Bridging NL2SQL for CAN Signal Analytics at NIO: From Externalized Schemas to CTE Pipelines

Wei Zhou*
Shanghai Jiao Tong Univ.
weizhoudb@sjtu.edu.cn

Haoyu Zhao*
Shanghai Jiao Tong Univ.
parrnell@sjtu.edu.cn

Ruiguang Zhong
NIO
taurus.zhong@nio.com

Shaoli Yu
Shanghai Jiao Tong Univ.
geniustanker@sjtu.edu.cn

Xuanhe Zhou
Shanghai Jiao Tong Univ.
zhouxuanhe@sjtu.edu.cn

Xue Yang
Shanghai Jiao Tong Univ.
yangxue-2019-sjtu@sjtu.edu.cn

Xiaosong Jia
Fudan University
jiaxiaosong1997@gmail.com

Guoliang Li
Tsinghua University
liguoliang@tsinghua.edu.cn

Shengzhong Liu
Shanghai Jiao Tong Univ.
sl29@illinois.edu

Kang Zhang
China Telecom Corporation Ltd. Shanghai Branch
zhangkang.sh@chinatelecom.cn

Jie Wu
NIO
kenn.wu@nio.com

Fan Wu
Shanghai Jiao Tong Univ.
fwu@cs.sjtu.edu.cn

ABSTRACT

Large Language Models (LLMs) promise natural-language access to data, yet applying them to Controller Area Network (CAN) signal SQL analytics in the automotive industry is challenging. First, CAN data warehouses store tens of thousands of signals as opaque numeric columns. Multiple signals are often packed into a single column as bit segments, requiring external Database CAN (DBC) files beyond the native database schema for proper definitions. Second, engineers' natural-language queries often use terms that map to multiple signals in the DBC files, with little direct indication of which signals are being referenced. Finally, signal analysis in production involves multi-step SQL pipelines using layered Common Table Expressions (CTEs) for tasks such as temporal alignment, more complex than those handled by existing NL2SQL methods.

To address these, we present `NIO-Weaver`, an end-to-end CAN signal analytics system within a graph-augmented multi-agent framework. First, to uncover signals hidden in opaque numeric columns, we introduce an automated graph-construction pipeline that leverages DBC files to build a domain-knowledge graph, associating signals with potential high-level terms. Second, to bridge the semantic gap between natural-language queries and multiple signals, we propose an optimized retrieval method that extracts terms from queries and performs hybrid retrieval. We combine regex matching, embedding search, and constrained graph traversal to identify candidate signals for further processing. Finally, to simplify the generation of multi-step CTE-based pipelines, we introduce a hierarchical generation paradigm. We decompose queries into plans, incrementally generating executable CTEs, while self-correction mechanisms refine missing signals and incorrect SQL. Experiments on the real-world industrial CAN warehouse at NIO demonstrate that `NIO-Weaver` outperforms baselines by up to 67.2%.

PVLDB Reference Format:
Wei Zhou, Haoyu Zhao, Ruiguang Zhong, Shaoli Yu, Xuanhe Zhou, Xue Yang, Xiaosong Jia, Guoliang Li, Shengzhong Liu, Kang Zhang, Jie Wu, and Fan Wu. Bridging NL2SQL for CAN Signal Analytics at NIO:

*Equal Contribution.

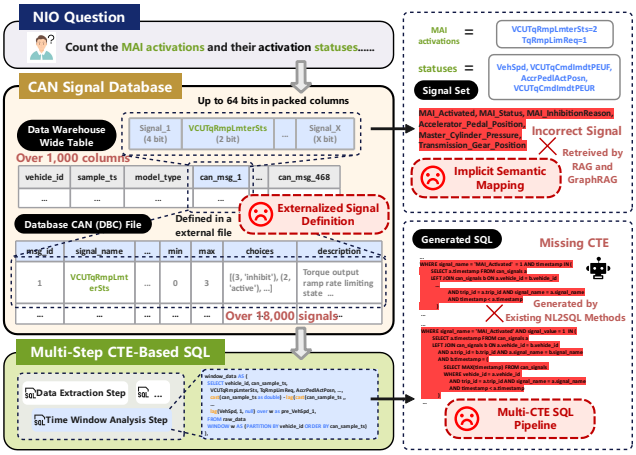

Figure 1: NL2SQL in Automotive CAN Signal Analytics.

From Externalized Schemas to CTE Pipelines. PVLDB, 19(12): 4330 - 4343, 2026.
doi:10.14778/3827998.3828036

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/OpenDataBox/NIO-Weaver.

1 INTRODUCTION

NL2SQL techniques bridge the gap between natural language and databases [31, 42, 45, 51], enabling users to query data without writing complex SQL queries. These techniques are highly valuable in specialized fields where experts must quickly analyze massive and complex datasets. In automotive engineering, where modern

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828036

vehicles function as advanced IoT platforms, vast amounts of signal data are continuously transmitted via the Controller Area Network (CAN) bus [12, 33] into large-scale, time-partitioned databases. Analyzing these signals using SQL queries, directly from expert intent, is vital for driving data-informed decisions throughout the vehicle's lifecycle [13, 27]. It plays a key role in supporting experts in vehicle development, feature validation, and intelligent diagnosis [20, 47, 48, 52, 54].

For instance, during the research and development phase, experts analyze Mis-Acceleration Intervention (MAI) events, a safety system that prevents accidental acceleration by monitoring the vehicle and either reducing engine power or applying the brakes. To ensure this system works correctly, they must review vehicle data, such as speed, accelerator pedal position, and torque commands. As shown in Figure 1, the CAN signal analysis relies on large, time-partitioned databases (e.g., Hive at NIO) to handle fast, real-time IoT data [9, 35]. It introduces three unique characteristics compared with typical NL2SQL techniques.

❶ **Self-Contained Schema *vs.* Externalized Signal Definition.** While conventional databases directly embed metadata in table and column definitions, modern CAN data warehouses exhibit extreme schema opacity. They pack tens of thousands of logical signals into generic payload columns with little semantic meaning. For example, as shown in Figure 1, a single column like `can_msg_1` in the warehouse table contains up to 64 bits, multiplexing distinct elements such as a 2-bit `VCUTqRmpLmterSts`. The critical metadata needed to interpret these signals, such as message IDs, descriptions, and specific value choices (e.g., 2 representing active), is externalized in DBC files. Consequently, NL2SQL models must resolve schema entities entirely outside the database catalog [28].

❷ **Structural Schema Linking *vs.* Implicit Semantic Mapping.** Unlike traditional NL2SQL methods that rely on direct semantic matching or schema relations, CAN signal analytics presents a much wider semantic gap between user queries and SQL queries. Experts typically query high-level vehicle behaviors rather than exact, often-abbreviated signal names, meaning a single term frequently maps to complex, multi-signal predicates. For instance, in Figure 1, when asked to "Count the MAI activations," standard RAG methods incorrectly retrieve name-similar signals such as `MAI_Activated` from over 18,000 options. In reality, this state requires a specific combination of exact values: `VCUTqRmpLmterSts=2` and `TqRmpLimReq=1`. This highlights the critical need for deep semantic grounding beyond basic structural schema linking.

❸ **Relational Query Generation *vs.* Multi-CTE SQL Pipeline.** Traditional NL2SQL benchmarks rigorously evaluate complex relational logic, including nested subqueries, advanced joins, and set operations [26, 41]. However, real-world CAN analytics require a different query complexity of time-series execution pipelines. These workloads rely heavily on layered Common Table Expressions (CTEs) for data extraction, temporal alignment, and time-window analysis. For instance, in Figure 1, the gold SQL for an MAI event requires a `window_data` CTE utilizing complex functions like `LAG(...) OVER` to calculate temporal gaps (`ts_gap_1`) and capture previous vehicle speeds (`pre_VehSpd_1`). Current LLM-based NL2SQL methods struggle with this complexity, generating simplistic, non-executable statements (e.g., `WHERE signal_name = 'MAI_Activated'`) that lack the necessary data-processing CTEs.

While recent LLM-based NL2SQL and RAG frameworks [7, 17, 38] offer compelling solutions for natural-language data access, they primarily target self-contained schemas with lexical or semantic overlap, making direct adaptation to CAN analytics difficult. Specifically, general NL2SQL methods [19, 32] are not natively equipped to resolve external payload definitions or reliably synthesize complex, multi-step temporal pipelines. Furthermore, general-purpose GraphRAG [4, 55] mainly processes raw unstructured text, often struggling to traverse the constrained, typed structures of DBC files to deduce implicit multi-signal states.

Our Methodology. To implement an effective NL2SQL framework for CAN analytics, we propose `NIO-Weaver`, a graph-augmented multi-agent system that decouples intent retrieval from SQL generation via three core components. ***(1) Semantic-Aware Graph Builder***: To overcome opaque schemas, the *Graph Builder* leverages DBC files with low-level metadata, including message IDs and bit layouts to construct a grounded knowledge graph. It extracts physical entities and links high-level analytical intents directly to low-level signal metadata, effectively bridging the semantic gap; ***(2) Constraint-Guided Signal Retriever***: To address implicit semantic grounding across over 18,000 entries, the *Signal Retriever* adopts an intent-aware approach to extract query entities and employs a hybrid, constraint-guided graph traversal to precisely isolate relevant signals, avoiding the noise of traditional methods; ***(3) Pipeline-Planned SQL Writer***: To alleviate the difficulty of complex structures synthesis, the *SQL Writer* constructs multi-step CAN analytics pipelines via sequential CTE operations. It generates SQL queries following a step-by-step plan, and subsequently refines them through adaptive signal augmentation and hybrid execution-semantic validation.

