Cerebra: Aligning Implicit Knowledge in Interactive SQL Authoring

Yunfan Zhou
State Key Lab of CAD&CG
Zhejiang University
Hangzhou, Zhejiang, China
yf.zhou@zju.edu.cn

Qiming Shi
State Key Lab of CAD&CG
Zhejiang University
Hangzhou, Zhejiang, China
qimingshi@zju.edu.cn

Zhongsu Luo
State Key Lab of CAD&CG
Zhejiang University
Hangzhou, Zhejiang, China
zhongsuluo@zju.edu.cn

Xiwen Cai
Department of Digital Intelligence
China Mobile
Shenzhen, Guangdong, China
caixiwen@chinamobile.com

Yanwei Huang
HKUST
Hong Kong S.A.R., China
yanwei.huang@connect.ust.hk

Dae Hyun Kim
Department of Computer Science and Engineering, Yonsei University
Seoul, Republic of Korea
Graduate School of Artificial Intelligence, POSTECH
Pohang, Republic of Korea
dhkim16@yonsei.ac.kr

Di Weng*
School of Software Technology
Zhejiang University
Ningbo, Zhejiang, China
dweng@zju.edu.cn

Yingcai Wu
State Key Lab of CAD&CG
Zhejiang University
Hangzhou, Zhejiang, China
ycwu@zju.edu.cn

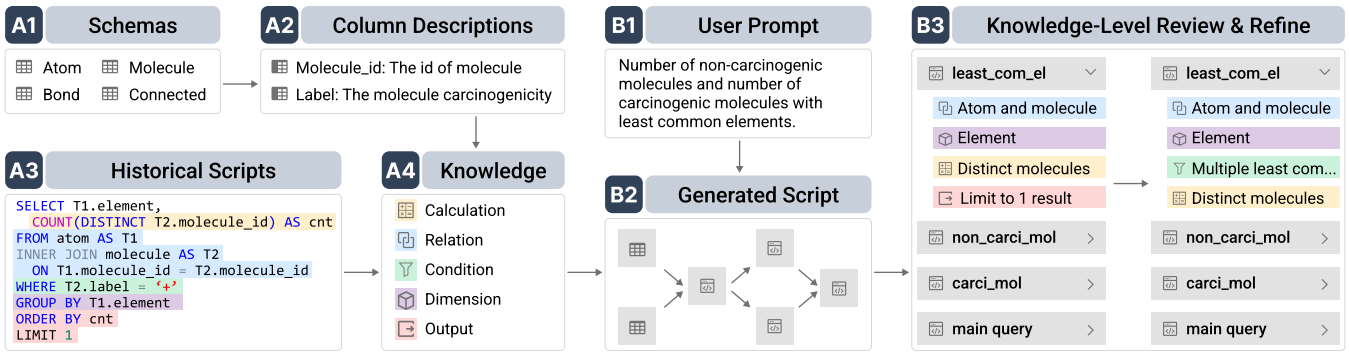

Figure 1: The workflow of *Cerebra* consists of two stages: offline (A1-A4) and online (B1-B3). In the offline stage, *Cerebra* first generates column descriptions (A2) based on the database schemas (A1). It then extracts five types of implicit knowledge (A4) in natural language from user-authored historical scripts (A3). In the online stage, when a user submits a natural language instruction (B1), *Cerebra* retrieves relevant knowledge items and generates the corresponding SQL script (B2). The generated script is then parsed and presented in the Knowledge View (B3), where users can review and refine the knowledge items to iteratively improve the script.

Abstract

LLM-driven tools have significantly lowered barriers to writing SQL queries. However, user instructions are often underspecified, assuming the model understands implicit knowledge, such as dataset schemas, domain conventions, and task-specific requirements, that isn't explicitly provided. This results in frequently erroneous scripts that require users to repeatedly clarify their intent. Additionally, users struggle to validate generated scripts because they cannot verify whether the model correctly applied implicit knowledge.

*Di Weng is the corresponding author.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CHI '26, Barcelona, Spain

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2278-3/2026/04
https://doi.org/10.1145/3772318.3790974

We present *Cerebra*, an interactive NL-to-SQL tool that aligns implicit knowledge between users and LLMs during SQL authoring. *Cerebra* automatically retrieves implicit knowledge from historical SQL scripts based on user instructions, presents this knowledge in an interactive tree view for code review, and supports iterative refinement to improve generated scripts. To evaluate the effectiveness and usability of *Cerebra*, we conducted a user study with 16 participants, demonstrating its improved support for customized SQL authoring. The source code of *Cerebra* is available at https://github.com/zjuidg/CHI26-Cerebra.

CCS Concepts

• **Human-centered computing → User interface programming**.

Keywords

Interactive SQL authoring, LLM-driven code assistance

ACM Reference Format:

Yunfan Zhou, Qiming Shi, Zhongsu Luo, Xiwen Cai, Yanwei Huang, Dae Hyun Kim, Di Weng, and Yingcai Wu. 2026. Cerebra: Aligning Implicit Knowledge in Interactive SQL Authoring. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems (CHI '26), April 13–17, 2026, Barcelona, Spain.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3772318.3790974

1 Introduction

With the rapid growth of available data in various fields, being able to explore data effectively with SQL queries has become crucial for data practitioners. Even for those who are proficient in SQL, authoring code that clearly expresses their intents can be difficult and time-consuming [40], while many other practitioners have only limited programming experience [76]. As a result, Natural Language to SQL (NL-to-SQL) techniques, especially LLM-driven ones [8, 45, 62] have been widely adopted to lower the barriers to query construction [41]. Recent advances in NL-to-SQL techniques have led to substantial improvements in basic semantic understanding and translation [31, 51], with state-of-the-art models achieving over 90% accuracy on the Spider benchmark [93] for well-formed natural language instructions and database schemas.

Despite recent advances, users continue to struggle with obtaining accurate results from NL-to-SQL systems when their natural language queries involve *implicit knowledge*. This refers to the assumptions, conventions, and contexts that users do not articulate explicitly but are crucial to accurate query generation. Such knowledge encompasses dataset-specific conventions and task-specific computations, creating a knowledge alignment gap that hinders accurate query interpretation and SQL generation. The BIRD toxicological database [47] (Figure 2) exemplifies this issue. A query like "*List all non-bonding elements*" assumes implicit knowledge that non-bonding elements are atoms in the "Atom" table with no corresponding entries in the "Connected" table. Typical NL-to-SQL models, lacking this understanding, may fail to generate the correct SQL query. Another example involves queries like "*What is the difference between the number of carcinogenic molecules and the number of non-carcinogenic molecules*". Such queries also assume implicit dataset-specific knowledge that carcinogenicity status is represented by labels such as "+" and "-" in a specific field. As a result, users frequently need to repeatedly clarify implicit knowledge such as the meaning of "non-bonding elements" and "carcinogenic" status, leading to frustration from iterative query refinement and numerous unexpected failures.

To address this gap, we conducted a preliminary study to investigate the role of implicit knowledge in the SQL authoring process. Through interviews with 10 data practitioners who regularly use LLM-driven NL-to-SQL tools, we found that users consistently supplement their natural language queries with extensive contextual information, including database schemas and business backgrounds, to bridge implicit knowledge gaps. Many participants (6/10) also reused existing SQL scripts in their prompts, effectively transferring the implicit domain logic and computational patterns embedded within proven solutions. However, users face a fundamental tension: concise queries fail because they omit essential implicit assumptions, while making all implicit knowledge explicit through detailed prompts becomes excessively burdensome and error-prone. Furthermore, users struggle to interpret the implicit assumptions that LLMs make when generating code, finding it difficult to align high-level requirements with low-level code fragments and locate errors when their implicit expectations are not met. While prior research has explored interactive interfaces for SQL construction, understanding, and modification [11, 16, 68, 76], the challenge of surfacing and aligning implicit knowledge in users' natural language instructions remains largely unaddressed. These findings highlight the need for NL-to-SQL tools that can better surface and align implicit knowledge between users and models.

In response to these challenges, we present *Cerebra*, an interactive NL-to-SQL tool designed to surface and align implicit knowledge between users and LLMs during SQL authoring. Drawing directly from our preliminary study findings, particularly participants' practice of reusing existing scripts to transfer implicit knowledge, *Cerebra* systematically captures and leverages the implicit knowledge embedded in users' historical SQL scripts. We first introduce a framework that categorizes implicit knowledge in NL-to-SQL processes into five types: calculation, condition, relation, dimension, and output. *Cerebra* extracts natural language descriptions and associated code fragments from users' past queries based on these categories, creating a repository of implicit knowledge patterns. During query authoring, *Cerebra* retrieves relevant implicit knowledge from this repository to enhance SQL generation, addressing the core challenge of conveying unstated assumptions from users to LLMs. To tackle the interpretation challenge identified in our study, *Cerebra* features a knowledge tree view that visualizes how the system inferred and applied implicit knowledge in generating SQL scripts. This transparency allows users to understand the model's assumptions and iteratively refine both the implicit knowledge and the generated SQL when results don't meet their expectations.

To evaluate the usability and effectiveness of *Cerebra*, we conducted a counterbalanced mixed-design user study on 16 participants. Participants of the user study completed the SQL authoring tasks within a significantly shorter period of time with *Cerebra* than a baseline tool without implicit knowledge integration. User feedback further confirms that *Cerebra* helps users convey their

A Atom

atom_id	molecule_id	element
TR000_1	TR000	cl
TR000_2	TR000	c
TR000_3	TR000	cl
TR000_4	TR000	cl
TR000_5	TR000	h

atom_id: Atom unique identifier

molecule_id: Identifier for the molecule to which the atom belongs

element: Chemical element symbol of the atom, such as "cl" for chlorine, "c" for carbon, "h" for hydrogen

B Molecule

molecule_id	label
TR000	+
TR001	+
TR002	-
TR004	-
TR006	+

molecule_id: Molecule unique identifier

label: Indicates carcinogenicity classification of the molecule. "+" indicates carcinogenic, "-" indicates non-carcinogenic

C Bond

bond_id	molecule_id	bond_type
TR000_1_2	TR000	-
TR000_2_3	TR000	-
TR000_2_4	TR000	-
TR000_2_5	TR000	-
TR001_10_11	TR001	=

bond_id: Unique identifier for a bond between two atoms in a molecule

molecule_id: Identifier for the molecule

bond_type: Chemical bond type. "-", "=", "#" represent single, double and triple bond, respectively

D Connected

atom_id	atom_id2	bond_id
TR000_1	TR000_2	TR000_1_2
TR000_2	TR000_1	TR000_1_2
TR000_2	TR000_3	TR000_2_3
TR000_3	TR000_2	TR000_2_3
TR000_2	TR000_4	TR000_2_4

atom_id: Identifier for an atom in a molecule

atom_id2: Identifier for a second atom connected to the first atom

bond_id: Identifier for a bond between the two atoms

Figure 2: The schema, sample data and column descriptions for the Toxicology database in a NL-to-SQL dataset, BIRD. The database consists of four tables: A) Atom, which records the atoms present in each molecule. B) Molecule, which stores the properties of each molecule, including its carcinogenicity label. C) Bond, which describes the chemical bonds between atoms within a molecule. D) Connected, which represents the connectivity relationships between pairs of atoms via specific bonds.

implicit knowledge in natural language more easily and better understand the implicit knowledge inferred by the model. The key contributions of this paper are three-fold:

- A preliminary study with 10 data practitioners that reveals the central role of implicit knowledge in NL-to-SQL tasks, identifying key challenges in implicit knowledge transfer between humans and LLMs.
- *Cerebra*, an interactive NL-to-SQL tool that captures implicit knowledge from historical scripts, decomposes user requests into subqueries, and provides transparent knowledge visualization to improve human-LLM alignment in SQL authoring.
- A comparative evaluation demonstrating the usability and effectiveness of *Cerebra* in supporting communication of implicit knowledge in NL-to-SQL tasks.

2 Related Work

In this section, we review previous research on interactive SQL authoring, NL-driven data querying, and human-AI collaboration.

2.1 Interactive SQL Authoring Tools

Authoring SQL queries is a common practice in database development. Traditional integrated development environments (IDEs) like Navicat [63] and DataGrip [30] provide basic SQL authoring functionalities such as code completion and code formatting. Since these IDEs require significant expertise to use effectively, several *visual programming* tools [1, 2, 5, 24, 58] are designed to lower the barrier by providing tree or graph representation of the code. For instance, Tioga-2 [2] incorporates a data flow diagram to represent SQL queries and users can directly manipulate nodes and links to edit the queries. However, these visual programming tools often rely on predefined templates, making it demanding to configure numerous options to construct queries.

To better support SQL authoring requirements, many SQL authoring tools integrate program synthesis techniques like *programming by example* [43, 82, 94] or *NL-to-SQL* [15, 45, 62], where users can declaratively specify their query requirements. Specifically, there is a substantial amount of research focusing on NL-to-SQL models, adopting rule-based [42, 66, 85], neural network-based [19, 81, 92], pre-trained language model-based [44, 46, 67] and LLM-based [15, 45, 62] methods. See [51] for a comprehensive survey of these models. The advancements in NL-to-SQL techniques enable SQL authoring tools to understand more complex natural language and table structures in sophisticated databases, thereby providing users with smarter code suggestions. However, these NL-to-SQL models cannot always provide accurate results that satisfy user needs, requiring considerable effort in validating the generated queries and correcting the mistakes. To bridge the gap, numerous interactive tools are proposed, many of which do not integrate LLMs, but instead focus on facilitating two key developer tasks: *code understanding* and *query refinement*.

Code understanding tools aim to demystify the logic of SQL queries by employing four primary strategies. First, to abstract away complex syntax, several tools focus on visualizing logical structure through delicately designed graphs [23, 40, 57, 74, 75]. For instance, QueryVis [40] translates SQL queries into diagrams based on first-order logic principles to reveal underlying logical patterns. Second, several tools facilitate inspecting intermediate execution states [6, 18, 33, 54, 97]. For example, Data Tweening [33] visualizes incremental transformations between result sets. Third, to facilitate easy code validation, tools such as DIY [59] and RATest [55] generate small sample databases or counterexamples in a sandbox environment. Fourth, several approaches utilize natural language generation to translate complex query logic into readable narratives [35, 36, 76, 78]. Systems like STEPS [78] and SQLucid [76] decompose SQL queries into clauses and provide step-by-step explanations. In addition, extensive research has been conducted in code debugging and we refer readers to the relevant surveys [9, 17]. Whereas the aforementioned tools provide various methods to assist users in syntactically understanding the logical structure and the execution output of SQL queries, our work emphasizes semantic understanding by presenting the implicit knowledge underlying

the generation process. This allows users to comprehend the assumptions and conventions reflected in the generated queries.

