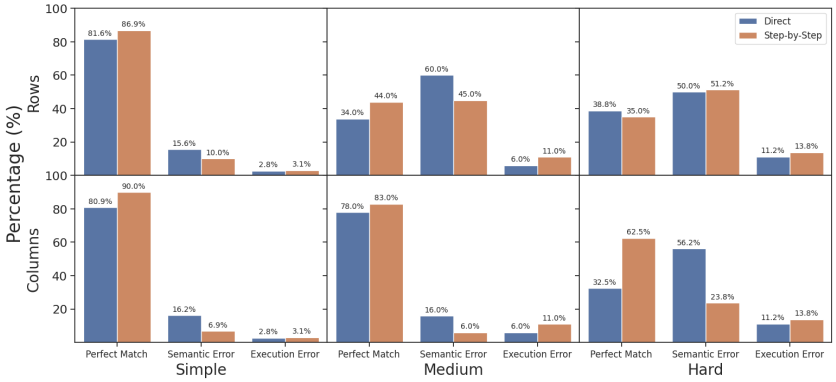


Fig. E.5: Percentage of perfect queries, semantic errors, and execution errors for the direct and step-by-step query generation methods using GPT-5.2-Codex.

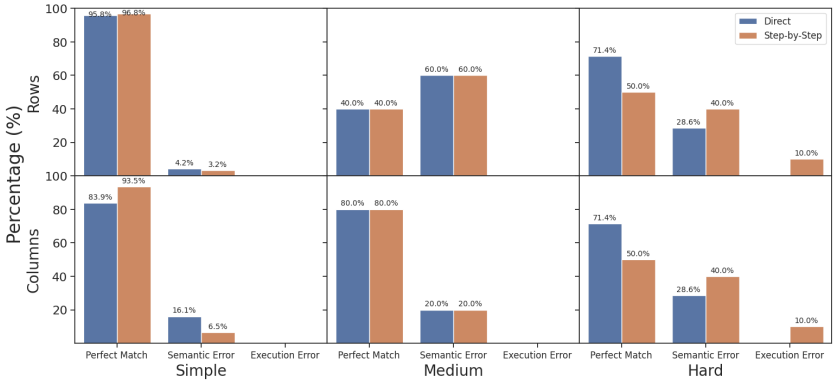


Fig. E.6: Number of perfect queries, semantic errors, and execution errors for the direct and step-by-step query generation methods using Gemini 3 Flash.

Appendix F: Few-shot prompting

Table F.1: Few-shot PM results by query difficulty when using Gemini 2.5 Pro (step-by-step).

Few-shot	Selection	Gemini 2.5 Pro					
		Simple		Medium		Hard	
		Rows	Columns	Rows	Columns	Rows	Columns
0-shot	-	0.88 ± 0.015	0.88 ± 0.015	0.50 ± 0.000	0.76 ± 0.084	0.37 ± 0.048	0.36 ± 0.084
1-shot	Random	**0.97 ± 0.000** ↑	0.92 ± 0.017 ~	0.50 ± 0.000 ~	0.82 ± 0.084 ~	0.32 ± 0.045 ↓	0.40 ± 0.122 ~
	QTS	0.96 ± 0.017 ↑	0.94 ± 0.014 ↑	0.50 ± 0.000 ~	0.74 ± 0.055 ~	0.36 ± 0.114 ~	0.38 ± 0.084 ~
	MQS	0.95 ± 0.017 ↑	0.93 ± 0.017 ↑	0.50 ± 0.000 ~	0.82 ± 0.045 ~	0.34 ± 0.055 ~	**0.46 ± 0.055** ~
3-shot	Random	0.95 ± 0.028 ↑	0.94 ± 0.000 ↑	**0.60 ± 0.000** ↑	0.82 ± 0.045 ~	0.40 ± 0.122 ~	0.42 ± 0.045 ~
	QTS	0.95 ± 0.017 ↑	**0.95 ± 0.017** ↑	0.50 ± 0.000 ~	**0.84 ± 0.055** ~	**0.48 ± 0.045** ↑	**0.46 ± 0.055** ↑
	MQS	0.94 ± 0.014 ↑	0.94 ± 0.000 ↑	0.50 ± 0.000 ~	0.82 ± 0.045 ~	0.32 ± 0.045 ↓	0.36 ± 0.055 ~

Notes. The best value per table column is bolded. Arrows compare the few-shot setting against the 0-shot baseline for each query difficulty using two-sided permutation tests on perfect-match scores grouped by run; ↑/↓ indicate significantly higher/lower results at $\alpha = 0.05$ with no multiple-testing correction, and ~ indicates not significant.