Product Impact. `NIO-Weaver` *is deployed at the leading intelligent EV manufacturer (NIO) for signal data analytics across the entire vehicle lifecycle. It is integrated across 3 core technology platforms and over 12 vehicle models, covering all 5 major operational stages (i.e., R&D, production, supply chain, sales, and service). In the R&D phase, the system processes an average of 33-44 daily NL2SQL queries per model, enabling engineers to query a data platform of over 30 PB, comprising 10,000+ tables and tens of thousands of vehicle signals.*

Contributions. We summarize our key contributions as follows:
(1) We propose `NIO-Weaver`, an end-to-end CAN signal analytics system built on a graph-augmented multi-agent framework. *To the best of our knowledge, this is the first work to integrate a DBC-grounded knowledge graph into the NL2SQL pipeline to handle the unique complexities of CAN signal analytics* (Section 3).
(2) We propose a graph builder that constructs a DBC-grounded knowledge base. By extracting and standardizing key metadata entities, it bridges the wide gap between high-level analytical intent and raw payload columns (Section 4).
(3) We design a signal retriever that replaces basic, noisy graph expansion with an intent-aware retrieval. By analyzing user queries and limiting retrieval paths via constraints, it accurately maps user concepts to the correct signals in a graph of over 18,000 nodes without pulling in unrelated data (Section 5).

(4) We develop a SQL writer that produces executable, multi-step SQL queries step-by-step. It first outlines a clear plan of multiple CTE steps, such as decoding, and refines them through adaptive augmentation and lightweight validation (Section 6).
(5) We evaluate `NIO-Weaver` on a real-world industrial CAN dataset at NIO. The experimental results demonstrate consistent improvements over standard LLM and RAG baselines, showing significant gains in both signal retrieval quality and final execution accuracy (up to 67.2%) for complex diagnosis queries (Section 7).

2 PRELIMINARIES

2.1 CAN Signal Analytics

During vehicle development and maintenance, automotive engineers and data scientists routinely analyze massive volumes of telemetry data to diagnose faults, validate system performance, and optimize control strategies. These complex analytical queries are executed on historical vehicle data derived from CAN signals in cloud-based data warehouses (e.g., Hive [36]) to enable large-scale processing [21]. Therefore, CAN signal SQL analytics has emerged as a crucial capability, automating SQL generation to retrieve and process signal data directly from natural-language queries. However, unlike traditional NL2SQL tasks that assume self-contained databases with descriptive schemas and explicit relational constraints, CAN analytics operates in decoupled, opaque environments, challenging standard NL2SQL paradigms.

Definition 2.1 (CAN Signal Analytics Environment)**.** The CAN signal analytics environment is formalized as a tuple $(\mathcal{T}, \mathcal{D})$. Here, $\mathcal{T}$ represents the wide table in the data warehouse storing raw, semantically opaque CAN message columns (e.g., `can_msg_*`), which contain the physical signal values densely packed and divided into bits. $\mathcal{D}$ denotes the external specifications (e.g., DBC files) that maintain the semantic mappings for all automotive signals, including their owning message IDs, bit layouts to decode data from the raw packed columns, and textual descriptions.

As illustrated in Figure 1, $\mathcal{T}$ contains generic columns like `can_msg_1`, which stores 64 packed bits of raw data. The external specification $\mathcal{D}$ provides the missing semantic mapping, dictating that a specific 2-bit signal `VCUTqRmpLmterSts` (representing "the Torque output ramp rate limiting state") is located within message ID 1. Since the database catalog lacks semantic context, answering a query inherently requires a two-stage pipeline: first recovering the essential signal specifications from the external knowledge base $\mathcal{D}$, and subsequently generating SQL that uses the user-defined decoding utility to materialize and process the signal values.

Definition 2.2 (Two-Stage CAN NL2SQL)**.** Given a natural-language query Q, we need to process this analytical request through a mandatory two-stage pipeline: (1) It first executes *Signal Retrieval*, formalized as $\mathcal{R} : (Q, \mathcal{D}) \mapsto M_Q$, which reasons exclusively over the external specification $\mathcal{D}$ to extract the required signal set M_Q; (2) It subsequently performs *SQL Generation*, formalized as $\mathcal{F} : (Q, M_Q, \mathcal{T}) \mapsto \mathcal{S}$, synthesizing an executable, multi-CTE SQL pipeline $\mathcal{S}$ that incorporates the decoding utilities to process the opaque columns in $\mathcal{T}$ and yields the domain-consistent result (e.g., valid signal values aligned with the expert definitions in $\mathcal{D}$).

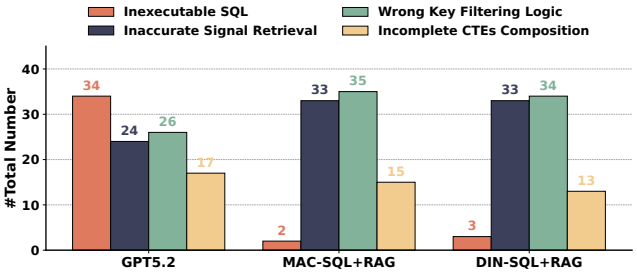

Figure 2: Error Distribution of Existing Methods.

As shown in Figure 1, to answer the query Q "Count the MAI activations and their activation statuses", it involves two stages: (1) The retrieval function $\mathcal{R}$ first bridges the semantic gap by reasoning over $\mathcal{D}$ to map the high-level analytical intent "MAI activations" to a multi-signal state predicate (e.g., `VCUTqRmpLmterSts = 2` and `TqRmpLimReq = 1`), alongside relevant signals (e.g., `VehSpd`) to form M_Q; (2) The generation function $\mathcal{F}$ uses M_Q to synthesize $\mathcal{S}$ as a multi-step CTE-based SQL pipeline. It begins by extracting raw data and decoding columns via utilities (e.g., `can_decode(can_msg_86, ..., VCUTqRmpLmterSts)`). After time alignment, it performs window analysis using operators such as `LAG` to compute time gaps and extract previous-row values for state-transition detection. An event is flagged only when current conditions are met, but previous conditions (e.g., `pre_VCUTqRmpLmterSts != 2`) were not, culminating in final aggregations for the expected domain-consistent result.

As discussed in Section 1, the difficulties arise because *(1) Externalized Signal Definitions:* Unlike self-contained databases with explicit relational constraints, the candidate signal space is defined by external specification $\mathcal{D}$ (e.g., DBC files). With over 18,000 signals, naive enumeration is impractical, and standard retrieval is ineffective; *(2) Implicit Semantic Grounding:* Standard methods for linking queries to schemas fail because the high-level analytical intent in Q does not directly match the database column names. Instead, it requires bridging a massive semantic gap to connect to the correct low-level signals using only the specification $\mathcal{D}$; *(3) Multi-CTE SQL Pipelines:* Answering queries requires synthesizing complex, multi-step Common Table Expression (CTE) pipelines rather than formulating standard relational SQL queries. These sequential pipelines are essential for executing operations, such as column decoding, rarely encountered in typical NL2SQL tasks.

2.2 Limitations of Existing Methods

To better understand the limitations of existing methods and motivate our system design, we conducted a preliminary study using 40 real-world CAN analytics requests at NIO. We evaluated standard RAG [22] for signal retrieval, alongside GPT-5.2, MAC-SQL [37], and DIN-SQL [30] for SQL generation. For GPT-5.2, we used a multi-step pipeline: prompting the LLM with all signal names to select candidates, matching these to full DBC specifications, and generating SQL using 3-shot examples. MAC-SQL and DIN-SQL were paired with standard RAG to fetch DBC evidence.