Another line of work provides interactive support for *query refinement*, which can be categorized into *NL-based* and *GUI-based* approaches. In *NL-based* tools, users can edit the queries by answering multi-choice clarification questions [20, 49, 87], modifying step-by-step NL explanations [76], or using free-form text [11]. For example, DialSQL [20] leverages a hierarchical encoder-decoder architecture to predict error categories, locate error spans in the query, and generate candidate choices for users to select from. SQLucid [76] proposes a fine-grained query refinement approach where users can modify steps in the NL explanation to correct errors at the clause or entity level. NL-EDIT [11] interprets natural language corrections provided by users to generate a sequence of edits that can be applied to the initial SQL query. *GUI-based* tools enable users to make simple query edits by providing widgets like menus [16, 42] and sliders [68, 73]. For instance, in DataTone [16], users can interact with the dynamically generated ambiguity widgets to correct table names, column names and data values in the query. While these refinement techniques help resolve ambiguity in natural language specifications, they generally do not capture or reuse implicit knowledge about dataset-specific conventions and task-specific computations, which our work explicitly models and exposes at the knowledge level.

Beyond SQL-specific tools, we draw inspiration from recent work on interactive NL-driven code authoring, which explores decomposing tasks into editable intermediate artifacts [32, 90, 95]. Systems like Phasewise and Stepwise [32] decompose AI-generated solutions into linear *lists* of editable assumptions and executable plans to support stepwise steering. CoLadder [90] and NeuroSync [95] expose structured and editable intent representations through hierarchical prompt *trees* and intent *graphs*, enabling programmers to externalize their goals and to refine inferred tasks. Inspired by these methods, *Cerebra* adopts a hybrid approach to query decomposition, combining a *graph*-based data flow diagram that reveals subquery dependencies with a *tree*-structured Knowledge View that provides code-knowledge highlighting for precise inspection and modification of the generated query.

2.2 Interactive NL-Driven Data Querying Tools

Beyond SQL generation, interactive data querying tools employ natural language interfaces (NLIs) to facilitate visual data exploration, addressing four primary interaction barriers when using natural language. To address the *abstraction* barrier inherent in describing high-level descriptive concepts (e.g., shapes, trends, patterns), tools such as SlopeSeeker [4] and ShapeSearch [71] allow users to query quantifiable trends and complex shapes by mapping qualitative adjectives to mathematical definitions. To overcome the *imprecision* barrier caused by vague terms (e.g., "large", "many", or "near"), systems like DataTone [16] and Eviza [68] utilize ambiguity widgets to help users select their intended data attributes and values, while approaches like [28] leverage multiple views to enable fuzzy spatial constraints for querying uncertain trajectory data. To mitigate the *formulation* barrier where users know what they exactly want but do not know how to use the system to form valid queries, Sneak Pique [69] provides data-aware autocompletion to preview results, NL2Rigel [25] maps natural language instructions to data transformation specifications, and NL4DV Toolkit [60] abstracts natural language processing complexities to aid developers in creating data querying interfaces. To lower the *continuity* barrier in maintaining conversational context, Evizeon [22] incorporates pragmatics to handle pronouns and incomplete utterances, whereas Orko [73] manages context across multimodal inputs like touch and speech.

While the aforementioned NLIs primarily address ambiguity by clarifying the operations and data references, they often overlook the *knowledge alignment* barrier, which is the gap between the user's domain-specific assumptions and the model's understanding. Unlike abstraction or imprecision barriers which arise from certain terms in the user instruction, this new barrier involves the implicit dataset-specific conventions and task-specific computations that users frequently omit from their instructions.

2.3 Human-AI Collaboration

Empirical Studies of SQL Authoring. A growing body of work has examined how people author and debug SQL, both in traditional environments [29, 57] and with NL-to-SQL tools [52, 61]. For instance, SQLShare's deployment logs and interviews show that scientists routinely reuse and layer views over user-uploaded tables to encode domain logic and processing pipelines [29]. For NL-to-SQL specifically, Ning et al. [61] derive a detailed taxonomy of NL-to-SQL model errors and demonstrate that end users have difficulty both detecting these errors and repairing them reliably through a user study. Building on these findings, several interactive systems study how explanations and direct manipulation affect SQL authoring [59, 76, 78]. These works characterize the challenges humans face in understanding and correcting SQL and NL-to-SQL outputs. In contrast, our work focuses on surfacing and aligning implicit knowledge that underlies those outputs, rather than merely their syntactic correctness.

Capturing Implicit Intentions in Human-LLM Interaction. Recent work has begun to examine how people surface and refine implicit intentions when interacting with LLMs. One line of systems [32, 70, 83] emphasizes structured task decomposition in text, helping users iteratively articulate assumptions and plans while seeing how those choices shape model behavior. A second line [88, 95, 98] focuses on visual or GUI-based representations of the model's internal reasoning or search space, so that users can better inspect and adjust what the model "thinks" it is doing. A third line [27, 53, 89] explores multimodal inputs as a more natural substrate for intent expression, allowing people to use sketches, spatial layouts, or other modalities to convey expectations that are hard to verbalize. Building on these research directions, our work extends them to the NL-to-SQL setting by modeling the implicit knowledge behind SQL queries, and by providing an interface where users can directly inspect and iteratively refine that knowledge.

3 Preliminary Study

To effectively develop LLM-driven NL-to-SQL tools, it is essential to understand the unique challenges and requirements faced by users. Prior research [59, 61, 76] conducted user studies of NL-to-SQL tools and uncovered key requirements involving query understanding, error detection, and interactive repair. These findings have

highlighted directions for enhancing LLM-driven NL-to-SQL tools. However, the role of implicit knowledge in SQL authoring has not been thoroughly investigated. To bridge this gap, we conducted a preliminary study[1] involving participants expert in SQL authoring.

3.1 Participants

We recruited 10 data practitioners (denoted as P1-P10, 9 male and 1 female, Age_{mean} = 35.6, Age_{std} = 5.40) by sending invitations via social media. They were from industry with diverse backgrounds such as Geographic Information System, Enterprise Resource Planning, and Stock Market Analysis. All participants were expert in SQL programming ($Experience_{mean}$ = 11.5 years, $Experience_{std}$ = 6.02 years) and regularly authored queries in their projects (at least once a week). They also reported moderate familiarity with LLM-driven NL-to-SQL tools (M = 4.7) on a 7-point Likert scale (1 = not at all familiar, 7 = extremely familiar). Their detailed demographic information is provided in Table 4 in the appendix. Participants consented to having their shared SQL scripts and their voices recorded.

3.2 Procedure

Prior to each interview, we provided participants with a brief overview of the study, including its purpose, procedure, and compensation, before obtaining their consent for data collection. During the one-on-one, semi-structured interviews, we first gathered background information on participants' work domains, typical tasks, and dataset characteristics. We then asked participants to describe at least one recent experience of using LLM-driven NL-to-SQL tools to author SQL scripts, encouraging them to walk through the process and discuss any difficulties or issues encountered. Furthermore, we followed up by discussing specific challenges or interesting behaviors that emerged during the interview. All the interviews were audio-recorded for subsequent analysis. The entire interview lasted around 40 minutes and each participant received 50 Chinese Yuan as compensation for their time.

3.3 Findings

We recorded all interviews and transcribed the audio recordings into text. To analyze the user feedback, we conducted an inductive content analysis [37]. The first author initially reviewed the recording transcripts to identify the relevant comments. Then similar comments were grouped into several themes. Finally, the authors reviewed the transcripts and discussed the themes on weekly meetings to reach the consensus about the key findings. In this subsection, we summarize the key findings regarding the general workflow and the challenges participants faced from the interview.

3.3.1 How Do People Work with LLM-Driven NL-to-SQL Tools to Author SQL Scripts? Drawing from the interview and participants' demonstrations of recent usage examples of LLM-driven NL-to-SQL tools, we observed a three-stage workflow: participants begin with *prompt formulation* where they craft initial queries and provide contextual information, followed by *code review* where they validate the generated SQL against their expectations, and conclude with *code refinement* where they iteratively correct and improve the results when discrepancies are found. The similar workflows are also observed in prior studies [84].

All participants supplemented contextual information in addition to their natural language queries. In the *prompt formulation* stage, participants recognized that their natural language requests alone might be insufficient and consistently provided additional context to bridge potential knowledge gaps. This supplementation took multiple forms: participants explicitly included database schemas (P1-P6, P9, P10) to clarify structural relationships and table connections that their queries assumed but didn't state. They also provided business backgrounds (P1, P3-P9) to ensure the model understood domain-specific terminology and conventions within their organizational context. For example, P3 noted that he would send a paragraph from a data report he was working on to ChatGPT for business context. The need for context supplementation highlights how natural language queries alone are insufficient, and participants must anticipate and address implicit knowledge gaps.

Participants preferred to reuse existing scripts to help LLMs learn implicit knowledge embedded within them. We noticed that more than half of the participants (6/10) tended to incorporate previously written queries into the prompt to obtain queries that aligned with the provided ones (P1, P4, P6-P9). P1 pointed out a scenario for reuse, "*I am writing queries periodically within a specific domain, my query logic is fairly fixed – only minor adjustments to parameters or subqueries*". Other participants highlighted the benefits of reuse. First, existing scripts contain knowledge about complex data transformations, filtering conditions, and computational formulas that would require extensive explanation if articulated from scratch. As P4 described, "*I usually find a script I wrote before and tweak it using AI – it's much faster that way.*" Second, since previously written scripts embodied well-tested domain logic, reusing scripts inherited this implicit knowledge while reducing validation overhead, which echoed prior research [13]. As P6 noted, "*I don't feel confident using AI to write a SQL script from scratch, so I typically reuse existing queries in the prompt.*"

3.3.2 What Makes LLM-Driven NL-to-SQL Tools Challenging to Use in Practice? We organize findings around two primary challenges of implicit knowledge transfer between humans and LLMs.

Challenges in conveying implicit knowledge (human → LLM). In the *prompt formulation* stage, participants (7/10) reported that when they omitted details they viewed as conventions or practices within their domain, LLMs typically failed to accurately interpret these instructions. Such details can be categorized into data-related and computation-related knowledge. The data-related knowledge mentioned by participants involved data values meanings (e.g., "*A value of 1 denotes offline sales, while a value of 2 indicates online sales.*" (P3)), data format constraints (e.g., "*The string length of this attribute is no more than 20.*" (P8)), and table relations (e.g., "*These eight tables are intricately interconnected.*" (P2)). The computation-related knowledge involved alias-naming standards (e.g., "*I won't give aliases of derived attributes in that way.*" (P7)), filtering conditions (e.g., "*I would avoid fuzzy matches to filter data – they produce unpredictable results.*" (P4)), and specific calculations (e.g., "*In our domain, it struggles to compute month-over-month growth rates.*" (P5)). The lack of explicit transmission of this implicit knowledge often leads to errors in the generated SQL.

[1]The study has received approval from State Key Lab of CAD&CG, Zhejiang University.

Faced with repeated failures and misunderstandings by LLMs, participants (7/10) tried to explicitly specify the knowledge by providing more detailed and procedural prompts, but found this approach burdensome. As P7 reflected, "*To get the AI to understand what I want, I end up specifying every detail like conditions, fields and joins in natural language. Finally, I spend more time writing natural language than I would have spent just writing the SQL myself.*" Furthermore, several participants (P1, P3, P8, P10) pointed out that it was error-prone to articulate such knowledge in a clear and structured prompt. "*The longer your prompt is, the more likely it is to contain inaccuracies or mistakes.*" (P8) This frustration led several participants (P4, P6) to imagine an ideal system that would be aware of task-specific context. For instance, "*It would be better if it understood the project I was working on. I would just tell it what data I wanted without further explanations.*" (P6)

Challenges in interpreting implicit knowledge (LLM → human). In the *code review* stage, participants (8/10) faced persistent difficulties in making sense of the underlying knowledge behind the LLM-generated SQL scripts, not only due to poor readability of the code (P6, P7), but more commonly because of the lack of clear alignment between their high-level requirements and the low-level code fragments (P1, P3, P6, P7, P9). As P9 summarized, "*SQL syntax itself isn't hard to understand – what's hard is figuring out which of my requirements it meets and which it doesn't.*" This lack of alignment meant participants struggled to infer what knowledge or assumptions the LLM had made in generating specific parts of the script. As a result, it became burdensome to exactly locate the code fragments failing to meet their requirements, making subsequent *code refinement* difficult. To find the error, participants had to repeatedly modify the script and review the execution results of individual subqueries or clauses (P2, P7, P8, P10). As P10 commented, "*It takes much effort to find which subquery is wrong, or which step causes the problem.*" Even if participants located the erroneous code fragments, they found it imprecise to edit the code using natural language in the *code refinement* stage (8/10).

3.4 User Requirements

We identified three requirements for LLM-driven NL-to-SQL tools in SQL authoring based on our findings. In the remaining parts of the paper, **R1**-**R3** denote the user requirements:

R1. Conveying implicit knowledge. In our interview study, participants consistently needed to provide rich contextual information, such as norms and practices of querying a dataset, in order for LLMs to understand their query intentions. However, the volume and granularity of such information, or *implicit knowledge*, made it burdensome to convey to the LLMs. Users therefore need more efficient ways to externalize implicit knowledge without repeatedly specifying every detail during query authoring.

R2. Verifying the use of implicit knowledge. Participants found it difficult to interpret how implicit knowledge was manifested in the generated SQL code. This difficulty often arose because such knowledge was typically organized or presented in an unstructured way, making it challenging for users to identify and verify relevant logic within complex queries. Bridging this gap requires mechanisms that help users comprehend the knowledge-code relationships, therefore enabling closer alignment between user knowledge and LLM's output.

R3. Refining the generated queries at the knowledge level. Participants often needed to iteratively adjust LLM-generated SQL queries when the results did not fully match their expectations. However, it was difficult for them to conveniently refine queries, because they had to work at the code level and manually locate the specific fragments that reflected their intended knowledge. Users need more convenient ways to iterate on queries at the knowledge level, so that they can adjust what the query means without laboriously editing low-level SQL code.

4 *Cerebra*

In this section, we begin with an overview of *Cerebra*'s workflow, then define and classify implicit knowledge in SQL authoring. We further describe how *Cerebra* retrieves and presents this knowledge to support code generation, review, and iterative refinement.

4.1 Overview

Cerebra is an interactive NL-to-SQL tool that aligns implicit knowledge between users and LLMs during SQL authoring. It supports SQLite [72], though it can be generalized to other SQL dialects.

4.1.1 System Workflow. In our preliminary study (Section 3.3.1), we observed that participants preferred reusing historical scripts during prompt formulation to enhance the model's code generation. Drawing inspiration from this practice, we incorporated a reuse mechanism directly into *Cerebra*'s workflow to make the implicit knowledge accessible and facilitate intelligent code suggestions (**R1**). As shown in Figure 1, *Cerebra* adopts a two-stage workflow. In the offline stage, the system establishes the knowledge base for subsequent query generation. Specifically, it generates natural language descriptions for database schemas, parses the user's historical SQL scripts into code fragments, and extracts knowledge in the form of natural language. In the online stage, when a user submits a natural language query, *Cerebra* retrieves relevant knowledge to augment the LLM prompt. It then generates the SQL script and structures the output to support the decomposition of complex queries for user review.