Table F.2: Few-shot PM results by query difficulty when using Claude Opus 4.6 (step-by-step).

Few-shot	Selection	Claude Opus 4.6					
		Simple		Medium		Hard	
		Rows	Columns	Rows	Columns	Rows	Columns
0-shot	-	0.97 ± 0.000	0.94 ± 0.000	**0.44 ± 0.070**	0.72 ± 0.042	**0.59 ± 0.032**	0.49 ± 0.057
1-shot	Random	0.97 ± 0.000 ~	0.93 ± 0.017 ~	0.32 ± 0.045 ↓	0.72 ± 0.045 ~	0.42 ± 0.045 ↓	0.46 ± 0.055 ~
	QTS	0.97 ± 0.000 ~	0.94 ± 0.000 ~	0.30 ± 0.000 ↓	0.70 ± 0.000 ~	0.48 ± 0.045 ↓	**0.50 ± 0.000** ~
	MQS	0.97 ± 0.000 ~	0.96 ± 0.017 ~	0.42 ± 0.045 ~	**0.82 ± 0.045** ↑	0.52 ± 0.045 ↓	0.48 ± 0.045 ~
3-shot	Random	0.90 ± 0.014 ↓	0.88 ± 0.014 ↓	0.38 ± 0.084 ~	**0.82 ± 0.045** ↑	0.38 ± 0.045 ↓	0.42 ± 0.084 ~
	QTS	**0.97 ± 0.014** ~	**0.97 ± 0.000** ↑	0.40 ± 0.000 ~	0.70 ± 0.000 ~	0.48 ± 0.045 ↓	0.48 ± 0.045 ~
	MQS	0.94 ± 0.000 ↓	0.94 ± 0.000 ~	0.30 ± 0.000 ↓	0.70 ± 0.000 ~	0.58 ± 0.045 ~	0.46 ± 0.055 ~

Notes. The best value per table column is bolded. Arrows compare the few-shot setting against the 0-shot baseline for each query difficulty using two-sided permutation tests on perfect-match scores grouped by run; ↑/↓ indicate significantly higher/lower results at $\alpha = 0.05$ with no multiple-testing correction, and ~ indicates not significant.

````
## Example 1 (tables: dataquality, ...)
# Important Information for the query
External Knowledge:
# User Request: ...
```sql
...
```
## End Example

# Important Information for the query
External Knowledge:
# User Request:
````

Listing F.1: Example few-shot prompt

Appendix G: Function calling and structured outputs

Table G.1: Perfect match rates by LLM, query generation method, query difficulty, and with or without function calling (without self-correction).

	Simple		Medium		Hard	
Model	w/o Func. call	w/ Func. call	w/o Func. call	w/ Func. call	w/o Func. call	w/ Func. call
ID Match (Rows) – Direct						
Claude Opus 4.6	0.844 ± 0.000	**0.850 ± 0.014**	0.250 ± 0.053	0.280 ± 0.045	**0.400 ± 0.000**	**0.460 ± 0.055**
Gemini 2.5 Pro	0.809 ± 0.023	0.787 ± 0.014	**0.360 ± 0.117**	**0.320 ± 0.045**	0.160 ± 0.070	0.200 ± 0.071
Gemini 3 Flash	**0.866 ± 0.015**	0.769 ± 0.034	0.300 ± 0.000	**0.320 ± 0.042**	0.300 ± 0.000	0.360 ± 0.052
ID Match (Rows) – Step-by-Step						
Claude Opus 4.6	0.938 ± 0.000	0.938 ± 0.000	0.380 ± 0.042	0.390 ± 0.088	**0.530 ± 0.048**	**0.460 ± 0.084**
Gemini 2.5 Pro	0.847 ± 0.023	0.863 ± 0.017	**0.490 ± 0.032**	**0.500 ± 0.000**	0.250 ± 0.053	0.240 ± 0.055
Gemini 3 Flash	**0.969 ± 0.000**	**0.944 ± 0.025**	0.290 ± 0.032	0.310 ± 0.088	0.300 ± 0.000	0.240 ± 0.052
Column Match – Direct						
Claude Opus 4.6	0.797 ± 0.016	0.787 ± 0.014	0.640 ± 0.052	0.600 ± 0.071	0.320 ± 0.042	0.300 ± 0.000
Gemini 2.5 Pro	0.797 ± 0.016	**0.831 ± 0.017**	0.660 ± 0.084	0.560 ± 0.089	0.180 ± 0.079	0.220 ± 0.045
Gemini 3 Flash	**0.803 ± 0.015**	0.806 ± 0.020	**0.800 ± 0.000**	**0.710 ± 0.057**	**0.400 ± 0.000**	**0.420 ± 0.042**
Column Match – Step-by-Step						
Claude Opus 4.6	**0.938 ± 0.000**	**0.938 ± 0.000**	0.650 ± 0.053	0.660 ± 0.126	**0.480 ± 0.042**	**0.460 ± 0.052**
Gemini 2.5 Pro	0.884 ± 0.015	0.869 ± 0.026	0.750 ± 0.071	0.780 ± 0.084	0.290 ± 0.057	0.260 ± 0.055
Gemini 3 Flash	**0.938 ± 0.000**	**0.938 ± 0.000**	**0.800 ± 0.000**	**0.830 ± 0.095**	0.400 ± 0.000	0.420 ± 0.042