(Observation 1) Inaccurate Signal Retrieval. Existing methods struggle to identify correct signals from high-level analytical intents rather than exact signal names. Figure 2 shows GPT-5.2 failing in 24 cases, while MAC-SQL and DIN-SQL (paired with RAG) fail

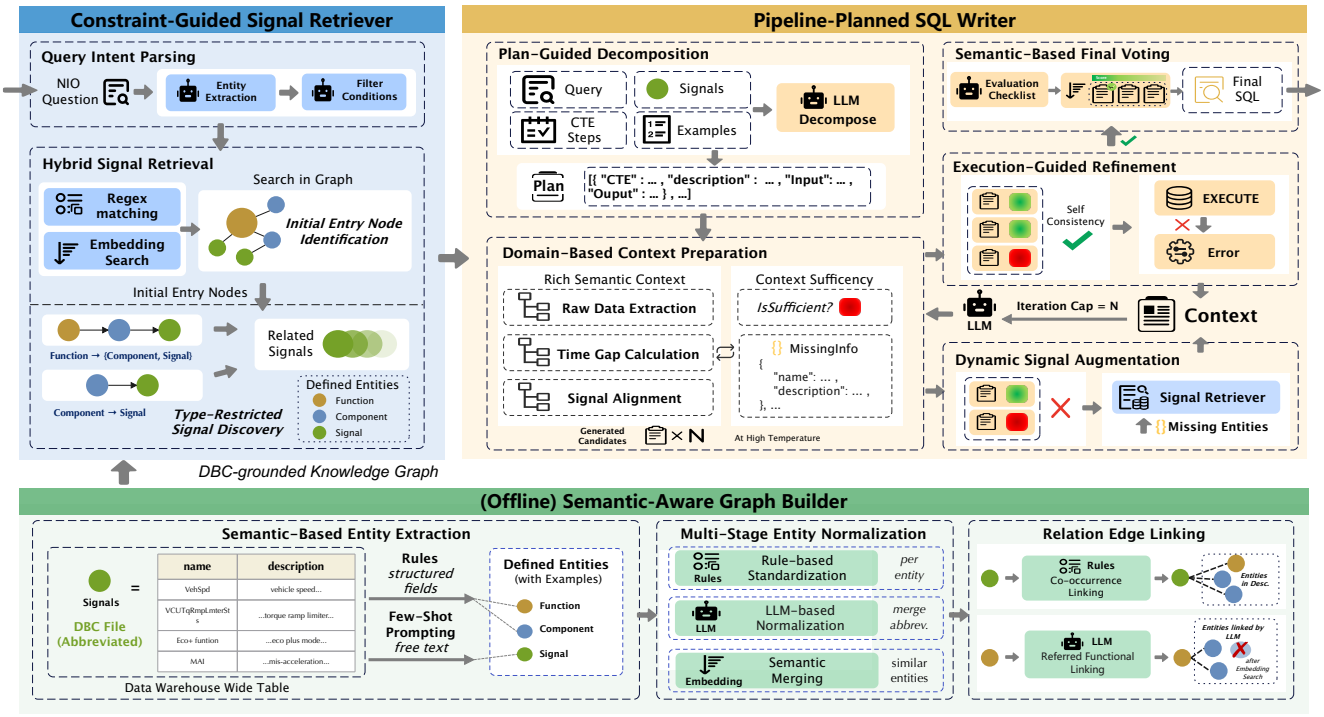

Figure 3: System Architecture and Workflow of `NIO-Weaver`.

in 33 out of 40 cases. For example, when a request mentions a complex vehicle function such as Mis-Acceleration Intervention, these baselines typically capture only a single name-matching signal (e.g., `MAISts`) or omit one of the jointly required signals (e.g., retrieving `VCUTqRmpLmterSts` but missing `TqRmpLimReq`). This prevents the construction of the activation predicate, yielding semantically incomplete SQL. This failure highlights the severe semantic gap between natural-language queries and DBC-defined technical names when multiple signals are required [11].

(Observation 2) Wrong Key Filtering Logic. Even when methods successfully retrieve the correct signals, they often produce incorrect filtering conditions. As shown in Figure 2, wrong key filtering logic occurs in 26 cases out of 40 requests for GPT-5.2. This error can also be observed when pairing methods with standard RAG, i.e., 35 cases for MAC-SQL+RAG and 34 for DIN-SQL+RAG, indicating that an executable query does not imply semantic correctness. Typical errors include: (1) applying the correct signal names but filtering for the wrong values; (2) dropping necessary conditions by treating a complex multi-signal state as a single filter; or (3) completely missing standard domain checks. Consequently, the generated SQL may execute successfully, but it will ultimately count the wrong events or analyze an incorrect time window.

(Observation 3) Incomplete CTEs Composition. Existing NL2SQL methods often produce incomplete pipelines, which are typically required in CAN analytics. As displayed in Figure 2, GPT-5.2 produces incorrect SQL queries that can not be executed in 34 out of 40 cases. Methods with the standard RAG module also omit key pipeline steps in the executable SQL queries. Common mistakes include missing data alignment (e.g., failing to order the data by vehicle) or skipping time-gap calculations. These issues motivate a plan-guided, validation-aware SQL synthesis component that iteratively repairs the structure and augments the triggered signals when the current processed context is insufficient.

3 SYSTEM OVERVIEW

Architecture. Figure 3 presents the architecture of `NIO-Weaver`.

- ***(1) Semantic-Aware Graph Builder***: To reliably connect high-level analytical intent to specific signals, we construct a DBC-grounded knowledge graph. Unlike general GraphRAG methods [4] that process unstructured text into noisy and incorrect graphs due to free-text construction, we extract core entities using established rules and few-shot LLMs to form a strict, domain-specific schema, which is further refined through three cleanup strategies (Section 4).
- ***(2) Constraint-Guided Signal Retriever***: To accurately identify necessary signals often missed by standard retrieval methods, we extract key entities and constraints from the query using LLM-based reasoning and rule-based matching. These guide a hybrid search that combines vector embeddings, exact matching, and type-constrained traversal to discover all functionally related signals required for the analytical intent (Section 5).
- ***(3) Pipeline-Planned SQL Writer***: To prevent failures when generating complex, multi-step CTE SQL in a single pass, we adopt a pipeline-planned approach. We use few-shot LLMs to decompose the generation into a sequence of standard CAN analysis steps. We then produce multiple SQL candidates, proactively retrieve any missing signals, and finalize the output through execution-guided error correction and semantic-based voting (Section 6).

System Workflow. It comprises two stages: (1) an offline stage for constructing the DBC-grounded knowledge graph, and (2) an online stage for retrieving signals and writing SQL queries.

❶ **Offline Stage.** To build a knowledge graph from raw DBC files, the *Semantic-Aware Graph Builder* first extracts entities using predefined rules and LLM-based reasoning. It then ensures consistency by cleaning and consolidating these entities via rule-based formatting, LLM-guided normalization, and similarity merging. Finally, it connects the refined entities to form the graph's structure, establishing relations based on common patterns such as co-occurrence.

Importantly, this offline stage is the only stage that requires human intervention, where domain experts define the initial graph schema (i.e., node and link definitions, rather than manual graph construction) and traversal rules, and may optionally review newly added metadata during periodic maintenance.

❷ **Online Stage.** First, the *Constraint-Guided Signal Retriever* extracts entities from the user query and uses combined text and embedding searches to locate an entry node in the graph. It then systematically traverses the graph to gather the required signals. Next, the *Pipeline-Planned SQL Writer* uses these signals to generate a step-by-step plan for generating the multi-CTE SQL candidates. During generation, it actively checks whether all necessary signals are present; if any are missing, it triggers iterative retrieval to identify them. Once the retrieval is complete, it refines the SQL candidates based on execution feedback. Finally, it evaluates the valid candidates and uses a voting process to output the final SQL. Crucially, the online stage is fully automated and requires no human intervention. By leveraging dynamic signal augmentation, execution-guided refinement, and semantic-based voting, it eliminates the need for experts to identify signals or repair SQL queries.

Generality and Robustness. The modular design of `NIO-Weaver` enables adaptation to new domains and evolving metadata without architectural redesign. Developers can modify only the domain-specific interfaces, such as schema extraction rules, typed traversal constraints, or the CTE-step vocabulary, while keeping the core modules unchanged. For example, if a domain uses an internal database catalog rather than external DBC files, the graph builder can be redirected to extract metadata from the catalog and rerun offline to refresh nodes and embeddings. Since the online pipeline treats the graph as a dynamic input, it automatically operates on the updated metadata without further changes.

4 SEMANTIC-AWARE GRAPH BUILDER

A significant challenge in CAN signal analytics is the reliance on external signal definitions (e.g., DBC files), which provide only low-level metadata and do not explicitly establish mappings from high-level analytical intent to the low-level signals. To bridge this gap, we propose a *Semantic-Aware Graph Builder* that constructs a DBC-grounded knowledge graph to facilitate the identification of concrete signals. Unlike general-purpose KG construction or GraphRAG methods that extract open-ended triples from unstructured text, we construct a highly constrained, task-specific graph. By strictly limiting nodes and relations to domain-specific types, we deliberately trade open-ended generality for domain precision to filter out noise and preserve executable metadata. Moreover, compared to unconstrained LLM-based KG extraction, we employ LLMs strictly as controlled tools within a deterministic pipeline for targeted tasks, such as resolving abbreviations or extracting implicit concepts. These outputs are then rigorously cleaned through rule-based formatting and semantic merging to prevent hallucinations and ensure the structural integrity for accurate SQL generation.

4.1 DBC-Grounded Knowledge Graph

While general graph extraction methods offer valuable flexibility for open-domain text, applying them directly to automotive signal data can sometimes yield overly large or noisy entities and relations. To tailor the graph to our specific domain and successfully bridge the gap between high-level analytical intent and raw vehicle signals, we design a strict graph schema with well-defined entity and edge types for our knowledge base. Built specifically for CAN signal analytics, this schema tightly constrains which entities and relations can exist, based on the metadata in the vehicle's DBC files.

Table 1: Entity Node and Relation Edge in `NIO-Weaver`.