4.1.2 User Interface. The interface (Figure 3) is designed to support implicit knowledge sensemaking (**R2**) and knowledge-level query refinement (**R3**) by visually decomposing a complex request into components. The Input Box (Figure 3 A) accepts natural language instructions. The Script View (Figure 3 B) visualizes the generated SQL as a data flow diagram, allowing users to trace dependencies between subqueries rather than reading raw code linearly. The Knowledge View (Figure 3 C) serves as the core component for alignment. It translates code fragments into natural language "knowledge items", allowing users to verify the knowledge used in this query without parsing complex syntax. Finally, the Data View (Figure 3 D) facilitates validation by displaying execution results. These views are tightly coordinated, such that selecting a component in the diagram highlights the corresponding knowledge item and reveals the intermediate data. This enables users to iteratively

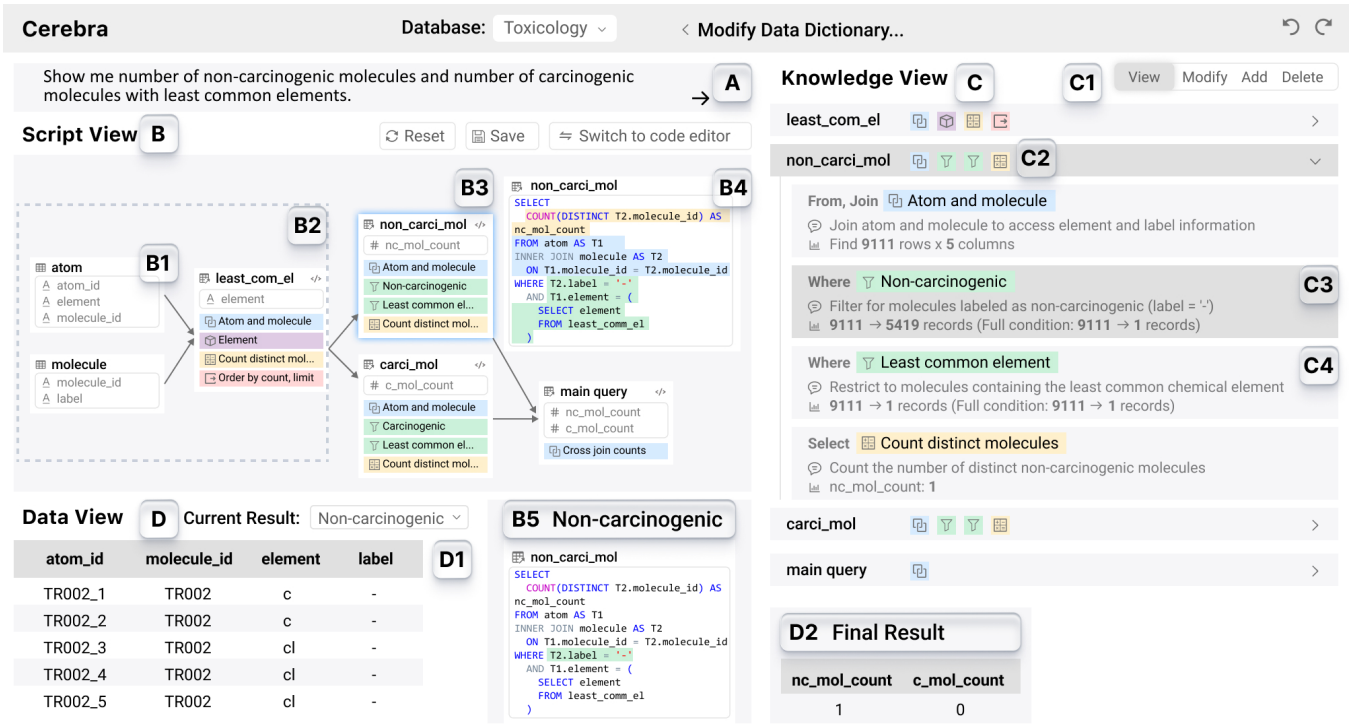

Figure 3: The interface of *Cerebra*. A) The input box where users enter their natural language queries. B) The Script View, which visualizes the structure of the generated SQL using a data flow diagram. This view displays input database tables (B1) and subqueries (B3), with links between nodes indicating dependencies (B2). Clicking on a subquery (B3) reveals its corresponding code, with highlights for relevant code fragments (B4). C) The Knowledge View listing implicit knowledge items (e.g., "non-carcinogenic" (C3) in the "non_carci_mol" subquery (C2)). Each knowledge item (C3) is linked to the corresponding code fragments in the subquery (B5). Users can iteratively refine their queries by modifying, adding, or deleting knowledge items (C1). D) The Data View, which displays the intermediate results (D1) of code fragments corresponding to specific knowledge items (C3). It can also display execution results of subqueries, such as the final results of the whole query (D2).

refine the query by focusing on specific components rather than the entire script.

4.2 Implicit Knowledge

Implicit knowledge in the NL-to-SQL process refers to the specifications that are not explicitly provided in users' natural language instructions. Typical implicit knowledge includes dataset-specific conventions and task-specific computations, which users tend to assume are understood by the model. Throughout the rest of this paper, we use the term *knowledge item* to refer to a piece of implicit knowledge, such as a filtering condition or calculation formula. Each knowledge item is associated with a specific code fragment or subquery in the SQL script.

4.2.1 Implicit Knowledge Examples. To illustrate how implicit knowledge manifests in practice, we present two examples from the Toxicology database (Figure 2) mentioned in Section 1. We focus on implicit knowledge related to the dataset-specific conventions and task-specific computations.

Example of dataset-specific conventions. Assume a data practitioner in chemistry is using the Toxicology database. She knows that "carcinogenic molecules" refers to the molecules with the "+" value in the "label" column (Figure 2 B). Here, the mapping from "carcinogenic" to the "label" column with value "+" is a form of implicit knowledge, since it is difficult to directly infer the meaning of this column just from its name, "label". However, she often assumes that the NL-to-SQL model possesses such knowledge, and thus uses terms like "carcinogenic" directly in her natural language instructions without explicitly specifying the corresponding column or value in the database.

Example of task-specific computations. Another type of implicit knowledge arises from conventions in how certain computations are performed. For example, when a data practitioner is querying the tables "atom" (Figure 4 A1) and "molecule" (Figure 4 A2), she typically interprets the "least common element" as the least common element among molecules with known carcinogenicity (Figure 4 B1). In contrast, the model may default to computing the "least common element" across all molecules in the database, regardless of whether their carcinogenicity is known (Figure 4 B2). This difference indicates that task-specific computations are often implicitly assumed by data practitioners, but may not be explicitly stated in natural language, thus require the model to possess such implicit knowledge for correct interpretation.

A1 Atom

atom_id	molecule_id	element
TR010_1	TR010	c
TR011_1	TR011	o
TR011_2	TR011	o
TR012_1	TR012	c
TR013_1	TR013	c

atom_id: Atom unique identifier

molecule_id: Identifier for the molecule to which the atom belongs

element: Chemical element symbol of the atom, such as "cl" for chlorine, "c" for carbon, "h" for hydrogen

A2 Molecule

molecule_id	label
TR010	-
TR012	-
TR013	-
TR014	+
TR015	+

molecule_id: Molecule unique identifier

label: Indicates carcinogenicity classification of the molecule. "+" indicates carcinogenic, "-" indicates non-carcinogenic

B1 Least common element

```
SELECT T1.element
FROM atom AS T1
INNER JOIN molecule AS T2
  ON T1.molecule_id =
     T2.molecule_id
GROUP BY T1.element
ORDER BY
  COUNT(DISTINCT T1.molecule_id)
LIMIT 1
```

This query only considers molecules with **known carcinogenicity**

Execution Result: **"c"**

B2 Least common element

```
SELECT T1.element
FROM atom AS T1
GROUP BY T1.element
ORDER BY
  COUNT(DISTINCT T1.molecule_id)
LIMIT 1
```

This query considers **all molecules**

Execution Result: **"o"**

Figure 4: An example of task-specific computations, which represent a form of implicit knowledge. Suppose data practitioners define the "least common element" as computed in B1 when working with the Toxicology database using the table "atom" (A1) and "molecule" (A2). If the model does not capture this implicit knowledge, it may produce incorrect results such as B2.

4.2.2 Implicit Knowledge Types. We categorize implicit knowledge according to where it appears in SQL queries based on the grammar of SQLite [72], a lightweight relational database that is widely adopted in various NL-to-SQL datasets [7, 39, 47, 93]. We identify five types of implicit knowledge: calculation, condition, relation, dimension, and output. Below, we describe each type in detail.

Calculation (Expressions in `SELECT`/`ORDER BY` clauses). Calculation knowledge involves assumptions about how attributes are computed, which are often reflected in specific formulas or the use of functions. For example, when a user asks for the "click-through rate" (CTR), the precise calculation varies between teams or organizations, as the term can refer to clicks divided by page views ($CTR = clicks \ / \ page\ views$) or clicks divided by unique users ($CTR = clicks \ / \ unique\ users$), or other more complex formulas, even though the same term is used.

Condition (Expressions in `WHERE`/`HAVING` clauses). Condition knowledge captures underspecified constraints that users expect, often conveyed through modifiers or contextual cues. For example, a query like "*calculate the number of molecules containing non-bonding elements*" leaves the definition of "*non-bonding elements*" unspecified. What counts as a non-bonding element may differ across teams or organizations, similar to calculation knowledge.

Relation (`FROM`-`JOIN` clauses). Relation knowledge refers to understanding which tables should be joined and how they are connected. For instance, as illustrated in the task-specific computations example (Section 4.2.1), when a data practitioner queries "*the least common element*", this requires joining the "atom" and "molecule" tables (Figure 4 B1), even though the join logic is not explicitly specified in the natural language instruction.

Dimension (`GROUP BY` clauses). Dimension knowledge relates to the granularity at which results are aggregated. For example, when a data practitioner asks for "*the average order value*", she does not specify whether the average should be calculated across all orders, per region, or by a combination of dimensions such as region and customer type.

Output (`ORDER BY`/`LIMIT`/`DISTINCT` clauses). Output knowledge involves how results should be presented, such as ordering, limiting, or deduplication. Similar to the dimension knowledge, users may not state these requirements explicitly, but there are often conventions or expectations for the output data.

4.3 Usage Scenario

Assume that Emma, a data practitioner in Chemistry, was working on a toxicology database project. She had authored many SQL scripts with the aid of LLM-driven NL-to-SQL tools to answer a wide range of questions. However, Emma noticed that the underlying model often generated incorrect SQL code, leading to considerable effort in carefully reviewing the generated scripts. Even after pinpointing the issues, Emma still needed to write lengthy prompts to repeatedly explain the dataset-specific conventions and clarify the task-specific computations. Seeking better coding assistance, she turned to *Cerebra*. Emma set up her workspace by connecting to the database and importing her historical SQL scripts, enabling *Cerebra* to extract knowledge automatically.

To start the query, Emma entered a natural language instruction in the input box (Figure 3 A): "*Show me number of non-carcinogenic molecules and number of carcinogenic molecules with least common elements.*" Based on this instruction, *Cerebra* automatically retrieved relevant knowledge from Emma's past queries and generated a SQL script in the Script View (Figure 3 B). Emma noticed a significant improvement compared to other NL-to-SQL tools she had used previously. *Cerebra* had automatically joined the "atom" and "molecule" tables (Figure 3 B2) to compute the least common elements in the "least_com_el" subquery, saving her effort of repeated clarification.

However, Emma was skeptical about the result (Figure 3 D2), as she believed there should be more than one molecule that met the criteria, so she first clicked the "non_carci_mol" subquery (Figure 3 B3) to check the inferred knowledge. In the Knowledge View, she immediately verified that *Cerebra* had correctly interpreted "Non-carcinogenic" as filtering molecules with label = "-" (Figure 3 C3). Besides, she noticed that the "Least common element" (Figure 3 C4) only retained a single record. Realizing that this might be due to an error in the previous "least_com_el" subquery, she clicked the subquery (Figure 5 B1) and identified an output knowledge item (Figure 5 C3) that was restricting the result to one record (Figure 5 D1). Emma identified the cause: she had

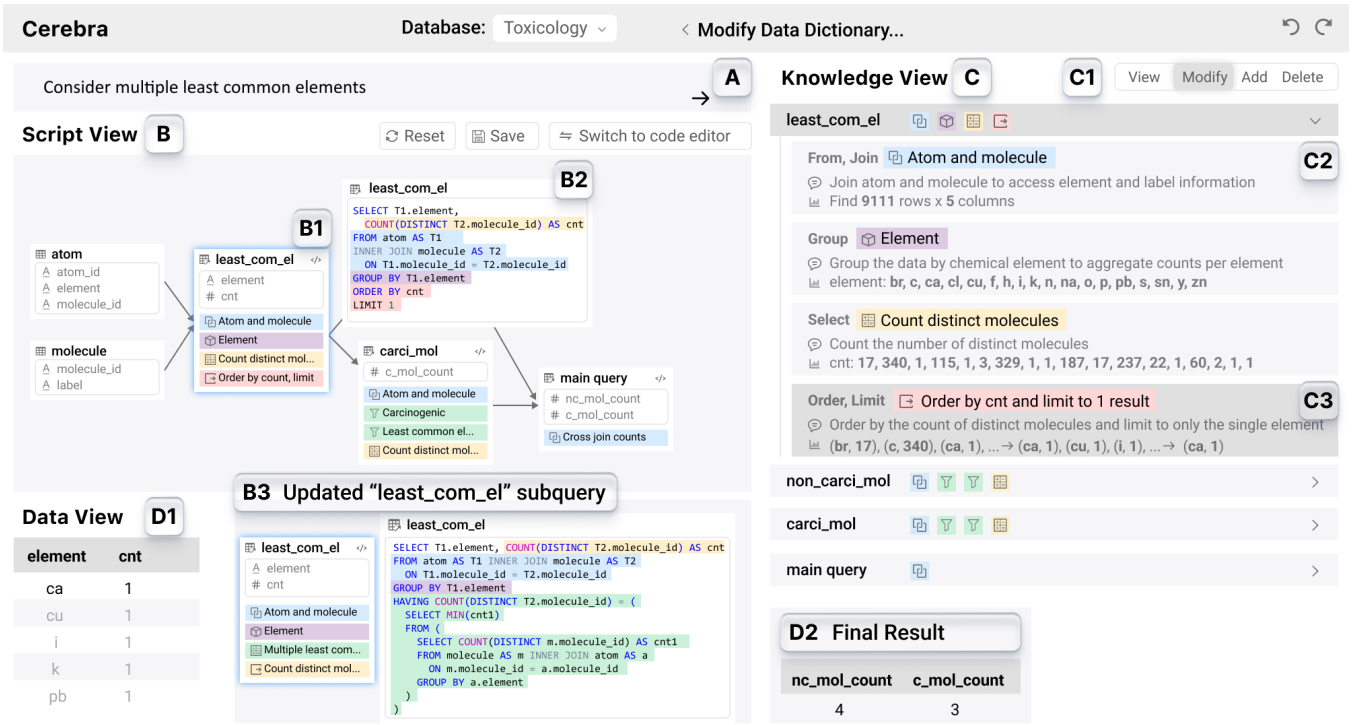

Figure 5: The usage scenario of refining the generated scripts. A) The input box, where users can enter natural language instructions to improve the query. B) The Script View, which displays the subquery that queries the least common elements (B1) and its corresponding code (B2). The refined script is shown in B3. C) The Knowledge View, which lists implicit knowledge items (e.g., C2, C3). Users can switch to "modify" mode (C1) to update the knowledge item (C3). D) The Data View, which shows the intermediate results (D1) of the knowledge item (C3). After refinement, the result of the whole query is shown in D2.

previously authored a script to query "one of the least common elements," causing *Cerebra* to automatically incorporate a `LIMIT 1` clause in the current SQL script.