Notes. Best results per column in each of the four sections of the table are bolded.

Table G.2: Structured versus base step-by-step plan comparison, by difficulty (with self-correction).

	Medium		Hard	
Model	Free-form	Structured outputs	Free-form	Structured outputs
Rows				
Gemini 3 Flash	0.40 ± 0.000	0.30 ± 0.100 ↓	0.40 ± 0.000	0.38 ± 0.045 ~
Gemini 2.5 Pro	**0.50 ± 0.000**	**0.50 ± 0.000**	0.37 ± 0.048	0.34 ± 0.055 ~
Claude Opus 4.6	0.44 ± 0.070	0.30 ± 0.000 ↓	**0.59 ± 0.032**	**0.58 ± 0.130** ~
Columns				
Gemini 3 Flash	**0.81 ± 0.032**	**0.88 ± 0.045** ↑	0.30 ± 0.000	**0.30 ± 0.000**
Gemini 2.5 Pro	0.80 ± 0.047	0.84 ± 0.055 ~	**0.31 ± 0.057**	0.28 ± 0.045 ~
Claude Opus 4.6	0.72 ± 0.042	0.70 ± 0.000 ~	**0.31 ± 0.032**	0.28 ± 0.045 ~

Notes. Each pair shows rows and columns as mean ± std over run-level means; the best value per column is bolded. Markers on structured cells reflect a unpaired permutation test, ↑/↓ when the adjusted p-value rejects equality, ~ otherwise.

```
date_to_mjd_tool = types.FunctionDeclaration(
            name="date_to_mjd",
            description=(
                "Convert ONE date expression to a single MJD value. Accepts ISO 8601 (e.g. '2022-12-01'), or relative
    expressions anchored to the current time (e.g. '300 days ago', '7 days ago'). "
                "For date ranges, call this tool once per boundary. "
                "ALeRCE stores all dates as MJD."
            ),
            parameters=types.Schema(
                type=types.Type.OBJECT,
                properties={
                    "date_expression": types.Schema(
                        type=types.Type.STRING,
                        description="A single date: ISO 8601 ('2023-09-01') or relative ('300 days ago', '7 days ago')."
                    ),
                    "label": types.Schema(
                        type=types.Type.STRING,
                        description="A short snake_case key summarizing the date as mjd_[month]_[day] or mjd_[relative]_[
    count], e.g. 'mjd_august_23' or 'mjd_hours_1'."
                    ),
                },
                required="date_expression",
            ),
        )
```

Listing G.1: Specialized tool defined in Python to transform date formats to MJD.

```
class PlanStep(BaseModel):
    step: str = Field(
        description="A single decomposition step describing part of the plan to generate the SQL query.")
class DecompositionPlanOutput(BaseModel):
    steps: listPlanStep = Field(
        description="Ordered decomposition steps for the SQL generation plan.",
        min_length=1,)
```

Listing G.2: Pydantic model for the structured output step-by-step planning.

```
{"type": "json_schema",
    "name": "DecompositionPlanOutput",
    "strict": true,
    "schema": {
      "$defs": {
        "PlanStep": {
          "additionalProperties": false,
          "properties": {
            "step": {
              "description": "A single decomposition step describing part of the plan to generate the SQL query.",
              "title": "Step",
              "type": "string"
            }
          },
          "required":
            "step"
          ,
          "title": "PlanStep",
          "type": "object"
        }
      },
      "additionalProperties": false,
      "properties": {
        "steps": {
          "description": "Ordered decomposition steps for the SQL generation plan.",
          "items": {
            "$ref": "#/$defs/PlanStep"
          },
          "minItems": 1,
          "title": "Steps",
          "type": "array"
        }
      },
      "required":
        "steps"
      ,
      "title": "DecompositionPlanOutput",
      "type": "object"
    }
 }
```

Listing G.3: Pydantic format converted to JSON schema for API calls.