Element	Description	Example
Entity Node ($\mathcal{V}$)		
Signal	Represent a specific, measurable data point transmitted across the vehicle's CAN bus.	`VehSpd`
Component	Represent a physical hardware part or a specific electronic control unit within the car.	`FrontElectricMotor`
Function	Represent a high-level vehicle feature, driving mode, or operational state (e.g., group functions).	`ECOPlusMode`
Relation Edge ($\mathcal{E}$)		
BELONGS_TO	Link a raw Signal to the specific automotive Component or Function it supports.	Signal → Component Signal → Function
CONTROLS	Link a high-level vehicle Function to the physical Component it actively operates or commands.	Function → Component

Formally, we define the DBC-grounded knowledge graph as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ represents the set of entity nodes and $\mathcal{E}$ represents the set of relation edges in Table 1.

Entity Node. We define entity nodes $\mathcal{V}$ as the core physical and logical elements of the vehicle's electronic architecture, strictly restricted to three types: $\mathcal{V}$ = {Signal, Component, Function}. (1) A Signal represents a specific, measurable data point on the CAN bus (e.g., `VehSpd` is the vehicle speed); (2) A Component denotes a physical hardware part or electronic control unit (e.g., `FrontElectricMotor` is the front electric motor for generating propulsion torque); and (3) a Function captures high-level features or operational states (e.g., `ECOPlusMode` is a specific energy-saving driving state), consolidating similar concepts such as functions and driving modes to handle semantic overlap. Using this limited node types helps keep related information together and avoids data scattering. CAN analytics queries primarily focus on "what" is working (Function), "where" it is located (Component), and "how" it is measured (Signal). Introducing more types would unnecessarily fragment useful information and hinder search efficiency without benefiting SQL generation.

Relation Edge. Relation edges $\mathcal{E}$ define the structural connections among entities (which contain the most important semantic information), facilitating the flow of data and commands within the automotive network. While we present two common edge types for clarity, $\mathcal{E}$ = {BELONGS_TO, CONTROLS}, the specific edge types are secondary to the system's overall functionality. (1) The BELONGS_TO edge links a raw Signal to the specific automotive Component or Function it supports, derived directly from the engineering DBC file; (2) The CONTROLS edge links a high-level vehicle Function to the physical Component it actively operates or commands. Although we focus on these two edge types as examples, `NIO-Weaver` remains flexible and can easily be extended to additional edge types by explicitly defining them. By keeping the edge types simple, we significantly reduce graph complexity and improve search efficiency, without limiting the system's ability to handle diverse and complex automotive queries.

4.2 Entity Node Extraction

The task of extracting relevant entities from semi-structured metadata in DBC files is complicated by the fact that they are often buried within free-text descriptions that may vary in terminology and formatting. To overcome this, we propose a robust extraction

and cleaning approach that recognizes both explicit and implicit references to critical signal data, ensuring that the resulting entity nodes accurately support downstream signal retrieval and analysis.

Semantic-Based Entity Extraction. While Signals are well-defined in structured metadata fields in the DBC files, Components and Functions are frequently embedded within unstructured free-text descriptions. To identify these implicit entities, we use few-shot LLMs to process serialized JSON inputs containing (1) signal names, (2) signal descriptions, (3) signal value choices, and (4) physical limits. The LLMs output an entity array, where each entry includes (1) an entity type (as defined in Section 4.1), (2) a standardized name expanded from any abbreviations, and (3) a comprehensive description detailing its working mechanism and application scenarios. For example, from `WinFrntLeInitLossSts`, the LLM extracts "left door component" (Component), explicitly isolating hardware and functional intent while excluding generic attributes such as "status".

Multi-Stage Entity Normalization. Metadata noise (e.g., naming inconsistencies, abbreviations) in legacy DBC files might fragment naive graphs, often treating synonyms such as "EPS" and "ElecPwrStr" as distinct. To produce a compact graph, we resolve redundancies via a three-stage pipeline: *(1) Rule-Based Standardization:* (i) Enforces CamelCase formatting, (ii) normalizes special characters (e.g., substituting "+" with "Plus"), and (iii) resolves label conflicts (e.g., prioritizing Component over Function); *(2) LLM-Based Normalization:* Maps remaining abbreviations to canonical forms (e.g., "Tq" to "Torque"); *(3) Embedding-Based Semantic Merging:* Merges entities whose name embeddings exceed a strict cosine similarity threshold (e.g., $\tau = 0.9$), ensuring a concise, duplicate-free graph.

4.3 Relation Edge Linking

To successfully ground the extracted Functions and Components within relevant Signal sets, rather than leaving them as isolated entities, we introduce two complementary linking mechanisms. These mechanisms connect entities to specific signals and to one another while strictly adhering to our predefined graph schema.

Heuristic Co-occurrence Linking. We leverage the metadata structure in the DBC files to link entities. If a Component or Function entity is extracted directly from the text description of a specific Signal, we establish an explicit edge between them. For example, we connect the `WinReRiDropSts` (Signal) to the `RearRightWindowRegulator` (Component) since it is mentioned within that signal's description in the DBC files.

Inferred Functional Linking. To capture deeper system dependencies that are not explicitly documented in the DBC files, we introduce an LLM-inferred linking process. Given a specific Function, the LLM hypothesizes which Components it likely controls and generates a preliminary list of candidate component names. We then ground these candidates within the existing graph by retrieving the most semantically similar Component nodes using vector embeddings. To prevent hallucinated or incorrect relationships, a connecting edge is established only if the best match exceeds a strict confidence threshold (e.g., cosine similarity $\geq$ 0.8). For instance, if the `WindowAntiPinch` (Function) is described as "preventing the window from closing", the LLM might propose the candidate `WindowRegulator` (Component). It is rigorously verified against existing Component nodes before the edge is created.

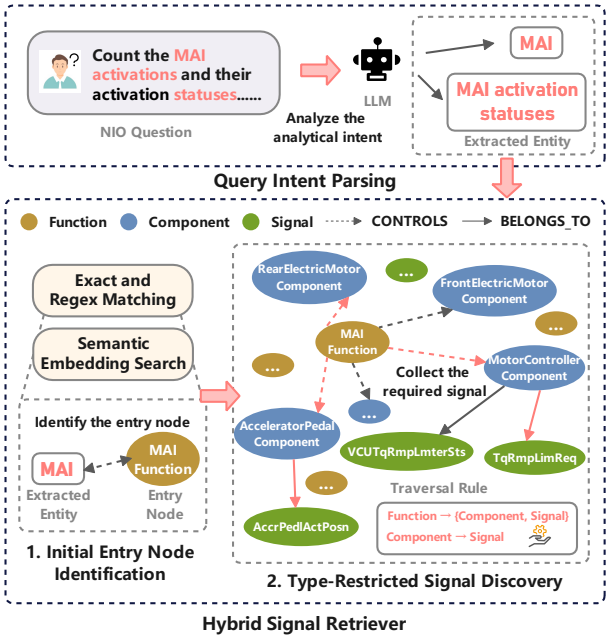

Figure 4: Example of *Constraint-Guided Signal Retriever.*

5 CONSTRAINT-GUIDED SIGNAL RETRIEVER

While the *Semantic-Aware Graph Builder* establishes a domain-specific knowledge graph for CAN signal analytics, it remains difficult to navigate the graph to isolate the precise subset of signals required to answer a user query. Specifically, experts typically describe high-level symptoms (e.g., "unexpected braking"), whereas the database stores low-level signal names, and a single symptom might correspond to multiple signals. To bridge this gap, we propose the *Constraint-Guided Signal Retriever* that employs a hybrid retrieval strategy. As illustrated in Figure 4, unlike typical graph retrieval systems that rely solely on raw query embeddings, we integrate query intent parsing with rule-constrained graph traversal to ensure comprehensive and accurate signal identification.

5.1 Query Intent Parsing

Since experts rarely refer to specific signals by their exact signal names, directly leveraging the raw query to retrieve the signals might result in missing or irrelevant context. To address this issue, we introduce a query intent parsing step that extracts entities and SQL filtering conditions before retrieving the required signals.

We first perform intent parsing over the query using the entity extraction strategy in Section 4.2, isolating specific mentions of Functions, Components, and Signals. Apart from the entities, we also extract concise filter conditions that describe the required computations or logical checks, encompassing both explicit constraints and implicit physical dependencies. For example, if a user queries the "activation status" of a safety feature, the LLM deduces the implicit physical dependencies required to evaluate that state, such as the vehicle speed, accelerator pedal position, and motor torque output, rather than just matching the literal text. Concurrently, to capture cases where exact technical CAN terms are used, we apply rule-based regular expression patterns tailored to known signal

Table 2: Vocabulary of CTE-Based Plan Steps Extracted from Real-World NIO Production for CAN Signal Analytics.