To adapt the query to her current task, Emma switched to the "modify" mode (Figure 5 C1). She selected the problematic knowledge item (Figure 5 C3), and entered the instruction "*Consider multiple least common elements*" in the input box (Figure 5 A). *Cerebra* responded by replacing the output knowledge in red with a new condition knowledge in green (Figure 5 B3), and regenerated the corresponding script of the subquery. Emma observed that the final result changed from (1, 0) in Figure 3 D2 to (4, 3) in Figure 5 D2, consistent with her expectations. By reviewing the script and intermediate results, Emma verified the correctness of the outcome and completed her query authoring task.

4.4 Retrieving Knowledge for Intelligent Code Suggestions

To offer intelligent code suggestions (**R1**), *Cerebra* retrieves knowledge and injects it in prompts during the code generation process.

4.4.1 Setting Up Knowledge Extraction Process. *Cerebra* extracts knowledge leveraging users' past scripts. However, it is not a straightforward process since the accompanying code documentation such as data dictionaries is usually unavailable [50, 80], which makes it difficult to interpret the high-level semantics behind the scripts. To address this challenge, we designed an auxiliary data dictionary view in *Cerebra* (Figure 6). When users first import their database and historical scripts, they can click "Modify Data Dictionary..." on the top navigation bar (Figure 3) to switch to the data dictionary view, where they are able to interactively edit the column descriptions to supplement it with dataset-specific conventions.

The data dictionary view consists of a table list (Figure 6 A) which lists all data tables in the database and a data description view (Figure 6 B) offering column descriptions as well as sample data. *Cerebra* generates initial column descriptions according to the sample data. If users find the description of a certain column is vague (Figure 6 B1), they can double-click the descriptive text and input their partial descriptions (Figure 6 B2), and let *Cerebra* complete the rest of the description (Figure 6 B3). Once users fix the data dictionary, they can click the button (Figure 6 C) to extract knowledge based on the dictionary and past scripts (Section 4.4.2).

While interactive editing of column descriptions lowers users' effort in building a data dictionary from scratch, it can still be tedious when many columns have unclear names like the name "label" (Figure 6 B1). To mitigate the issue, *Cerebra* also considers column aliases in users' past queries (e.g., `SELECT label AS flag_carcinogenic`), as these aliases often capture users' own interpretations. These aliases are automatically parsed and used internally to enhance the quality of suggested column descriptions.

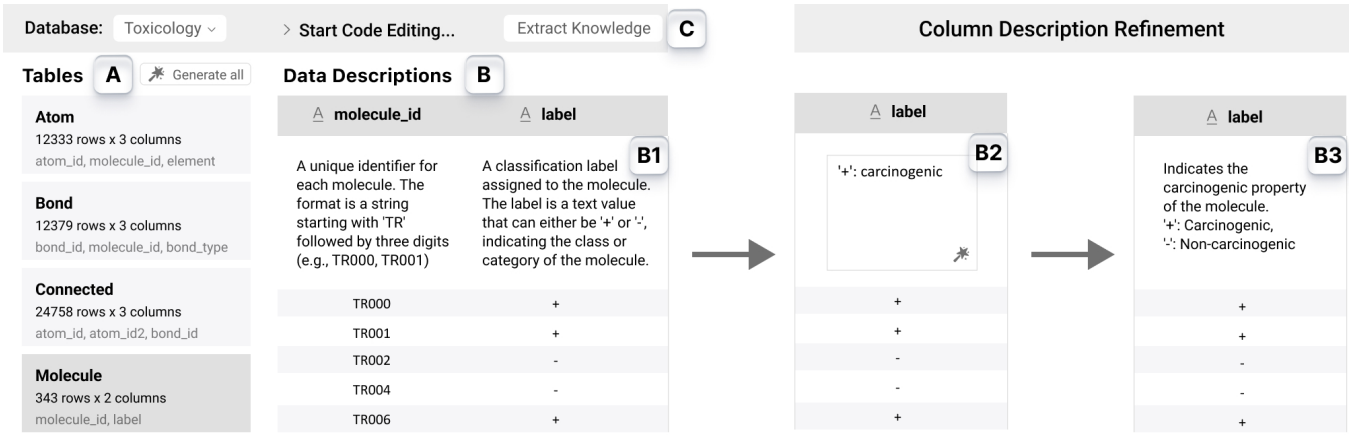

Figure 6: The data dictionary interface of *Cerebra*. A) The Table View lists basic metadata for each database table, including shapes and column names. B) The Data Descriptions View displays sample values along with generated descriptions for each column. Users can edit the column descriptions by double-clicking on the descriptive text (B1) and entering partial descriptions (B2). *Cerebra* will generate the remainder of the description based on the unique values and format of the column (B3).

By leveraging such aliases alongside sample data, *Cerebra* can better suggest meaningful column descriptions and further reduce the effort of manual annotation.

4.4.2 Extracting Knowledge from Historical Code Fragments. Directly prompting LLMs to extract knowledge from historical scripts often leads to syntactic errors in the identified code fragments, and the extracted fragments cannot be always executable to obtain the intermediate results. To address these issues, *Cerebra* first employs a rule-based approach to parse historical scripts into code fragments, before it leverages LLMs to interpret each fragment to obtain the knowledge in natural language form.

Cerebra decomposes historical scripts into well-formed code fragments that correspond to the five types of implicit knowledge (Section 4.2.2) through the Abstract Syntax Tree (AST). For calculation knowledge, expressions are extracted by splitting the `SELECT` or `ORDER BY` clause at commas, treating each formula as a standalone calculation fragment, while preserving the integrity of nested formulas. For condition knowledge, compound logical conditions in `WHERE` or `HAVING` clauses are further decomposed into atomic conditions, ensuring that each simple predicate is extracted as an individual fragment. For the remaining knowledge types (dimension, output, and relation), the corresponding `GROUP BY`, `ORDER BY`/`LIMIT`/`DISTINCT`, and `FROM`/`JOIN` clauses are each extracted as a whole, preserving their original structure. Detailed parsing rules can be found in the supplementary material.

After parsing, *Cerebra* leverages LLMs to interpret each historical query by first generating a natural language description of the whole script using the data dictionary (Figure 6) for context. For example, for the query in Figure 4 B1, the script-level description is: "*Find the least common element.*" Then, *Cerebra* prompts the LLM to produce descriptions for each individual code fragment. For instance, the fragment `FROM atom AS T1 INNER JOIN molecule AS T2` is described as: "*Join atoms and molecules to filter molecules with known carcinogenicity.*" Both script-level and fragment-level descriptions are embedded for retrieval, enabling *Cerebra* to better match user instructions to relevant knowledge.

4.4.3 Injecting Knowledge into Prompts. When users input a natural language instruction (Figure 3 A), *Cerebra* retrieves both script-level and fragment-level knowledge before injecting the knowledge into prompts to generate SQL scripts. Initially, it matches the user instruction against the embedded script-level descriptions of historical queries, using cosine similarity to measure semantic relevance. Given that the user instructions often do not correspond exactly to any single past script, *Cerebra* then decomposes the user's instruction into multiple keywords with the help of the LLM, and matches each keyword with the embedded fragment-level descriptions. For example, given an instruction such as "*Show me number of non-carcinogenic molecules with non-bonding element*", *Cerebra* extracts keywords like "*non-carcinogenic molecules*" and "*non-bonding element*", and then retrieves relevant SQL fragments corresponding to these keywords. Additionally, *Cerebra* presents all retrieved scripts and associated knowledge to the LLM for filtering and re-ranking, prioritizing examples most relevant to the current task. Finally, *Cerebra* uses the re-ranked scripts, associated knowledge, and the data dictionary as input for the LLM to generate the SQL code.

4.5 Making Implicit Knowledge Explicit for Effective Code Review

To address the challenge of knowledge-code misalignment identified in our preliminary study, *Cerebra* is designed to facilitate code review by presenting both the code structure and the knowledge inferred during code generation (**R2**). This enables users to easily inspect which aspects of their knowledge are satisfied or missing in the generated query.

4.5.1 Visualizing the Data Flow of SQL Queries in the Script View. The Script View (Figure 3 B) provides users with an overview of the entire SQL script. It uses a data flow diagram to visualize the table-level lineage of the generated SQL script, including input

tables (e.g., Figure 3 B1), subqueries (e.g., Figure 3 B3), and their dependencies. Each subquery is represented as a card displaying key metadata such as subquery names, output columns, column types, and a list of knowledge items. Clicking on a subquery card (Figure 3 B3) highlights the corresponding SQL code (Figure 3 B4) and shows the intermediate execution results.

To support the inspection of intermediate execution results, *Cerebra* computes a dependency graph among subqueries based on the ASTs of the SQL script. For each subquery, dependencies are recursively resolved by identifying all referenced tables and Common Table Expressions (CTEs). Therefore, to compute the execution result of a specific subquery, *Cerebra* will automatically execute all its dependent subqueries in the correct order to avoid syntax errors caused by undefined subqueries.

4.5.2 Presenting Knowledge in the Knowledge View. The Knowledge View (Figure 3 C) presents a structured list of the knowledge items extracted from the SQL script, organized by subqueries and clauses. All knowledge items are grouped in a flat, two-level hierarchy: the outer level corresponds to subquery names (Figure 3 C2), while the inner level lists individual knowledge items (e.g., Figure 3 C3) associated with specific SQL code fragments, following the execution order. While this flat, two-level design inevitably omits some hierarchical information of knowledge items (e.g., two dependent knowledge items in different subqueries), it simplifies reading without being overwhelmed by deeply nested SQL structures. Besides, users can still inspect the hierarchical dependencies by referring to the Script View if needed.

Each knowledge item corresponds to one of the five implicit knowledge types (see Section 4.2.2) and is presented with a two-line display: the first line is a code fragment-level natural language description similar to that in Section 4.4.2, while the second line presents relevant metadata of execution results of the code fragment for further validation. Specifically, for the code fragment corresponding to calculation knowledge, the metadata shows sample values computed by the expression in the `SELECT` statement; for condition knowledge, it reports both the number of records filtered by the atomic condition and by the overall composite condition; for relation knowledge, it displays the number of rows and columns after joining the relevant tables; for dimension knowledge, it lists the distinct values present in the aggregation dimension; and for output knowledge, it presents a sample of the final output records.

When users click on a knowledge item (Figure 3 C3), *Cerebra* displays the complete execution results in the Data View (Figure 3 D1), along with highlighting the corresponding code fragment (Figure 3 B5). Since computing intermediate results for code fragments can be time-consuming, *Cerebra* adopts an asynchronous loading strategy to incrementally present results as they become available, helping shorten user waiting time and improve interactivity.

4.6 Refining Knowledge for Effective Query Modification

While historical script retrieval enables *Cerebra* to capture the implicit knowledge embedded in users' prior work, we found that code reuse alone is insufficient for supporting users' evolving query requirements. To better support code editing, *Cerebra* allows users to incrementally refine the queries by modifying, adding, or deleting (Figure 3 C1) the knowledge in the Knowledge View (**R3**).

When modifying, users can select a subquery (e.g., Figure 3 C2) or a specific knowledge item (e.g., Figure 3 C3), and enter an instruction in the input box (Figure 3 A) to update the corresponding code fragments. To add new knowledge, users can build on an existing subquery or select the entire SQL query (i.e., the main query in Figure 3) and provide an additional instruction. This triggers *Cerebra* to retrieve and integrate relevant knowledge from historical scripts. For deleting, users simply select knowledge items to remove them from the query.

To ensure the syntactic correctness of the entire query after any modification, *Cerebra* maintains a dependency graph for all subqueries and their associated knowledge items, constructed from the AST of the whole SQL script. When a user edits a knowledge item, *Cerebra* first locates the subquery that incorporates the modified code fragment. It then recursively identifies all downstream subqueries that depend on the outputs of the affected subquery, following the edges in the dependency graph. For each downstream subquery, *Cerebra* automatically updates the SQL code to reflect changes in referenced fields or aliases. It then re-executes these subqueries in topological order, ensuring that any intermediate results and final outputs remain consistent with the updated knowledge.

4.7 System Implementation

We implemented the frontend of *Cerebra* using TypeScript in web browsers, in combination with libraries involving Vue.js, Monaco Editor [56], and dagre [10]. The Monaco Editor was used to display and highlight the source code of subqueries, while dagre was used for automatic layout of the dataflow diagram in the Script View. The backend of *Cerebra* was implemented in Python, utilizing SQLGlot [79] for SQL parsing, SentenceTransformers [65] to compute word embeddings, and Qwen3 [86] as the underlying LLM.

5 User Study

To evaluate the effectiveness and usability of *Cerebra*, we conducted a user study[2] where participants completed SQL authoring tasks using *Cerebra* and a baseline tool. We focus on research questions:

- **RQ1 (overall experience)**: To what extent does *Cerebra* improve the NL-to-SQL authoring process?
- **RQ2 (steering strategies)**: How do users navigate different steering mechanisms to align the model with their intent?
- **RQ3 (mental models)**: How do users understand implicit knowledge and adjust the explicitness of their prompts?
- **RQ4 (transparency & trust)**: How does *Cerebra* affect users' understanding and trust in the generated SQL?

5.1 Participants

In our user study, we recruited 16 data practitioners (denoted as U1-U16, 10 male and 6 female, $Age_{mean} = 25.75$, $Age_{std} = 5.57$) through two channels: 11 participants were recruited via a university's internal forum, and the other 5 were recruited from industry through social media. Participants had diverse backgrounds, including Cloud Computing, Biology, Chemical Engineering, Gaming, and

[2]The study has received approval from State Key Lab of CAD&CG, Zhejiang University.

E-Commerce. All participants were expert in SQL programming ($Experience_{mean}$ = 3.94 years, $Experience_{std}$ = 4.02 years) and routinely authored queries in their work (at least once a week). They also reported moderate familiarity with LLM-driven NL-to-SQL tools (M = 4.8) on a 7-point Likert scale (1 = not at all familiar, 7 = extremely familiar). Their detailed demographic information is presented in Table 5 of the appendix. All participants gave consent for the recording of both their voices and programming processes.