CTE Operation Type	Description	Key SQL Constructs
Raw Data Extraction	Select required CAN payload columns enforcing filters (e.g., time or vehicle) for partition pruning.	`SELECT ... WHERE datetime >= ...`
Signal Decoding	Convert columns into physical signal values (e.g., via the can_decode UDF) and bind stable aliases.	`can_decode(msg, ..., signal_name) AS ...`
Time-Gap Computation	Measure the time elapsed between consecutive records to identify gaps where the signal was lost.	`ts - LAG(ts, 1) OVER ( PARTITION BY ... ORDER BY ...) AS ts_gap`
Signal Alignment	Align signals recorded at different rates by carrying the last known value forward to fill missing data.	`LAG(col, 1) OVER ( PARTITION BY ... ORDER BY ...) AS pre_col`
Window Analysis	Extract sequential information, such as looking up a signal's previous value in chronological order.	`ROW_NUMBER() OVER ( PARTITION BY ... ORDER BY ...)`
Condition Flagging	Encode domain predicates and rule checks into boolean or state flags (e.g., evaluating time gaps).	`IF(condition, 1, 0), CASE WHEN ... THEN ... END`
Event Grouping	Group flagged data points into distinct events via cumulative sum to separate continuous episodes.	`SUM(flag) OVER ( PARTITION BY ... ORDER BY ...) AS event_id`
Aggregation & Statistics	Summarize over grouped events to calculate descriptive metrics such as durations and averages.	`GROUP BY ..., MAX(ts) - MIN(ts), AVG(val)`
Binning	Discretize continuous values, such as durations, into fixed ranges to analyze their distribution.	`CASE WHEN val >= ... THEN ... END`

naming conventions beyond the LLM-based extraction. For example, as shown in Figure 4, given a query regarding "Mis-Acceleration Intervention (MAI)", it extracts the entity `MAI` alongside filter conditions for activation counting (the frequency of the event) and activation status (the associated vehicle parameters, such as speed or motor torque, during the event). This structured representation effectively filters out noise, focuses the retrieval process, and directly informs the downstream SQL generation logic.

5.2 Hybrid Signal Retrieval

To accommodate diverse query patterns and prevent the retrieval set from expanding with irrelevant noise [24], we propose a hybrid retrieval strategy. We organize this process into two phases to generate a highly relevant candidate signal set M_Q from the knowledge graph: (1) locating the initial entry nodes, and (2) identifying all reachable signal nodes through constrained traversal.

(1) Initial Entry Node Identification. We first identify an initial set of entry nodes (i.e., the retrieval starting points), which may comprise Functions, Components, or Signals, using two complementary matching strategies based on the parsed query entities: (i) *Exact and Regex Matching:* To accommodate engineering debugging queries that contain specific signal patterns, we apply deterministic regular expression matching to the graph's signal names (e.g., extracting all Signal nodes matching `.*_Stat`); (ii) *Semantic Embedding Search:* We generate vector embeddings for each extracted entity and perform a top-k similarity search against the node embeddings in the graph. It effectively captures synonymous analytical intent (e.g., mapping the term `RPM` to the entity node `EngineSpeed`). Any Signal nodes directly identified constitute the entry nodes.

(2) Type-Restricted Signal Discovery. Since expert queries often express high-level intents rather than exact signal names, the retrieved entry nodes frequently include Function and Component nodes. To pinpoint exact signals without triggering an explosion of irrelevant signals, we expand these non-signal nodes using strict traversal rules based on the vehicle's physical hierarchy: (i) Function nodes only expand to Component or Signal neighbors (`Function` → {`Component`, `Signal`}), mapping functions to the hardware they control or monitor (e.g., `MAIFunction` expands to `MotorControllerComponent` in Figure 4); (ii) Component nodes only expand to Signal neighbors (`Component` → `Signal`), linking hardware to the raw data it generates (e.g., `MotorControllerComponent` expands to `VCUTqRmpLmterSts` and `TqRmpLimReq`); and (iii) Signal nodes are strict terminal leaves with no outward expansion. It prevents shared generic signals (e.g., `DoorOpen`) from pulling in unrelated components (e.g., the `Trunk`).

Ultimately, the candidate signal set M_Q comprises the union of initially retrieved Signal entries and all Signal nodes accessible via these traversal rules. To prevent missing crucial signals, we introduce on-demand dynamic signal augmentation driven by intermediate SQL generation requirements (Section 6.2). While M_Q strictly contains the necessary signals, keeping the intermediate Component and Function nodes as structural context helps generation accurately formulate logical groupings and filter conditions.

6 PIPELINE-PLANNED SQL WRITER

While existing NL2SQL methods excel at standard relational SQL, CAN analytics often require complex procedural pipelines to process data in a sequence of specialized Common Table Expressions (CTEs). Direct, single-pass generation of these operations remains challenging [43]. To address this, we propose *Pipeline-Planned SQL Writer*, which operates the workflow by decomposing the NL query into a structured, CTE-based plan. As illustrated in Algorithm 1, by decoupling semantic retrieval from logic generation, we instruct LLMs to iteratively generate, validate, and refine SQL.

6.1 Plan-Guided Decomposition and Generation

To address the limitations of standard NL2SQL methods, which typically miss or fail to produce the complete multi-CTE steps for complex CAN analytics pipelines, we propose a plan-guided decomposition and generation mechanism. Instead of generating the final SQL directly, it instructs LLMs to produce an explicit, step-by-step logical plan before generating the SQL queries.

Plan-Guided Decomposition. From inspecting hundreds of gold-standard CAN analytics SQL queries, we distill a library of nine recurring pipeline operations, each representing a CTE step of a distinct analytical purpose. As shown in Table 2, these operations span the entire analytics lifespan, progressing from basic extraction (Raw Data Extraction) and temporal normalization (Signal Decoding, Time-Gap Computation, Signal Alignment) to state evaluation (Window Analysis, Condition Flagging, Event Grouping), and concluding with summarization (Aggregation & Statistics, Binning). We provide this comprehensive library to the LLM as a standard vocabulary, allowing it to instantiate each plan step as a CTE in the final SQL. This CTE-step vocabulary is semi-automatically generated from historical queries via a four-step pipeline: (1) parsing queries into individual CTE blocks, using an LLM when standard parsers fail; (2) normalizing blocks with placeholders to extract core structural signatures (e.g., `SUM(...) OVER`); (3) clustering the top-k frequent patterns by structural similarity and analytical purpose;

Algorithm 1 Pipeline-Planned SQL Writer

Input: NL Query Q, Signal Set M_Q, Filter Condition C_Q, Table $\mathcal{T}$
Output: Final SQL $\mathcal{S}$

$Context \leftarrow \text{PrepareContext}(Q, M_Q, C_Q, \mathcal{T})$
for $n = 1$ **to** max iterations **do**
  // 1. Plan-Guided Decomposition and Generation
  $Candidates \leftarrow \text{GeneratePlanAndSQL}(Context)$
  // 2. Dynamic Signal Augmentation
  $MissingEntities \leftarrow \text{SufficientCheck}(Candidates)$
  if $MissingEntities \neq \emptyset$ **and** $n <$ max iterations **then**
    $NewSignals \leftarrow \text{SignalRetriever}(MissingEntities)$
    $M_Q \leftarrow M_Q \cup NewSignals$
    $Context \leftarrow \text{PrepareContext}(Q, M_Q, C_Q, \mathcal{T})$
  else
    // 3. Execution-Guided Refinement
    $ValidCandidates \leftarrow \text{Repair}(Candidates, Context)$
    break
  end if
end for
// 4. Semantic-Based Final Voting
Final SQL $\mathcal{S} \leftarrow \text{FinalVote}(ValidCandidates, Context)$
return Final SQL $\mathcal{S}$

and (4) employing an LLM to propose candidate templates, which are integrated after rigorous execution testing and expert review.

Domain-Grounded Context Preparation. To effectively ground the planning and generation process, we first construct a rich semantic context. Given a user query Q, the retrieved signal set M_Q, and the parsed filter conditions, we augment the prompt with database schema definitions and a dynamically retrieved set of few-shot examples with standard CAN analytics patterns from real-world scenarios at NIO. These examples are selected via embedding-based similarity to provide reusable planning hints (e.g., window-based aggregation) while avoiding the leakage of specific SQL details. Crucially, the "key SQL constructs" (Table 2) associated with these planning hints are not executable modules, but textual SQL syntax patterns provided within this prompt. Instead of invoking external functions, the vocabulary instructs the LLM using specific syntax templates (e.g., `can_decode(...)` for decoding or `last_value(...)` for temporal alignment) for each CTE operation.

Incremental CTE Generation. We first employ Chain-of-Thought (CoT) prompting to select an ordered subset of these operations and formulate a structured step plan. We formally define this logical plan as a strict, dependency-preserving execution sequence. Given the CTE operation vocabulary O, a query plan is $P_Q = [s_1, \ldots, s_k]$. Each step $s_i = (o_i, I_i, O_i)$ applies an operation $o_i \in O$ to inputs I_i yielding intermediate outputs O_i. This sequence must form a topological order, ensuring all inputs are fully resolved by preceding steps. For instance, "computing statistics on ECOPlus mode" (Table 7) forms a simple topological sequence: applying Raw Data Extraction to isolate the signal, followed sequentially by Binning to categorize the values. To translate this logical plan into executable SQL, we generate multiple plan-to-SQL candidates by synthesizing SQL queries via instantiating the plan into a sequence of CTEs conditioned on the prepared context. During generation, we enforce a strict topological rule: while candidates may vary in their internal implementation logic, they must preserve the chosen plan steps and refer to only the valid schema. For each signal in M_Q, we explicitly handle decoding using the functions (e.g., `can_decode(payload, msg_id, signal_name)`) and bind them to aliases (e.g., `veh_spd`) for consistent reference. To encourage a diverse set of plausible SQL candidates, we use a moderately high sampling temperature (e.g., 0.7) to produce diverse SQL queries.

6.2 Iterative Augmentation and Refinement

While the plan-guided methods yield structurally sound SQL queries, they may still fail due to hallucinations (attempting to filter by missing signals), execution errors (syntax faults or type mismatches), or semantic errors (incorrect logic despite successful execution). To further enhance the SQL generation accuracy, we implement context augmentation, execution-guided refinement, and semantic-based voting to ensure the final output SQL is accurate.

Dynamic Signal Augmentation. To prevent failures from missing signals, we explicitly instruct LLMs to evaluate if the current signal set M_Q is sufficient for the plan. Using a self-consistency mechanism across all SQL candidates, the LLM evaluates its CTEs to output a boolean sufficiency status and a list of potential missing entities. We deduplicate these entities and trigger an additional retrieval, adding newly discovered signals to M_Q and restarting generation. To prevent infinite loops, we cap iterations, returning the results and error messages if the context remains insufficient.

Execution-Guided Refinement. To ensure generated SQL queries are robust [40], we validate candidate execution on mirrored target databases. For strict industrial environments where executing unverified queries is restricted, we provide a flexible fallback using offline syntax parsers and LLM-based judgments to simulate feedback [44, 46]. When validation fails, we use the recorded error feedback to prompt the LLM for repairs. To encourage rapid convergence, repairs are strictly limited to local fixes (e.g., resolving invalid columns) without altering the underlying plan. This iterative refinement continues until validation passes or the maximum iteration limit is reached, defaulting to a randomly selected candidate alongside its final error message if unresolved.

Semantic-Based Final Voting. To filter out subtle logic errors that might pass validation but fail to yield accurate analytical insights, we employ a semantic-based voting mechanism. Specifically, executable SQL candidates proceed to a final review by a separate judge agent, which ensures the SQL queries are semantically faithful to the high-level analytical intent. The judge evaluates the full context to produce a scalar score and a violated checklist about (1) semantic fidelity (e.g., ensuring explicit constraints such as `speed > 80` are correctly implemented), (2) completeness of required filters, (3) structural readability (e.g., use of CTEs), and (4) robustness to nulls and edge cases. To prevent stylistic preferences from dominating the outcome, the judge first enforces strict hard constraints: the SQL must refer only to existing columns, use only the signals in M_Q, and strictly preserve the given plan steps. The highest-scoring candidate is selected as the final SQL $\mathcal{S}$.

7 EXPERIMENTS

We assess `NIO-Weaver` to answer the following questions.

• **RQ1 (Overall Performance):** How does NIO-Weaver compare to state-of-the-art LLMs and NL2SQL frameworks, particularly when combined with traditional RAG-based signal retrieval methods, in terms of SQL generation accuracy?
• **RQ2 (Retrieval Quality):** How does our *Constraint-Guided Signal Retriever* compare to LLM-based and general RAG-based methods in effectively bridging the semantic gap in CAN signal analytics?
• **RQ3 (Ablation Study):** What is the contribution of each of the three components in NIO-Weaver to the overall performance across retrieval and SQL generation tasks?

7.1 Experiment Setup

Database and DBC Specifications. We use a real-world CAN telemetry database from NIO, stored in a cloud-based data warehouse (e.g., Hive [36]) as a time-partitioned, denormalized wide table. The database contains data from 2,781 deduplicated vehicles, comprising 3,414,636 samples (rows) across more than 300 days. The table has 468 columns, of which 441 are related to CAN signals, and the user-defined function (can_decode) is used to extract the relevant signal values. We use DBC files as the semantic specification for signal meanings and decoding rules, which define a total of *18,372 signals* across *441 message IDs*, demonstrating a substantial semantic gap in CAN signal analytics.

Datasets. We evaluate NIO-Weaver using two NL2SQL datasets, both based on the same warehouse schema and DBC specifications. ***(1) Industrial*** queries consist of 102 real-world production requests from NIO. Each query is paired with its required signal set and a ground-truth multi-CTE SQL; ***(2) Synthetic*** queries include 200 manually crafted questions based on domain templates (e.g., window-based event detection) and are paraphrased into natural language. Each synthetic query is paired with the required signals and a ground-truth SQL, verified through database execution and human examination. Both datasets are further stratified into *Simple / Moderate / Hard* subsets for fine-grained analysis. A query is considered more difficult if it involves: (i) more signals that do not directly link to the query, including implicit dependencies, (ii) a larger number of CTE steps (e.g., decoding, alignment, aggregation), and (iii) more complex reasoning (e.g., window joins).

Evaluation Metrics. We employ generation and retrieval metrics.
(1) SQL Generation. To evaluate SQL generation, we adopt two F1-based metrics designed to capture the structural and data correctness required for industrial CAN signal analytics.

Value-Level F1 (*VL*): Measures the similarity of execution results between the generated and the gold SQL. We treat the results as unordered multisets, capturing value overlap after normalization. Formally, let $M_V = \left|\mathcal{V}(R_{gold}) \cap \mathcal{V}(R_{gen})\right|$ be the matched value count, we compute this metric as follows.

$$P = \frac{M_V}{|\mathcal{V}(R_{gen})|}, \quad R = \frac{M_V}{|\mathcal{V}(R_{gold})|}, \quad VL = \frac{2PR}{P+R}$$

Enhanced Semantic Component F1 (*ESC*): Evaluates the semantic accuracy of the SQL logic, building on Spider's Component Match [41]. We extract a multiset of components $C(S)$ from each CTE block, including standard SQL elements, UDF calls, window functions, and time-window predicates. After normalizing aliases and syntax variations, let $M_C = \left|C(S_{gold}) \cap C(S_{gen})\right|$ be the matched component count. We report the macro-average ESC:

$$P = \frac{M_C}{|C(S_{gen})|}, \quad R = \frac{M_C}{|C(S_{gold})|}, \quad ESC = \frac{2PR}{P+R}$$

We prioritize these metrics over the widely adopted *Exact Match* and *Execution Accuracy* from existing benchmarks [26, 41] to better capture the requirements of industrial CAN signal analytics: ❶ First, *Exact Match* is overly strict for complex analytical logic. CAN signal analytics SQL queries require multi-layer CTEs, UDF-based signal decoding, and window functions, meaning identical business logic can be correctly implemented using varying syntax (e.g., a LAG function versus a self-join). *ESC* resolves this by evaluating abstract semantic components, ensuring logically equivalent queries are not unfairly penalized; ❷ Second, a binary *EX* score is fundamentally unreliable for time-series CAN analytics. Valid SQL implementations often produce minor timestamp boundary mismatches during event period splitting. *EX* unfairly fails the entire query for these isolated differences. Crucially, lowering numeric precision or sorting results cannot fix this: varying methods for handling error flags or sampling rates fundamentally alter actual event boundaries. These represent structural divergences in logic, not trivial numerical variations. *VL* mitigates this by assessing the proportion of matching multiset values, effectively capturing partial correctness; ❸ Third, production environments demand strict verification of both logic and data. A binary *EX* score cannot distinguish a truly robust query from one that incidentally matches an aggregate value despite omitting a required signal. By jointly applying *VL* and *ESC*, we rigorously validate both the final data output and the underlying query logic, ensuring the generated SQL is transparent and reliable for engineering analytics [18].

(2) Signal Retrieval. For evaluating signal retrieval accuracy, we use standard *Recall* and *Precision* metrics. These metrics assess how well the method retrieves the required signals from the database, crucial for grounding the SQL generation.

Implementation Details. The default LLM used in NIO-Weaver for tasks such as entity extraction is *Qwen-3-Max*, which is run with a temperature of 0 for reproducibility. Only the *Pipeline-Planned SQL Writer* utilizes a higher temperature setting (0.7) to encourage a diverse set of 5 plan-to-SQL candidates. Signal augmentation and iterative refinement are limited to at most 3 iterations. The default embedding model is *Qwen-text-embedding-v4*.

7.2 Overall Performance (RQ1)

We compare NIO-Weaver against top LLMs and adapted NL2SQL methods across three categories. **(1) LLM Only:** We assess *Qwen-3-Max*, *Gemini-3-Pro*, *GPT-5.2*, and *DeepSeek-V3.2*. For signal retrieval, we provide the LLM with a list of signals and ask it to select relevant ones. For SQL generation, we offer the selected signals, their definitions, the table schema, and 3-shot examples. **(2) Adapted NL2SQL Methods**: We evaluate five baselines grouped into three types [17, 34]: *(a) Decomposition:* Methods that divide the task into sub-tasks like schema linking and generation (MAC-SQL [37], DIN-SQL [30], OpenSearch-SQL [39]); *(b) Prompt Optimization:* Methods that enhance performance via higher-quality prompts, such as selecting few-shot examples or integrating external knowledge

Table 3: Overall Performance Comparison across Different Datasets on *VL* (%) and *ESC* (%).