5.2 Apparatus and Materials

5.2.1 Baseline. To evaluate the integration of knowledge in *Cerebra*, we designed the baseline tool (denoted as Baseline) as an ablation version that removed all features related to knowledge in both code generation and interface design. Specifically, in code generation, Baseline used the same backend LLM, but generated SQL queries solely based on user instructions, data schemas, and column descriptions, without retrieving knowledge from existing scripts. In the interface, Baseline omitted all knowledge presentation by replacing the Knowledge View with a simple chat interface, where users could enter natural language instructions and receive generated SQL scripts with textual explanations. The data flow diagram in the Script View was retained, but knowledge items in the subquery cards and their corresponding highlights in the code were removed. Baseline enabled users to generate, review, and refine SQL scripts via LLM-driven code generation.

5.2.2 Datasets. To design a comparative study, we selected two datasets, denoted as *European Football* and *Toxicology*, on the cross-domain NL-to-SQL dataset, BIRD [47]. Compared to other datasets such as Spider [93] and Spider 2.0 [39], BIRD features not only complex and large-scale databases, but also a larger number of SQL scripts per individual database. This allowed us to better evaluate the effectiveness of knowledge extraction and historical script reuse in *Cerebra*. Each dataset contains 11 data columns and more than 50 existing SQL scripts. Both datasets come with predefined database schemas and column descriptions, which were provided to participants directly and did not require any user editing.

5.2.3 Tasks. For each dataset, we designed a statistical task that required participants to compose a SQL query. The final query of each task was designed to be approximately 30 lines long and to include 3 subqueries. To reduce the impact of initial model output deviations on the experiment, we provided a dedicated initial prompt for each task, following the practice adopted in prior work [84]. Detailed task descriptions, including initial prompts and expected outputs, are provided in the supplementary materials.

5.3 Procedure

We adopted a counterbalanced mixed design to compare *Cerebra* and Baseline. We denoted the two tools as C(erebra) and B(aseline), and the two datasets as E(uropean Football) and T(oxicology). Participants were divided into four groups to cover all combinations of tools and datasets in the following experimental conditions: [CT, BE], [BE, CT], [BT, CE], [CE, BT], allowing each participant to experience both tools and mitigating learning effects.

At the start of the study, we informed participants about the relevant background. We then introduced the first code authoring task, during which they practiced with the assigned tool in a warm-up phase and received a slide deck containing task descriptions, relevant knowledge, and data dictionaries. After completing the first task, which involved authoring SQL scripts with either *Cerebra* or Baseline, participants filled out a NASA-TLX [21] questionnaire, with each question measured on a 7-point Likert scale, to assess perceived workload. Subsequently, we switched to the second code authoring task using a different dataset and tool, following the same procedure. After both code authoring tasks, we conducted a semi-structured interview comprising three parts. First, we asked participants to compare their experiences with *Cerebra* and Baseline in terms of code generation, understanding, and refinement. Second, we encouraged participants to share their suggestions for improving the tool. Third, we asked participants follow-up questions regarding specific issues that arose during the SQL authoring process. The entire study took around 80 minutes and each participant received 70 Chinese Yuan as compensation.

During the study, we recorded all screen activities and audio. To analyze the interaction patterns, two authors (SQL authoring experience: $>$ 3 *years*) manually annotated participants' actions from the screen-capture videos. Disagreements were discussed and resolved in periodic meetings. When consensus could not be reached, a third author (SQL authoring experience: $>$ 10 *years*) was consulted to make the final decision. For the semi-structured interviews, we transcribed all audio recordings and conducted an inductive content analysis [37] to identify recurring themes and patterns in participants' feedback.

5.4 Results

5.4.1 Overall User Experience (RQ1). Most participants (14/16) thought that they could better understand the generated code when using *Cerebra*, saying it "*intuitive*" (U4, U5, U7, U8, U12) or "*clear*" (U2, U3, U10, U15). Many participants (13/16) found that their natural language prompts were shorter and more concise when using *Cerebra*. U2 further remarked, "*It is more like human communication rather than writing the lengthy code*". Meanwhile, nearly three-quarters of participants (11/16) said that the code generation of *Cerebra* was "*better*" compared to Baseline, saying it "*accurate*" (U1-U8, U10) or "*smart*" (U12, U16). Participants also found it easy to refine the generated SQL. As U11 said, "*I don't need to tell it (Cerebra) which lines of code I want to modify since I can directly select the knowledge that I want to change.*"

These subjective experiences of improved workflow were reflected in objective measures. Participants completed tasks significantly faster with *Cerebra* than with Baseline (ANOVA, main effect of tool: $p < 0.01$). Furthermore, participants reported feeling less burdened when using *Cerebra*, rating their perceived Performance ($p < 0.01$) and Effort ($p < 0.05$) workload significantly lower on the NASA-TLX scale compared to Baseline. Detailed quantitative results are provided in Appendix B.

To better understand the actual usage of *Cerebra*, we analyzed participants' interaction patterns based on the annotated screen-capture videos. we grouped participants' actions into two broad categories: **query construction** and **code understanding**. Actions related to **query construction** include **Prompt Initialization**

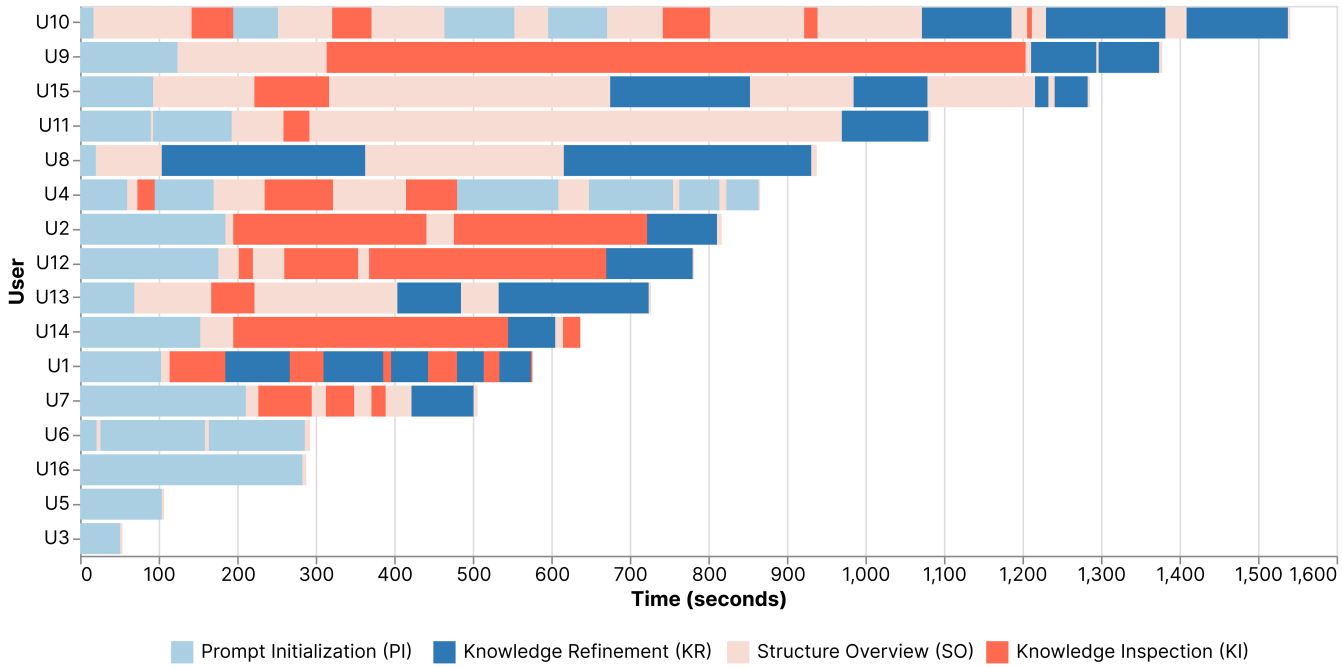

Figure 7: Interaction timelines for all participants using *Cerebra*. Each bar shows one participant's SQL authoring task. Participants may initialize a prompt (PI), have an overview of the generated SQL (SO), inspect the knowledge (KI), and refine the knowledge (KR). Participants are ordered by the task completion time in descending order.

(PI), which refers to composing or rewriting the prompt to generate the whole SQL query, and **Knowledge Refinement (KR)**, which leverages Knowledge View to update the corresponding components of the SQL. **Code understanding** actions include **Structure Overview (SO)** and **Knowledge Inspection (KI)**. SO captures interactions with the Script View and its linked Data View to understand the overall structure of the generated SQL, while **KI** corresponds to interactions with the Knowledge View and its linked Data View to check the knowledge.

Figure 7 reveals several interaction patterns across participants. First, a small group of participants (U6, U16, U5, U3) completed the task in a very short time and their timelines were dominated by PI segments, with almost no time spent on **KI** or **KR**. These participants treated *Cerebra* primarily as a prompt-based code generator and rarely engaged with the Knowledge View. Second, several participants (U10, U4, U6) repeatedly returned to PI throughout the task, issuing multiple fresh prompts instead of relying mainly on knowledge-level edits, which suggests a preference for resetting the solution when they perceived large mismatches with their intent. Third, some participants (U9, U15, U11, U2, U12, U14) showed longer continuous periods of SO and **KI** after the first query was generated, followed by shorter **KR** segments. They invested substantial time in understanding the generated script and its intermediate results before attempting any edits at the knowledge level. Across these participants, a common pattern was that many of them did not adhere to a single strategy, but instead switched back and forth between **code understanding** actions and **query construction** actions as the task progressed.

5.4.2 Knowledge-Level Steering and Agency (RQ2). Participants (8/16) described a spectrum of steering strategies across natural language prompts, knowledge refinement, and direct code edits, and shifted between these based on three primary factors: workload, accuracy, and expressiveness. First, some participants (U5, U9, U10, U15) would roughly assess the workload of different steering strategies, choosing the way that minimized effort given the perceived level of misalignment. As U15 explained, "*If the generated SQL is completely wrong, then I'd definitely just write a new NL instruction and regenerate. But if it only gets some small parts wrong, then using the knowledge refinement or directly editing the SQL is more convenient.*" Second, historical model accuracy also shaped which channel they used. After repeated failures of query refinement, participants (U2, U7, U9, U12, U16) became reluctant to keep trying to rephrase the natural language instruction. Moreover, some participants (U12, U15) reported that certain details of implementation and refinement were hard to describe in natural language, especially when they already had a clear target SQL statement in mind. As U12 reflected on her experience of repeatedly correcting an incorrect answer through natural language only, "*After I saw the wrong answer, I started thinking about how to phrase my instructions in natural language, so that the system would change its answer into the one I wanted. Sometimes I felt that process was quite exhausting.*"

5.4.3 Mental Models of Implicit Knowledge (RQ3). Participants developed mental models of what knowledge the tools possessed by observing the model's behavior and adjusting how explicitly they specified the knowledge in prompts. Many participants (12/16) began tasks with probing the implicit knowledge boundaries of both

tools through trial and error. As U6 pointed out, "*I would start with a relatively concise prompt to explore how much it (Cerebra) already knows.*" When the tool's behavior exceeded their expectations, some participants became more willing to rely on shorter descriptions in subsequent turns. As U12 commented, "*I didn't expect it (Cerebra) to be that smart, but after seeing it perform well, I tended to describe my problem with a much shorter prompt.*" As participants (U1, U4, U5, U12, U13, U16) progressed through the tasks, they formed distinct views about which aspects of their own knowledge had to be spelled out and which could be left implicit, and these views differed across individuals. For example, as U4 noted, "*For custom, project-specific concepts, the model (of Baseline) definitely doesn't know them, so I need to explain them; but for the ones like the concepts of atoms and molecules that are already recognized in a domain, I rely more on the model's own knowledge.*" U5 focused on the contested terminology, "*If the same norm can have different meanings in different domains, I will clarify it.*" U16 drew a boundary between simple and intricate knowledge: "*Things like ratio are often complex, so I can't assume it (Baseline) knows exactly how I want that computed.*"

5.4.4 Transparency and Trust (RQ4). Several participants (U2, U3, U6, U7, U12) described Baseline as closer to a black box, whereas *Cerebra*'s Script View and Knowledge View made the structure and intermediate steps more intuitive to inspect. U3 appreciated the Knowledge View, "*The tool (Cerebra) lets me see each small step, with color highlighting that makes it easier to understand.*" Several participants noticed that *Cerebra* was better at showing the semantics of the query (U1, U6, U7, U12). As U1 put it, "*Directly reading code is an inherently inefficient way to check a query. The strength of the tool (Cerebra) is not that it makes the code easier to read, but that it more explicitly exposes the execution logic.*" When using *Cerebra*, many participants (13/16) relied on the Data View and the metadata attached to knowledge items to validate the correctness of the query, while there was a small number of participants (3/16) still treated raw code as the ultimate source of truth. For example, U9 remarked, "*In the first tool (Cerebra), the code is hidden by default. If there were an always-on code view like that in Baseline, I would feel more trust.*" Some participants (2/16) further expressed a desire for *Cerebra* to explicitly display its own knowledge before they entered their prompts, highlighting the need for greater transparency. As U15 suggested, "*It (Cerebra) could use natural language to tell us what knowledge it already has, so that we can, in turn, write our prompts using the terms it is familiar with.*"

6 Technical Evaluation

We conducted a technical evaluation of the performance and robustness of *Cerebra*'s core components. While the user study focused on the end-to-end user experience, this evaluation aimed to quantitatively assess the system's model capabilities, specifically the accuracy of knowledge extraction, the precision and recall of knowledge retrieval, and the robustness of the knowledge refinement.

6.1 Dataset Preparation

Since there are no existing open-source datasets specifically designed for evaluating SQL generation based on historical scripts and implicit knowledge retrieval, we built a custom dataset derived from BIRD [47]. The custom dataset contains 232 evaluation tasks spanning 4 databases, including *Toxicology* and *European Football*, which were utilized in our user study, as well as two additional databases, namely *Codebase Community* and *Formula 1*. Each task consists of a natural language description, a ground-truth SQL query, and ground-truth knowledge items to be reused.

To balance the data quality and human effort, we employed a two-stage task construction process for each database. In the first stage, we treated the SQL scripts in BIRD dataset as users' historical queries and randomly sampled 1 to 5 scripts as context. We used Qwen3 [86], the same LLM employed in our system implementation (Section 4.7), to automatically generate candidate tasks. These tasks were created under constraints that require the synthesized SQL to be executable, to produce non-empty results, and to maintain a complexity of approximately 30 lines of code, consistent with the difficulty level used in our user study (Section 5.2). In the second stage, two authors conducted cross-validation of all tasks, manually verifying the consistency between the natural language description and the SQL query, as well as the correctness of the identified knowledge reuse. Any issues identified were manually corrected.

6.2 Apparatus and Metrics

Similar to Section 6.1, we utilized Qwen3 for all procedures involving LLM usage throughout our experiments, and the all-MiniLM-L6-v2[3] text embedding model for knowledge retrieval. Across all system components, we adopted a consistent procedure in which two authors manually cross-validated the semantic correctness of the outputs to determine the metric (e.g., execution accuracy).

6.2.1 Knowledge Extraction. Direct evaluation of the extraction accuracy is difficult, because the BIRD dataset provides no ground-truth knowledge items and the extracted knowledge in *Cerebra*'s workflow is represented in natural language. We therefore introduced an indirect evaluation metric, namely *SQL Reconstruction Accuracy*, which measures the execution accuracy of SQL queries reconstructed solely from the extracted knowledge.

6.2.2 Knowledge Retrieval and Code Generation. For knowledge retrieval, we calculated the precision and recall of retrieved knowledge items. Here, a knowledge item is defined in the same way as in *Cerebra*'s workflow (Section 4.2) and corresponds to a specific code fragment in the historical SQL scripts. For code generation, we measured execution accuracy for both direct generation and retrieval-augmented generation.

6.2.3 Knowledge Refinement. This evaluation used 67 tasks that failed during initial retrieval-augmented code generation. We define one refinement step as follows: given an incorrect SQL query and its ground-truth counterpart, the model first produces a natural language instruction about code modification, and is then invoked again with only the incorrect SQL and this instruction to revise the original query. We limited the refinement process to a maximum of 5 steps to avoid non-convergent cases.

6.3 Results

6.3.1 Knowledge Extraction Accuracy. As shown in Table 1, the proposed knowledge extraction method demonstrated high accuracy,

[3] https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

achieving a success ratio exceeding 95% across all databases. The Codebase Community database yielded the highest performance with a 96.77% success ratio out of 186 total historical queries. Despite the various domains and size of historical queries, the absolute number of failed tasks remained consistently low at exactly 6 per database. This consistency indicated that our extraction method could effectively capture the necessary knowledge to construct valid SQL queries for the vast majority of cases.

Table 1: SQL reconstruction accuracy of Knowledge Extraction across four databases.

Database	History	Success	Success Ratio
Toxicology	145	139	95.86
European	129	123	95.35
Codebase	186	180	96.77
Formula 1	174	168	96.55

6.3.2 Knowledge Retrieval Performance. Table 2 summarizes the performance of the retrieval module. The results indicated a design choice for high recall to ensure coverage of relevant knowledge. Across all databases, the number of retrieved items ("Retrieved") was consistently higher than the ground-truth knowledge items ("Items"), which naturally diluted Precision (ranging from approximately 43% to 51%) but boosted Recall. The Codebase Community database achieved the highest Recall at 87.67%, while the European Football exhibited the most balanced performance, achieving the highest Precision (51.63%) and F1 Score (63.34%).

Table 2: Performance metrics for Knowledge Retrieval.

Database	Items	Retrieved	Precision	Recall	F1
Toxicology	334	546	43.04	71.18	53.64
European	262	422	51.63	81.92	63.34
Codebase	251	514	43.49	87.67	58.14
Formula 1	360	645	44.74	73.20	55.53

6.3.3 Impact of Knowledge Retrieval and Refinement. Table 3 presents an ablation study comparing different configurations of the SQL generation pipeline. The *Direct* generation baseline performed poorly across all databases, with success ratios ranging from 22.22% to 40.43%, highlighting the difficulty of generating accurate SQL queries without integration of knowledge. The introduction of relevant knowledge (*RAG*) yielded a notable improvement, increasing the success rate by 29% to 47% across four databases. Furthermore, the *Pipeline* which combines both RAG and the *Refinement* module, consistently achieved the highest accuracy, surpassing 90% across all four databases evaluated, with the highest accuracy of 94.03% on Codebase Community. The Pipeline success ratio demonstrated the robustness of the Knowledge Refinement module in correcting errors in the generated SQL.

Table 3: Ablation study on Code Generation and Knowledge Refinement. *Direct* denotes generation without knowledge retrieval, *RAG* adds retrieval augmentation, *Refine* tests the refinement module independently, and *Pipeline* represents the complete pipeline.

Database	Mode	Tasks	Success	Success Ratio
Toxicology	Direct	55	20	36.36
	RAG	55	40	72.73
	Refine	15	10	66.67
	Pipeline	55	50	90.91
European	Direct	47	19	40.43
	RAG	47	33	70.21
	Refine	14	11	78.57
	Pipeline	47	44	93.62
Codebase	Direct	67	27	40.30
	RAG	67	48	71.64
	Refine	19	15	78.95
	Pipeline	67	63	94.03
Formula 1	Direct	63	14	22.22
	RAG	63	44	69.84
	Refine	19	14	73.68
	Pipeline	63	58	92.06

6.3.4 Failure Cases. In knowledge extraction, subtle errors may occur even when the correct facts were captured. Specifically, the extracted descriptions did not always strictly preserve the original output format, which could cause the reconstruction step to change the order of columns or to include unnecessary columns. Knowledge retrieval was further affected by these imperfections, since in *Cerebra*'s workflow, the retrieval module matches user natural language instructions with knowledge that is also stored in natural language. Semantic gaps or differences in phrasing between the two may prevent certain relevant knowledge items from being retrieved, which then led to downstream code generation failures. Finally, SQL refinement sometimes failed to correct these problems because of hallucinations from the underlying LLM, which may introduce non-existent columns or modify predicates in ways that were not supported by the underlying schema, resulting in refined queries that still executed incorrectly.

7 Discussion

In this section, we discuss the implications of our work for NL-to-SQL systems, situate our approach within the context of existing research on knowledge in NL-to-SQL, and outline current limitations and directions for future work.

7.1 Characterizing and Reusing Knowledge in NL-to-SQL

7.1.1 Knowledge for NL-to-SQL. The notion of "knowledge" has received increasing attention in NL-to-SQL research since 2021 [51], and has been referred to by different names including external knowledge [39, 47], domain knowledge [14, 38], and ambiguity [26, 96]. From the perspective of coverage, "domain knowledge" and

"ambiguity" in prior work mainly include knowledge about dataset-specific conventions, such as abbreviations, conditions, relations between tables, synonyms, and entity-column mappings. However, they do not sufficiently cover knowledge about task-specific computations, which is also an important part included in implicit knowledge. From the perspective of knowledge representation, "external knowledge" in prior work is typically manifested as unstructured documentations and scripts scattered across the database project, making it challenging to utilize effectively. Our work categorizes implicit knowledge according to the place it appears in SQL queries (Section 4.2.2) to facilitate structured reuse of the knowledge from users' historical scripts.

7.1.2 Reuse as a Mechanism for Semantic Alignment in Data Analysis. Reuse has long been established as a fundamental strategy in data analysis tools to reduce authoring effort and lower technical barriers. Prior work has successfully demonstrated the value of reuse in generating infographics [64, 91], authoring data analysis code [48], recommending visualizations [12], and constructing data reports [77]. These approaches typically accelerate creation by reusing explicit artifacts, such as visual templates, code snippets, or narrative structures. *Cerebra* extends this paradigm to the domain of database querying, but shifts the emphasis of reuse from the structure or style to semantics. By treating historical scripts as a repository of codified conventions, *Cerebra* leverages reuse to ground the LLM's generation in the user's specific context. This shift suggests that in AI-assisted data analysis, the value of reuse lies in stabilizing the stochastic nature of LLMs by anchoring them to the verified logic embedded in the user's interaction history.

7.2 Knowledge Management in NL-to-SQL Generation

7.2.1 Deciding What Types of Knowledge to Make Explicit, Traceable, or Ephemeral. In high-stakes contexts such as finance or manufacturing, misapplied knowledge can lead to financial losses or legal liability. In these settings, knowledge on computational logic needs to be made fully explicit and fixed, rather than inferred by embeddings or LLMs. A future version of *Cerebra* could include alert mechanisms that flag when unverified knowledge is being used in such sensitive computations. A second category involves knowledge that must be traceable, for example norms and formulas whose authorship and temporal validity matter in multi-stakeholder environments. Here, who introduced a convention and when it was last updated are as important as the rule itself, such as interest rate parameters that change over time. In the future, *Cerebra* could be extended with auditing features that associate knowledge items with provenance and version histories. Finally, for the tasks with low risk, it is often more efficient to let LLMs generate task-specific computations ephemerally without the overhead of formalization, as long as users can revise the resulting knowledge when needed.

7.2.2 Maintaining Knowledge Validity under Schema Evolution. Since *Cerebra* extracts knowledge from specific historical code snippets, changes in table structures, column names, or data values may cause previously extracted knowledge to become partially stale or even misleading. One potential approach to improving the resilience of *Cerebra* is to establish links between the entities in these snippets (e.g., table and column names) and the live database schema. When schema updates occur, the system can identify the knowledge items impacted by the modified entities and automatically regenerate their corresponding SQL snippets as well as natural language descriptions. Future work can further investigate such incremental update mechanisms and evaluate their effectiveness in preserving the reliability of extracted knowledge.

7.3 Balancing LLM Automation and Human Intervention

To balance cognitive load against query precision, *Cerebra* hands off labor-intensive information processing tasks to the LLM. Such tasks involve column description generation in the data dictionary and knowledge extraction from massive SQL scripts, where manual processing is prohibitively expensive. Conversely, tasks like final SQL generation that directly determine the correctness of data queries, are designated to be verified and crafted by humans through the interface. However, errors in the upstream LLM automation (e.g., inaccurate column descriptions or extracted knowledge) can cascade [3], degrading the accuracy of downstream retrieval and generation. Yet, requiring users to manually inspect every generated description or extracted knowledge item is impractical, especially for ad-hoc queries where efficiency is paramount.

To address this trade-off between automation efficiency and error propagation, a compromise is to employ indirect metrics to rapidly localize LLM errors without exhaustive manual review [3, 34, 70]. For instance, our technical evaluation assesses the quality of knowledge extraction by measuring the accuracy of reconstructing SQL queries from the extracted knowledge, using reconstruction failure as a proxy for extraction error. In the future, we aim to generalize this approach by adopting frameworks like EvalGen [70] and EvalLM [34], allowing users to define high-level criteria or metrics (e.g., "*consistency with naming conventions*") to automatically audit the LLM's outputs. By shifting the human role from reviewing individual outputs to interactively defining and validating these evaluation criteria, *Cerebra* can better foster user trust and interpretability for such labor-intensive information processing while minimizing the manual effort required for verification.

7.4 Threats to Validity

In this section, we discuss the threats to validity for our preliminary study and user study.

For the preliminary study, we acknowledge that participants' individual workflows and prior experience with NL-to-SQL tools could introduce bias, potentially influencing the interview results. To mitigate this threat, we recruited a cohort of 10 data practitioners from diverse industrial domains, ensuring that our findings were not overfitted to a specific data schema or workflow.

For the user study, we recognize that participants performed tasks on domain-specific datasets (Sports, Chemistry) with which they lacked prior familiarity. However, this design aligns with our goal to evaluate *Cerebra* for a broad spectrum of data practitioners, regardless of their specific domain background. We argue that implicit domain knowledge consists of two distinct facets: task-specific computations (e.g., scientific formulas) and dataset conventions (e.g., how attributes are stored and linked). While end-user scientists

may possess the former, general data practitioners frequently face the challenge of navigating dataset conventions to write accurate SQL. For instance, calculating a "student's average grade" requires not just the arithmetic logic but also understanding the specific schema conventions for locating and aggregating those grades.

Therefore, our study tasks were designed to encompass both conventions and computations. To bridge the knowledge gap, we provided tutorial training that enabled participants to grasp the necessary context. The results showed that no participant failed due to a lack of domain knowledge, validating this approach. However, we acknowledge that evaluating *Cerebra* with domain experts like end-user chemical scientists could provide a deeper understanding of how implicit domain conventions are externalized and reused during SQL authoring. Such an evaluation poses additional challenges such as recruiting sufficient domain experts, designing comparable tasks across conditions, and disentangling the effects of SQL skill from domain familiarity. We leave such a study to future work to further extend the external validity of our findings.

7.5 Limitations and Future Work

We observe two limitations regarding the functionalities of *Cerebra*. First, the knowledge extracted from historical scripts in *Cerebra* is tailored to a single database schema and does not generalize well to other schemas. To address this, future work could explore transferring knowledge between datasets with semantically similar schemas. Second, *Cerebra* is currently implemented as a standalone web application, which does not fully align with SQL developers' workflows in desktop IDEs. This separation may introduce friction and limit *Cerebra*'s access to SQL artifacts typically managed within IDE environments and version control systems. Future work could integrate *Cerebra* as an IDE extension, potentially embedding the Script View within the editor panel while exposing the Knowledge and Data Views as lightweight, expandable side panels. This would provide a more seamless user experience and enable tighter coupling with developers' historical SQL contexts.

8 Conclusion

Data practitioners often provide natural language instructions involving implicit knowledge and struggle to validate whether NL-to-SQL tools have correctly inferred their requirements. In this paper, we present *Cerebra*, an interactive NL-to-SQL tool that aligns implicit knowledge between users and LLMs throughout SQL script generation and refinement. We began with a preliminary study to analyze the practices and challenges users encounter with existing NL-to-SQL tools, summarizing three key user requirements through interviews. Building on these findings, *Cerebra* automatically extracts and reuses implicit knowledge from users' historical scripts, provides a knowledge tree view to make the implicit knowledge explicit, and supports knowledge-level iterative query refinement. Our user study with 16 data practitioners demonstrates that *Cerebra* improves efficiency and helps users review the code more effectively. The notion of implicit knowledge and the knowledge reuse techniques can be generalized to other human-AI collaboration tasks. In the future, we plan to explore automatic confidence estimation and investigate cross-database knowledge transfer to further enhance the usability and generalizability of *Cerebra*.

Acknowledgments

The work was supported by Zhejiang Provincial Natural Science Foundation of China under Grant No. LD25F020003, NSFC (62402421, 62421003), and Ningbo Yongjiang Talent Programme (2024A-399-G). Dae Hyun Kim is partially supported by the grant 2025-22-0499 awarded by the Institute for AI and Social Innovation at Yonsei University. We sincerely appreciate the constructive feedback from the anonymous reviewers and all participants in our studies.