Method		Industrial Dataset						Synthetic Dataset					
		Simple		Moderate		Hard		Simple		Moderate		Hard	
		VL	ESC	VL	ESC	VL	ESC	VL	ESC	VL	ESC	VL	ESC
LLM Only	DeepSeek-V3.2 [1]	1.7	27.7	0.5	19.3	0.0	7.7	2.0	29.5	0.7	20.6	0.0	7.9
	Qwen-3-Max [5]	8.5	31.9	5.5	25.6	1.0	13.1	8.7	33.1	6.2	26.2	1.7	15.0
	Gemini-3-Pro [2]	16.0	41.3	9.8	36.0	1.9	21.3	19.7	44.1	10.3	36.4	2.4	23.4
	GPT-5.2 [3]	17.4	42.4	11.3	36.9	2.1	21.7	20.0	43.2	13.3	37.2	2.6	22.1
Decomposition	MAC-SQL [37]	32.4	54.5	19.6	45.0	8.8	36.7	32.5	56.6	21.7	47.3	9.1	36.1
	DIN-SQL [30]	31.0	53.8	19.7	44.3	9.1	37.2	31.7	55.8	21.9	46.9	9.8	37.7
	OpenSearch-SQL [39]	35.9	55.3	23.5	47.9	10.2	37.6	38.1	56.2	24.0	49.1	12.6	38.1
Prompt Optimization	DAIL-SQL [14]	39.1	58.6	24.0	49.0	13.6	39.6	41.0	59.9	24.7	52.4	14.0	41.1
Autonomous Agent	ReFoRCE [10]	39.7	58.5	24.1	49.1	13.7	38.7	41.2	59.3	25.1	51.0	13.9	39.8
NIO-Weaver (Ours)		**52.4**	**67.6**	**47.4**	**59.8**	**26.8**	**43.1**	**52.6**	**67.7**	**47.9**	**60.0**	**27.3**	**44.4**

Table 4: Latency Analysis of NIO-Weaver.

Stage	Mean	Median	P90	P95
Constraint-Guided Signal Retriever	6.46s	4.71s	6.67s	7.65s
Pipeline-Planned SQL Writer	11.10s	10.18s	16.96s	23.86s
End-to-End Latency	17.64s	14.39s	22.39s	36.08s

(DAIL-SQL [14]); *(c) Autonomous Agents:* Methods that use multi-turn database interactions to correct and refine SQL queries (ReFoRCE [10]). We standardize all baselines to use the same 3-shot examples as NIO-Weaver. We also incorporate LLM-based signal filtering into these methods. without this step, they achieve *0%* accuracy on the *Industrial* and *Synthetic* datasets.

Table 3 and Figure 5 report the overall performance using our two F1-based metrics, from which we have three observations. ***First,* NIO-Weaver *consistently outperforms all baseline methods across varying difficulty levels.*** On the *Industrial (All)* dataset, NIO-Weaver achieves 36.8% *VL* and 53.0% *ESC*, outperforming the best prompting-based baseline by 14.8% in *VL* and 7.3% in *ESC*. Similarly, performance improvements are consistent across the *Hard* query set (+13.1% *VL*, +4.4% *ESC*) and the *Synthetic* dataset (+14.4% *VL*, +6.7% *ESC*). These results demonstrate the effectiveness of integrating domain-specific signal grounding (*DBC-grounded Graph Builder*) and retrieval (*Signal Retriever*), alongside the decomposed multi-CTE SQL generation (*Pipeline Planned SQL Writer*). ***Second, LLM Only baselines struggle significantly to map user queries into specific CAN SQL queries.*** Even the best LLM Only approach (GPT-5.2) achieves only 7.9% *VL* and 30.4% *ESC* on the *Industrial (All)* dataset, with substantial performance regression on *Challenging* queries (to 2.1% *VL* and 21.7% *ESC*). It highlights the difficulty of aligning high-level analytical intents with the required signal set from the opaque CAN column names and of handling complex multi-CTE SQL queries, including operations such as decoding and temporal alignment logic, without domain-specific grounding. ***Third, NL2SQL frameworks, while helpful, are still limited by CAN-specific semantics.*** Methods such as DAIL-SQL and ReFoRCE show improvements, with DAIL-SQL achieving 21.8% *VL* and 46.2% *ESC*, and ReFoRCE attaining 22.0% *VL* and 45.7% *ESC* on the *Industrial (All)* dataset. However, their performance degrades on the *Hard* query set because they fail to effectively model multi-step SQL pipelines, which are essential in CAN analytics. For instance, in the query about analyzing compressor operating time in the ET7 model under specific conditions, DAIL-SQL captures the filtering condition `CompressorSts = 1 AND ReservoirValveSts = 1`, but fails to incorporate the necessary window analysis CTE or the event-segmentation step. In the same query, ReFoRCE typically extends to the filtering and aggregation steps but still overlooks the crucial intermediate logic, such as event segmentation. In contrast, NIO-Weaver effectively addresses these challenges by first structuring the SQL pipeline with the correct sequence of steps, ensuring that subsequent aggregation operates on correctly segmented events, rather than incomplete or misaligned data.

We have further conducted a latency analysis of NIO-Weaver across its core stages. As shown in Table 4, NIO-Weaver exhibits an average end-to-end latency of 17.64s and the 90th-percentile (P90) latency remains below 23s. Moreover, the *Pipeline-Planned SQL Writer* accounts for 62.9% of the total latency with an average latency of 11.10s compared to 6.46s of the *Constraint-Guided Signal Retriever*. These latencies are not prohibitive for industrial CAN signal analytics but highly practical for interactive usage and significantly improve upon manual efforts, where manual diagnostics often span minutes or even hours. To further reduce latency, our future work will focus on pre-filtering candidate signal sets and caching retrieved context to accelerate repeated analytical tasks.

7.3 Retrieval Quality (RQ2)

We evaluate the retrieval performance of NIO-Weaver against four baselines: **(1) LLM Only:** We supply the LLM with the complete set of deduplicated DBC-derived candidate signals, rather than a pre-filtered subset, and instruct it to select the relevant signals directly; **(2) Standard RAG [22]:** A dense retrieval baseline uses the original natural language (NL) query to retrieve the top 400 most similar signals (combining names and descriptions) based on cosine similarity. The LLM then processes these retrieved signals and the user query to select the final signals; **(3) GraphRAG [4]:** A graph-based retrieval approach that aggregates information from local to global contexts over an indexed knowledge graph to answer specific queries. It receives the original NL query wrapped with an instruction prompt asking for relevant CAN signals; **(4) HippoRAG 2 [16]:** A structured retrieval framework inspired by human memory. It improves context association through phrase and passage nodes, query-to-triple linking, and an LLM-based filtering step to guide graph retrieval. It also utilizes the original NL query similar to GraphRAG. For a fair comparison, all methods use the same LLM (*DeepSeek-V3.2*) and embedding model (*Qwen-text-embedding-v4*). Furthermore, to evaluate each method strictly on its native workflow, none of the baselines utilize the specialized LLM-based final selection module introduced in NIO-Weaver. Instead, *LLM-Only* and *Standard RAG* rely on their own standard LLM passes to select signals from their respective candidate pools, while for *GraphRAG* and *HippoRAG 2*, we extract and validate the selected signals directly from their native question-answering responses.

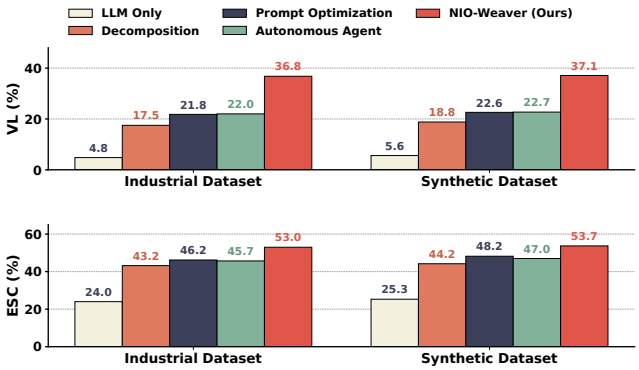

Figure 5: Overall Performance of Various Methods.

Table 5: Performance Analysis of NIO-Weaver Variants.

Methods	*VL* (%)	*ESC* (%)
w/o Signal Retriever	32.7	50.3
w/o Plan-Guided Decomposition	31.9	45.7
w/o Dynamic Signal Augmentation	35.2	52.1
w/o Execution-Guided Refinement	34.3	52.3
NIO-Weaver (Ours)	**36.8**	**53.0**

Table 6 and Figure 6 evaluate signal retrieval quality on the *Industrial* dataset, using *Recall* and *Precision* over the required signal set. Based on the results, we highlight three key observations. ***First,* NIO-Weaver *consistently achieves the highest Recall and Precision across all difficulty levels.*** NIO-Weaver achieves 50.7% *Recall* and 41.8% *Precision*, outperforming standard RAG by 28.0% in *Recall*, and 20.7% in *Precision*, as well as HippoRAG 2 by 29.4% in *Recall*, and 35.4% in *Precision*. The performance difference is particularly notable on *Hard* queries, where NIO-Weaver retains 38.7% *Recall* and 34.7% *Precision*, demonstrating its ability to effectively capture implicit dependencies; ***Second, general-purpose retrieval methods underperform due to the semantic gap and their unconstrained construction logic.*** Standard RAG achieves only 20.7% retrieval accuracy, while GraphRAG yields 0% across all difficulty levels. These failures stem from a reliance on lexical or semantic similarity that is absent in raw, opaque CAN signals. For example, in an ADAS activation query, GraphRAG retrieves only salient signals such as VLC while missing critical dependencies such as EASSts. Unlike general-purpose KG methods that extract arbitrary triples from unstructured text, our builder addresses this by constructing a highly constrained, task-specific graph. By strictly limiting nodes to three types, we effectively filter out the noise that causes unconstrained methods to fail, ensuring the structural integrity needed to capture implicit signal mappings; ***Third, while LLM-Only methods avoid graph construction, they present a steep tradeoff between token cost and complex reasoning.*** Directly prompting the LLM with all candidates incurs high inference token costs. Furthermore, while strong LLMs (e.g., GPT-5.2) reach roughly 51% *Recall* on *Simple* queries, performance drops to near 30% for both *Recall* and *Precision* on *Hard* queries. This proves long-context LLMs struggle with implicit dependencies, necessitating an external knowledge graph. In contrast, NIO-Weaver requires a higher initial offline setup for the DBC-grounded graph and traversal rules. However, this reusable overhead is easily amortized across repeated queries, ensuring a reliable runtime. NIO-Weaver consistently outperforms GPT-5.2 across all difficulties, achieving absolute precision improvements over 21%, confirming its necessity for accurate industrial signal grounding.