References

[1] Azza Abouzied, Joseph Hellerstein, and Avi Silberschatz. 2012. DataPlay: Interactive Tweaking and Example-Driven Correction of Graphical Database Queries. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 207–218. doi:10.1145/2380116.2380144

[2] A. Aiken, J. Chen, M. Stonebraker, and A. Woodruff. 1996. Tioga-2: A Direct Manipulation Database Visualization Environment. In *Proceedings of the International Conference on Data Engineering*. IEEE Press, 208–217. doi:10.1109/ICDE.1996.492109

[3] Ian Arawjo, Chelse Swoopes, Priyan Vaithilingam, Martin Wattenberg, and Elena L. Glassman. 2024. ChainForge: A Visual Toolkit for Prompt Engineering and LLM Hypothesis Testing. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*. ACM, Article 304, 18 pages. doi:10.1145/3613904.3642016

[4] Alexander Bendeck, Dennis Bromley, and Vidya Setlur. 2024. SlopeSeeker: A Search Tool for Exploring a Dataset of Quantifiable Trends. In *Proceedings of the International Conference on Intelligent User Interfaces*. ACM, 817–836. doi:10.1145/3640543.3645208

[5] Malik Olivier Boussejra, Rikuo Uchiki, Yuriko Takeshima, Kazuya Matsubayashi, Shunya Takekawa, Makoto Uemura, and Issei Fujishiro. 2019. aflak: Visual Programming Environment Enabling End-to-End Provenance Management for the Analysis of Astronomical Datasets. *Visual Informatics* 3, 1 (2019), 1–8. doi:10.1016/j.visinf.2019.03.001

[6] Xiwen Cai, Kai Xiong, Zhongsu Luo, Di Weng, Shuainan Ye, and Yingcai Wu. 2025. CodeLin: An In Situ Visualization Method for Understanding Data Transformation Scripts. *Visual Informatics* 9, 2 (2025), 100233. doi:10.1016/j.visinf.2025.03.002

[7] Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, Steve Ash, William Yang Wang, Zhiguo Wang, Vittorio Castelli, Patrick Ng, and Bing Xiang. 2023. Dr.Spider: A Diagnostic Evaluation Benchmark towards Text-to-SQL Robustness. In *Proceedings of the International Conference on Learning Representations*. https://openreview.net/forum?id=Wc5bmZZU9cy

[8] Chat2DB Team. 2024. Chat2DB. https://chat2db-ai.com/. Last accessed on 2025-09-06.

[9] Noptanit Chotisarn, Leonel Merino, Xu Zheng, Supaporn Lonapalawong, Tianye Zhang, Mingliang Xu, and Wei Chen. 2020. A Systematic Literature Review of Modern Software Visualization. *Journal of Visualization* 23, 4 (2020), 539–558. doi:10.1007/s12650-020-00647-w

[10] dagrejs. 2023. The Dagre Library. https://github.com/dagrejs/dagre/wiki. Last accessed on 2025-09-06.

[11] Ahmed Elgohary, Christopher Meek, Matthew Richardson, Adam Fourney, Gonzalo Ramos, and Ahmed Hassan Awadallah. 2021. NL-EDIT: Correcting Semantic Parse Errors through Natural Language Interaction. In *Proceedings of the North American Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, 5599–5610. doi:10.18653/v1/2021.naacl-main.444

[12] Will Epperson, Doris Jung-Lin Lee, Leijie Wang, Kunal Agarwal, Aditya G. Parameswaran, Dominik Moritz, and Adam Perer. 2022. Leveraging Analysis History for Improved In Situ Visualization Recommendation. *Computer Graphics Forum* 41, 3 (2022), 145–155. doi:10.1111/cgf.14529

[13] Will Epperson, April Yi Wang, Robert DeLine, and Steven M. Drucker. 2022. Strategies for Reuse and Sharing Among Data Scientists in Software Teams. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 243–252. doi:10.1145/3510457.3513042

[14] Yujian Gan, Xinyun Chen, and Matthew Purver. 2021. Exploring Underexplored Limitations of Cross-Domain Text-to-SQL Generalization. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 8926–8931. doi:10.18653/v1/2021.emnlp-main.702

[15] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proceedings of the VLDB Endowment* 17, 5 (2024), 1132–1145. doi:10.14778/3641204.3641221

[16] Tong Gao, Mira Dontcheva, Eytan Adar, Zhicheng Liu, and Karrie G. Karahalios. 2015. DataTone: Managing Ambiguity in Natural Language Interfaces for Data Visualization. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 489–500. doi:10.1145/2807442.2807478

[17] Sneha Gathani, Peter Lim, and Leilani Battle. 2020. Debugging Database Queries: A Survey of Tools, Techniques, and Users. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*. ACM, 1–16. doi:10.1145/3313831.3376485

[18] Torsten Grust, Fabian Kliebhan, Jan Rittinger, and Tom Schreiber. 2011. True Language-Level SQL Debugging. In *Proceedings of the International Conference on Extending Database Technology*. ACM, 562–565. doi:10.1145/1951365.1951441

[19] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation. In *Proceedings of Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 4524–4535. doi:10.18653/v1/P19-1444

[20] Izzeddin Gur, Semih Yavuz, Yu Su, and Xifeng Yan. 2018. DialSQL: Dialogue Based Structured Query Generation. In *Proceedings of Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 1339–1349. doi:10.18653/v1/P18-1124

[21] Sandra G. Hart and Lowell E. Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research. In *Human Mental Workload*. Advances in Psychology, Vol. 52. North-Holland, 139–183. doi:10.1016/S0166-4115(08)62386-9

[22] Enamul Hoque, Vidya Setlur, Melanie Tory, and Isaac Dykeman. 2018. Applying Pragmatics Principles for Interaction with Visual Analytics. *IEEE Transactions on Visualization and Computer Graphics* 24, 1 (2018), 309–318. doi:10.1109/TVCG.2017.2744684

[23] Ling Hu, Kenneth A. Ross, Yuan-Chi Chang, Christian A. Lang, and Donghui Zhang. 2008. QueryScope: Visualizing Queries for Repeatable Database Tuning. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1488–1491. doi:10.14778/1454159.1454209

[24] Zhipeng Hu, Changjie Fan, Qiwei Zheng, Wei Wu, and Bai Liu. 2021. Asyncflow: A Visual Programming Tool for Game Artificial Intelligence. *Visual Informatics* 5, 4 (2021), 20–25. doi:10.1016/j.visinf.2021.11.001

[25] Yanwei Huang, Yunfan Zhou, Ran Chen, Changhao Pan, Xinhuan Shu, Di Weng, and Yingcai Wu. 2024. Interactive Table Synthesis With Natural Language. *IEEE Transactions on Visualization and Computer Graphics* 30, 9 (2024), 6130–6145. doi:10.1109/TVCG.2023.3329120

[26] Zezhou Huang, Pavan Kalyan Damalapati, and Eugene Wu. 2023. Data Ambiguity Strikes Back: How Documentation Improves GPT's Text-to-SQL. In *NeurIPS Second Table Representation Learning Workshop*. https://openreview.net/forum?id=FflKTuIRTD

[27] Zeyuan Huang, Cangjun Gao, Yaxian Shan, Haoxiang Hu, Qingkun Li, Xiaoming Deng, Cuixia Ma, Yu-Kun Lai, Yong-Jin Liu, Feng Tian, Guozhong Dai, and Hongan Wang. 2025. SketchGPT: A Sketch-based Multimodal Interface for Application-Agnostic LLM Interaction. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, Article 157, 18 pages. doi:10.1145/3746059.3747598

[28] Zhaosong Huang, Ye Zhao, Wei Chen, Shengjie Gao, Kejie Yu, Weixia Xu, Mingjie Tang, Minfeng Zhu, and Mingliang Xu. 2020. A Natural-language-based Visual Query Approach of Uncertain Human Trajectories. *IEEE Transactions on Visualization and Computer Graphics* 26, 1 (2020), 1256–1266. doi:10.1109/TVCG.2019.2934671

[29] Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, and Ed Lazowska. 2016. SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment. In *Proceedings of the International Conference on Management of Data*. ACM, 281–293. doi:10.1145/2882903.2882957

[30] JetBrains. 2025. DataGrip: The Cross-Platform IDE for Databases & SQL. https://www.jetbrains.com/datagrip/. Last accessed on 2025-09-06.

[31] George Katsogiannis-Meimarakis and Georgia Koutrika. 2023. A Survey on Deep Learning Approaches for Text-to-SQL. *The VLDB Journal* 32, 4 (2023), 905–936. doi:10.1007/s00778-022-00776-8

[32] Majeed Kazemitabaar, Jack Williams, Ian Drosos, Tovi Grossman, Austin Zachary Henley, Carina Negreanu, and Advait Sarkar. 2024. Improving Steering and Verification in AI-Assisted Data Analysis with Interactive Task Decomposition. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, Article 92, 19 pages. doi:10.1145/3654777.3676345

[33] Meraj Khan, Larry Xu, Arnab Nandi, and Joseph M. Hellerstein. 2017. Data Tweening: Incremental Visualization of Data Transforms. *Proceedings of the VLDB Endowment* 10, 6 (2017), 661–672. doi:10.14778/3055330.3055333

[34] Tae Soo Kim, Yoonjoo Lee, Jamin Shin, Young-Ho Kim, and Juho Kim. 2024. EvalLM: Interactive Evaluation of Large Language Model Prompts on User-Defined Criteria. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*. ACM, Article 306, 21 pages. doi:10.1145/3613904.3642216

[35] Andreas Kokkalis, Panagiotis Vagenas, Alexandros Zervakis, Alkis Simitsis, Georgia Koutrika, and Yannis Ioannidis. 2012. Logos: A System for Translating Queries into Narratives. In *Proceedings of the International Conference on Management of Data*. ACM, 673–676. doi:10.1145/2213836.2213929

[36] Georgia Koutrika, Alkis Simitsis, and Yannis E. Ioannidis. 2010. Explaining Structured Queries in Natural Language. In *Proceedings of the International Conference on Data Engineering*. IEEE Press, 333–344. doi:10.1109/ICDE.2010.5447824

[37] Klaus Krippendorff. 2018. *Content Analysis: An Introduction to Its Methodology*. SAGE Publications.

[38] Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. 2021. KaggleDBQA: Realistic Evaluation of Text-to-SQL Parsers. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing*. Association for Computational Linguistics, 2261–2273. doi:10.18653/v1/2021.acl-long.176

[39] 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. In *Proceedings of the International Conference on Learning Representations*. https://openreview.net/forum?id=XmProj9cPs

[40] Aristotelis Leventidis, Jiahui Zhang, Cody Dunne, Wolfgang Gatterbauer, H.V. Jagadish, and Mirek Riedewald. 2020. QueryVis: Logic-Based Diagrams Help Users Understand Complicated SQL Queries Faster. In *Proceedings of the International Conference on Management of Data*. ACM, 2303–2318. doi:10.1145/3318464.3389767

[41] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The Dawn of Natural Language to SQL: Are We Fully Ready? *Proceedings of the VLDB Endowment* 17, 11 (2024), 3318–3331. doi:10.14778/3681954.3682003

[42] Fei Li and Hosagrahar V Jagadish. 2014. NaLIR: An Interactive Natural Language Interface for Querying Relational Databases. In *Proceedings of the International Conference on Management of Data*. ACM, 709–712. doi:10.1145/2588555.2594519

[43] Hao Li, Chee-Yong Chan, and David Maier. 2015. Query from Examples: An Iterative, Data-Driven Approach to Query Construction. *Proceedings of the VLDB Endowment* 8, 13 (2015), 2158–2169. doi:10.14778/2831360.2831369

[44] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2023. RESDSQL: Decoupling Schema Linking and Skeleton Parsing for Text-to-SQL. In *Proceedings of the Annual AAAI Conference on Artificial Intelligence*, Vol. 37. AAAI, 13067–13075. doi:10.1609/aaai.v37i11.26535

[45] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. CodeS: Towards Building Open-source Language Models for Text-to-SQL. *Proceedings of the ACM on Management of Data* 2, 3, Article 127 (2024), 28 pages. doi:10.1145/3654930

[46] Jinyang Li, Binyuan Hui, Reynold Cheng, Bowen Qin, Chenhao Ma, Nan Huo, Fei Huang, Wenyu Du, Luo Si, and Yongbin Li. 2023. Graphix-T5: Mixing Pre-Trained Transformers with Graph-Aware Layers for Text-to-SQL Parsing. In *Proceedings of the Annual AAAI Conference on Artificial Intelligence*. AAAI Press, Article 1467, 9 pages. doi:10.1609/aaai.v37i11.26536

[47] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Ma Chenhao, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. In *Advances in Neural Information Processing Systems*, Vol. 36. Curran Associates, Inc., 42330–42357. https://openreview.net/forum?id=dI4wzAE6uV

[48] Xingjun Li, Yizhi Zhang, Justin Leung, Chengnian Sun, and Jian Zhao. 2023. EDAssistant: Supporting Exploratory Data Analysis in Computational Notebooks with In Situ Code Search and Recommendation. *ACM Transactions on Interactive Intelligent Systems* 13, 1, Article 1 (2023), 27 pages. doi:10.1145/3545995

[49] Yuntao Li, Bei Chen, Qian Liu, Yan Gao, Jian-Guang Lou, Yan Zhang, and Dongmei Zhang. 2020. What Do You Mean by That? A Parser-Independent Interactive Approach for Enhancing Text-to-SQL. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 6913–6922. doi:10.18653/v1/2020.emnlp-main.561

[50] Mario Linares-Vásquez, Boyang Li, Christopher Vendome, and Denys Poshyvanyk. 2015. How do Developers Document Database Usages in Source Code?. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE Press, 36–41. doi:10.1109/ASE.2015.67

[51] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2025. A Survey of Text-to-SQL in the Era of LLMs: Where are We, and Where are We Going? *IEEE Transactions on Knowledge and Data Engineering* 37, 10 (2025), 5735–5754. doi:10.1109/TKDE.2025.3592032

[52] Chandra Maddila, Negar Ghorbani, Kosay Jabre, Vijayaraghavan Murali, Edwin Kim, Parth Thakkar, Nikolay Pavlovich Laptev, Olivia Harman, Diana Hsu, Rui Abreu, and Peter C Rigby. 2025. AI-Assisted SQL Authoring at Industry Scale. In *Proceedings of International Conference on Software Engineering: Software Engineering in Practice*. IEEE Press, 148–158. doi:10.1109/ICSE-SEIP66354.2025.00019

[53] Nicolai Marquardt, Asta Roseway, Hugo Romat, Payod Panda, Michel Pahud, Gonzalo Ramos, Steven M. Drucker, Andrew D. Wilson, Ken Hinckley, and Nathalie Riche. 2025. ImaginationVellum: Generative-AI Ideation Canvas with Spatial Prompts, Generative Strokes, and Ideation History. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, Article 159, 19 pages. doi:10.1145/3746059.3747631

[54] Zhengjie Miao, Tiangang Chen, Alexander Bendeck, Kevin Day, Sudeepa Roy, and Jun Yang. 2020. I-Rex: An Interactive Relational Query Explainer for SQL. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2997–3000. doi:10.14778/

3415478.3415528
[55] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. RATest: Explaining Wrong Relational Queries Using Small Examples. In *Proceedings of the International Conference on Management of Data*. ACM, 1961–1964. doi:10.1145/3299869.3320236
[56] Microsoft. 2025. Monaco - The Editor of the Web. https://microsoft.github.io/monaco-editor/. Last accessed on 2025-09-06.
[57] Daphne Miedema and George Fletcher. 2021. SQLVis: Visual Query Representations for Supporting SQL Learners. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Press, 1–9. doi:10.1109/VL/HCC51201.2021.9576431
[58] Eckart Modrow. 2014. SQLsnap. https://snapextensions.uni-goettingen.de/sqlsnapmanual.pdf. Last accessed on 2025-09-06.
[59] Arpit Narechania, Adam Fourney, Bongshin Lee, and Gonzalo Ramos. 2021. DIY: Assessing the Correctness of Natural Language to SQL Systems. In *Proceedings of the International Conference on Intelligent User Interfaces*. ACM, 597–607. doi:10.1145/3397481.3450667
[60] Arpit Narechania, Arjun Srinivasan, and John Stasko. 2021. NL4DV: A Toolkit for Generating Analytic Specifications for Data Visualization from Natural Language Queries. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2021), 369–379. doi:10.1109/TVCG.2020.3030378
[61] Zheng Ning, Zheng Zhang, Tianyi Sun, Yuan Tian, Tianyi Zhang, and Toby Jia-Jun Li. 2023. An Empirical Study of Model Errors and User Error Discovery and Repair Strategies in Natural Language Database Queries. In *Proceedings of the International Conference on Intelligent User Interfaces*. ACM, 633–649. doi:10.1145/3581641.3584067
[62] Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In *Advances in Neural Information Processing Systems*, Vol. 36. Curran Associates, Inc., 36339–36348. https://openreview.net/forum?id=p53QDxSIc5
[63] PremiumSoft CyberTech Ltd. 2025. Navicat. https://www.navicat.com/. Last accessed on 2025-09-06.
[64] Chunyao Qian, Shizhao Sun, Weiwei Cui, Jian-Guang Lou, Haidong Zhang, and Dongmei Zhang. 2021. Retrieve-Then-Adapt: Example-Based Automatic Generation for Proportion-Related Infographics. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (2021), 443–452. doi:10.1109/TVCG.2020.3030448
[65] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 3982–3992. doi:10.18653/v1/D19-1410
[66] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R. Mittal, and Fatma Özcan. 2016. ATHENA: An Ontology-Driven System for Natural Language Querying over Relational Data Stores. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1209–1220. doi:10.14778/2994509.2994536
[67] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 9895–9901. https://aclanthology.org/2021.emnlp-main.779
[68] Vidya Setlur, Sarah E. Battersby, Melanie Tory, Rich Gossweiler, and Angel X. Chang. 2016. Eviza: A Natural Language Interface for Visual Analysis. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 365–377. doi:10.1145/2984511.2984588
[69] Vidya Setlur, Enamul Hoque, Dae Hyun Kim, and Angel X. Chang. 2020. Sneak Pique: Exploring Autocompletion as a Data Discovery Scaffold for Supporting Visual Analysis. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 966–978. doi:10.1145/3379337.3415813
[70] Shreya Shankar, J.D. Zamfirescu-Pereira, Bjoern Hartmann, Aditya Parameswaran, and Ian Arawjo. 2024. Who Validates the Validators? Aligning LLM-Assisted Evaluation of LLM Outputs with Human Preferences. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, Article 131, 14 pages. doi:10.1145/3654777.3676450
[71] Tarique Siddiqui, Paul Luh, Zesheng Wang, Karrie Karahalios, and Aditya Parameswaran. 2020. ShapeSearch: A Flexible and Efficient System for Shape-based Exploration of Trendlines. In *Proceedings of the International Conference on Management of Data*. ACM, 51–65. doi:10.1145/3318464.3389722
[72] SQLite Development Team. 2025. SQLite. https://www.sqlite.org/index.html. Last accessed on 2025-09-06.
[73] Arjun Srinivasan and John Stasko. 2018. Orko: Facilitating Multimodal Interaction for Visual Exploration and Analysis of Networks. *IEEE Transactions on Visualization and Computer Graphics* 24, 1 (2018), 511–521. doi:10.1109/TVCG.2017.2745219
[74] C. Stolte, D. Tang, and P. Hanrahan. 2002. Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases. *IEEE Transactions on Visualization and Computer Graphics* 8, 1 (2002), 52–65. doi:10.1109/2945.981851
[75] Desheng Sun, Xiaoqi Yue, Chao Liu, Hongxing Qin, and Haibo Hu. 2024. SFLVis: Visual Analysis of Software Fault Localization. *Journal of Visualization* 27, 4 (2024), 585–602. doi:10.1007/s12650-024-00979-x
[76] Yuan Tian, Jonathan K. Kummerfeld, Toby Jia-Jun Li, and Tianyi Zhang. 2024. SQLucid: Grounding Natural Language Database Queries with Interactive Explanations. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, Article 12, 20 pages. doi:10.1145/3654777.3676368
[77] Yuan Tian, Chuhan Zhang, Xiaotong Wang, Sitong Pan, Weiwei Cui, Haidong Zhang, Dazhen Deng, and Yingcai Wu. 2025. ReSpark: Leveraging Previous Data Reports as References to Generate New Reports with LLMs. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, Article 181, 18 pages. doi:10.1145/3746059.3747644
[78] Yuan Tian, Zheng Zhang, Zheng Ning, Toby Jia-Jun Li, Jonathan K. Kummerfeld, and Tianyi Zhang. 2023. Interactive Text-to-SQL Generation via Editable Step-by-Step Explanations. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 16149–16166. doi:10.18653/v1/2023.emnlp-main.1004
[79] Toby Mao. 2025. SQLGlot. https://sqlglot.com/sqlglot.html. Last accessed on 2025-09-06.
[80] April Yi Wang, Dakuo Wang, Jaimie Drozdal, Michael Muller, Soya Park, Justin D. Weisz, Xuye Liu, Lingfei Wu, and Casey Dugan. 2022. Documentation Matters: Human-Centered AI System to Assist Data Science Code Documentation in Computational Notebooks. *ACM Transactions on Computer-Human Interaction* 29, 2, Article 17 (2022), 33 pages. doi:10.1145/3489465
[81] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In *Proceedings of Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 7567–7578. doi:10.18653/v1/2020.acl-main.677
[82] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 452–466. doi:10.1145/3062341.3062365
[83] Huichen Will Wang, Larry Birnbaum, and Vidya Setlur. 2025. Jupybara: Operationalizing a Design Space for Actionable Data Analysis and Storytelling with LLMs. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*. ACM, Article 1005, 24 pages. doi:10.1145/3706598.3713913
[84] Liwenhan Xie, Chengbo Zheng, Haijun Xia, Huamin Qu, and Chen Zhu-Tian. 2024. WaitGPT: Monitoring and Steering Conversational LLM Agent in Data Analysis with On-the-Fly Code Visualization. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, Article 119, 14 pages. doi:10.1145/3654777.3676374
[85] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 63 (2017), 26 pages. doi:10.1145/3133887
[86] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. https://arxiv.org/abs/2505.09388
[87] Ziyu Yao, Yu Su, Huan Sun, and Wen-tau Yih. 2019. Model-based Interactive Semantic Parsing: A Unified Framework and A Text-to-SQL Case Study. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing*. Association for Computational Linguistics, 5447–5458. doi:10.18653/v1/D19-1547
[88] Catherine Yeh, Donghao Ren, Yannick Assogba, Dominik Moritz, and Fred Hohman. 2025. Exploring Empty Spaces: Human-in-the-Loop Data Augmentation. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*. ACM, Article 847, 19 pages. doi:10.1145/3706598.3713491
[89] Ryan Yen, Jian Zhao, and Daniel Vogel. 2025. Code Shaping: Iterative Code Editing with Free-form AI-Interpreted Sketching. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*. ACM, Article 872, 17 pages. doi:10.1145/3706598.3713822
[90] Ryan Yen, Jiawen Stefanie Zhu, Sangho Suh, Haijun Xia, and Jian Zhao. 2024. CoLadder: Manipulating Code Generation via Multi-Level Blocks. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, Article 11, 20 pages. doi:10.1145/3654777.3676357
[91] Lu Ying, Xinhuan Shu, Dazhen Deng, Yuchen Yang, Tan Tang, Lingyun Yu, and Yingcai Wu. 2023. MetaGlyph: Automatic Generation of Metaphoric Glyph-based Visualization. *IEEE Transactions on Visualization and Computer Graphics* 29, 1 (2023), 331–341. doi:10.1109/TVCG.2022.3209447
[92] Tao Yu, Michihiro Yasunaga, Kai Yang, Rui Zhang, Dongxu Wang, Zifan Li, and Dragomir Radev. 2018. SyntaxSQLNet: Syntax Tree Networks for Complex and Cross-Domain Text-to-SQL Task. In *Proceedings of the Conference on Empirical*

Methods in Natural Language Processing. Association for Computational Linguistics, 1653–1663. doi:10.18653/v1/D18-1193

[93] 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 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 3911–3921. doi:10.18653/v1/D18-1425

[94] Sai Zhang and Yuyin Sun. 2013. Automatically Synthesizing SQL Queries from Input-Output Examples. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE Press, 224–234. doi:10.1109/ASE.2013.6693082

[95] Wenshuo Zhang, Leixian Shen, Shuchang Xu, Jindu Wang, Jian Zhao, Huamin Qu, and Lin-Ping Yuan. 2025. NeuroSync: Intent-Aware Code-Based Problem Solving via Direct LLM Understanding Modification. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, Article 30, 19 pages. doi:10.1145/3746059.3747668

[96] Fuheng Zhao, Shaleen Deep, Fotis Psallidas, Avrilia Floratou, Divyakant Agrawal, and Amr El Abbadi. 2025. Sphinteract: Resolving Ambiguities in NL2SQL through User Interaction. *Proceedings of the VLDB Endowment* 18, 4 (2025), 1145–1158. doi:10.14778/3717755.3717772

[97] Yunfan Zhou, Xiwen Cai, Qiming Shi, Yanwei Huang, Haotian Li, Huamin Qu, Di Weng, and Yingcai Wu. 2025. Xavier: Toward Better Coding Assistance in Authoring Tabular Data Wrangling Scripts. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*. ACM, Article 850, 16 pages. doi:10.1145/3706598.3714239

[98] Jiajun Zhu, Xinyu Cheng, Zhongsu Luo, Yunfan Zhou, Xinhuan Shu, Di Weng, and Yingcai Wu. 2025. ViseGPT: Towards Better Alignment of LLM-generated Data Wrangling Scripts and User Prompts. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, Article 147, 16 pages. doi:10.1145/3746059.3747689

A Demographic Information of participants

Table 4: Demographic Information of Participants in Preliminary Study.

User	Gender	Age	Background	Experience (Year)	Familiarity (1–7)	Frequency
P1	Male	26	Enterprise Resource Planning	7	7	Once a week
P2	Male	41	Natural Resources	10	2	Everyday
P3	Male	41	Geographic Information System	20	6	Once a week
P4	Male	28	Stock Market Analysis	3	3	Everyday
P5	Female	34	Education	10	4	Three times a week
P6	Male	32	Natural Resources	9	5	Everyday
P7	Male	39	Geospatial Data Query	20	7	Everyday
P8	Male	40	Land Affairs	17	3	Everyday
P9	Male	37	Geographic Information System	5	4	Once a week
P10	Male	38	Communications	14	6	Everyday

Table 5: Demographic Information of Participants in User Study.

User	Gender	Age	Background	Experience (Year)	Familiarity (1–7)	Frequency
U1	Female	26	Cloud Computing	3	7	Everyday
U2	Male	21	E-commerce Management Systems	5	7	Once a week
U3	Female	22	Gaming, Biology	1	5	Once a week
U4	Male	27	Stock Database Query	5	7	Once a week
U5	Male	24	Chemical Engineering	1	3	Once a week
U6	Male	37	Surveying	5	4	Once a week
U7	Male	26	Knowledge Graph Query	5	7	Once a week
U8	Female	20	Pharmaceutical Retail	1	5	Three times a week
U9	Male	40	Office Automation Data	17	3	Everyday
U10	Male	26	E-commerce	7	7	Once a week
U11	Male	20	Geoinformation Science	1	4	Once a week
U12	Female	23	Banking	1	3	Three times a week
U13	Female	26	Internet Backend Development	5	5	Everyday
U14	Male	23	Bike Sharing	2	5	Everyday
U15	Male	28	Financial Data Verification	3	3	Everyday
U16	Female	23	Application Commercialization	1	2	Three times a week

B Quantitative Results of User Study

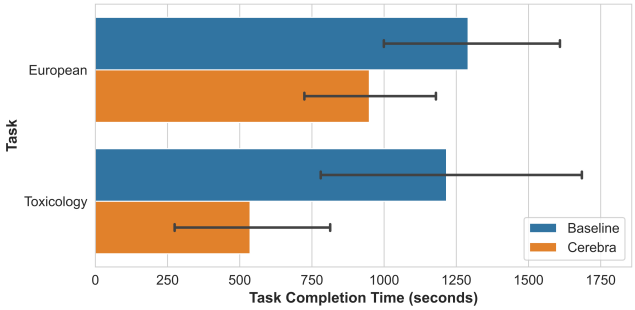

Figure 8: Task completion time for Baseline and *Cerebra*.

To evaluate whether *Cerebra* accelerated the script authoring process, we measured "task completion time" of each participant in each task. Shapiro-Wilk and Levene's tests confirmed that the raw data followed a normal distribution with equal variance. Consequently, we performed a two-way ANOVA to test the main effects of tools (*Cerebra* vs. Baseline) and tasks (European vs. Toxicology), as well as their interaction effects. We found that the main effects of tools were significant ($p < 0.01$), while the interaction effects and the main effects of tasks were not significant. According to the result of task completion time (Figure 8), participants spent less time completing the tasks with *Cerebra* than with Baseline, reflecting the overall advantage indicated by the ANOVA.

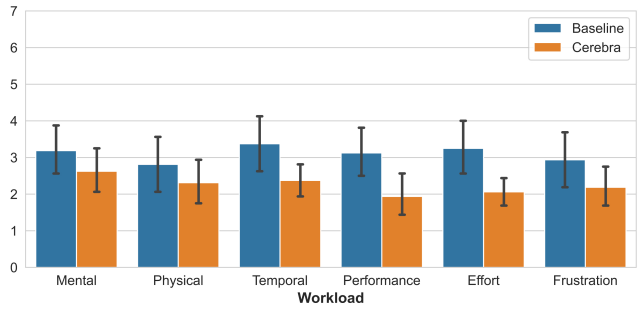

Figure 9: NASA-TLX workload for Baseline and *Cerebra*.

For perceived workload, we analyzed the NASA-TLX data, as shown in Figure 9. Normality tests indicated that only the Mental Demand dimension met the assumptions for a standard ANOVA. For the remaining dimensions (Physical Demand, Temporal Demand, Performance, Effort, and Frustration), we utilized the non-parametric Aligned Rank Transform (ART) ANOVA. The analysis revealed significant main effects of tools on the Performance ($p < 0.01$) and Effort ($p < 0.05$) workload scores. Post-hoc EM-Means comparisons confirmed that participants reported significantly lower Performance and Effort workload when using *Cerebra* than when using Baseline. All other NASA-TLX dimensions (Mental, Physical, Temporal, and Frustration) showed no significant effect of tools. Similar to the time analysis, neither the main effect of tasks nor the interaction between tools and tasks was significant across any dimension.