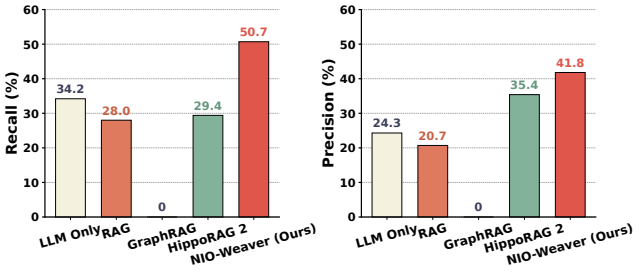

Figure 6: Signal Retrieval Performance Comparison ("LLM Only" represents the average performance of the LLMs).

Table 6: Signal Retrieval Performance Comparison.

Method	Recall (%)			Precision (%)		
	Simple	Moderate	Hard	Simple	Moderate	Hard
DeepSeek-V3.2 [1]	36.0	26.3	13.7	26.0	25.7	7.6
Qwen-3-Max [5]	47.7	41.6	23.2	31.4	36.9	16.5
Gemini-3-Pro [2]	51.0	47.7	30.0	45.6	33.1	29.1
GPT-5.2 [3]	51.5	47.7	31.5	32.1	26.4	21.5
RAG [22]	46.9	34.0	14.7	36.3	27.8	8.3
GraphRAG [4]	0.0	0.0	0.0	0.0	0.0	0.0
HippoRAG 2 [16]	48.3	36.7	15.0	49.8	37.3	31.1
NIO-Weaver (Ours)	**61.9**	**55.4**	**38.7**	**51.6**	**48.1**	**34.7**

7.4 Ablation Study (RQ3)

To assess the module impact, we evaluate several NIO-Weaver variants using *DeepSeek-V3.2* as the underlying LLM. *(1) w/o Signal Retriever*: Replaces *Constraint-Guided Signal Retriever* over the constructed graph with LLM-based filtering (*GPT-5.2*) for signal retrieval; *(2) w/o Plan-Guided Decomposition*: Disables the plan-guided decomposition mechanism in *Pipeline-Planned SQL Writer* during SQL generation; *(3) w/o Dynamic Signal Augmentation*: Disables the adaptive sufficiency check and retrieval for signal augmentation during SQL generation; *(4) w/o Execution-Guided Refinement*: Disables the execution-based validation and repair during generation.

Table 5 reports the results on the *Industrial* dataset, and we have three observations. ***First, Plan-Guided Decomposition is the most critical factor in generating correct SQL structures.*** Removing this mechanism causes the largest performance drop: *ESC* falls from 53.0% to 45.7% (-7.3%) and *VL* from 36.8% to 31.9% (-4.9%). This proves that step-by-step planning is essential for building complex, multi-CTE pipelines; without it, LLMs collapse logic into basic filters, such as skipping necessary window-based temporal bookkeeping in compressor-duration queries, resulting in semantically incorrect SQL; ***Second, the Constraint-Guided Signal Retriever provides a crucial advantage by ensuring the generated SQL executes correctly.*** Disabling it reduces *VL* from 36.8% to 32.7% and *ESC* from 53.0% to 50.3%. This shows that domain-specific graph retrieval solves the core problem of finding hidden signals. For instance, it identifies necessary sibling signals like EASSts in ADAS queries that LLM-based filtering misses, highlighting that even superior generators cannot compensate for incomplete signal sets at the retrieval stage; ***Third, adaptive signal augmentation and closed-loop feedback are essential for robust SQL generation.*** The *w/o Dynamic Signal Augmentation* variant reduces performance to 35.2% *VL* (-1.6%) and 52.1% *ESC* (-0.9%); removing *Execution-Guided Refinement* further lowers *VL* to 34.3% (-2.5%) while *ESC* remains at 52.3% (-0.7%). This suggests that augmentation and validation mechanisms are essential for dynamically recovering missing schema elements (e.g., ReservoirValveSts) and repairing syntactic errors to ensure final accuracy.

Table 7: Case Study of Real-World NL Query at NIO Production Addressed by NIO-Weaver.

Difficulty	Question	Involved CTEs	Signals	GPT5.2	DIN-SQL	OpenSearch-SQL	DAIL-SQL	ReFoRCE	NIO-Weaver
Simple	Statistics on ECOPlus mode activation status.	Raw Data Extraction → Binning	ECOPlusModSts	✓	✓	✓	✓	✓	✓
Simple	Statistics on the steering angle distribution of the NT2 at high speed.	Raw Data Extraction → Signal Alignment → Signal Decoding → Condition Flagging → Binning	SteerWhlAg, VehSpd	×	✓	✓	✓	✓	✓
Medium	Investigate the operation of the air compressor in the air suspension system of NT2 vehicles under low-temperature conditions (Ambient Temperature below -5℃).	Raw Data Extraction → Signal Alignment → Signal Decoding → Time-Gap Computation → Condition Flagging → Binning	AmbTemp, VehState, CompressorSts	×	✓	×	✓	✓	✓
Medium	Analyze the distribution of operating time for the compressor in the ET7 model under conditions where the compressor is active and the air tank is being replenished.	Raw Data Extraction → Condition Flagging → Window Analysis → Condition Flagging → Aggregation & Statistics → Binning	CompressorSts, ReservoirValveSts	×	×	×	×	×	✓
Hard	Analyze the distribution of operating duration under conditions of large steering angles and high torque.	Raw Data Extraction → Signal Alignment → Signal Decoding → Condition Flagging → Event Grouping → Binning	VehState, SteerWhlAg, SteerWhlAgAndSpdValid, VCUTqCmdImdtPEUF, VCUTqCmdImdtPEUFValid	×	×	×	×	✓	✓

7.5 Real-World Case Analysis

We analyze the generation of six real-world CAN analytics SQL queries in NIO. Each query requires a multi-CTE pipeline and involves a small set of execution-critical signals. Table 7 indicates whether each method generates an executable and semantically accurate SQL (✓) or fails (×). Based on Table 7, we outline three observations: ***First, only* NIO-Weaver *consistently handles long industrial pipelines.*** For a representative *Moderate* case of computing the compressor operating-time distribution under a conjunction of conditions, NIO-Weaver generates a long sequence of CTEs and strict predicate logic over multiple signals (e.g., CompressorSts, ReservoirValveSts), while all baseline methods fail on this specific case. This demonstrates that the components of NIO-Weaver are effective for synthesizing such complex pipelines. ***Second, conceptually simple CAN queries often require complex SQL implementations.*** While all methods successfully handle the most basic aggregation task, GPT-5.2 fails on a *Simple* query when the pipeline requires explicit temporal alignment, decoding, and event filtering. It highlights the difficulty of assembling an end-to-end executable pipeline from plausible SQL fragments under strict payload semantics. ***Third, Moderate queries expose failures in time-series state tracking.*** In these cases, the required logic frequently involves time-gap calculations, condition flagging, and episode-based aggregation. We observe that standard retrieval-augmented NL2SQL baselines become highly unstable when they must compose these sequential steps alongside the correct signal sets (e.g., AmbTemp, VehState).

8 RELATED WORK

NL2SQL. Recent LLM-based NL2SQL methods have evolved from optimizing few-shot prompts to deploying complex agentic workflows [14, 49, 53]. To handle advanced logic, multi-agent frameworks decompose the generation process into collaborative roles and automate self-correction [6, 15, 29, 37, 39]). Additionally, reasoning capabilities are enhanced via reinforcement learning, tree search algorithms [23], and synthetic data scaling [25]. Despite these advancements, these methods rely on a self-contained schema with visible semantics. In contrast, our automotive scenario involves opaque columns, with definitions externalized in DBC specifications. As a result, existing methods fail to bridge this semantic gap without a dedicated, domain-specific mechanism.

RAG. While standard Retrieval-Augmented Generation (RAG) [7, 50] effectively retrieves discrete text passages via embedding-based similarity, it struggles with queries that require a holistic understanding of the corpus. To address this, GraphRAG structures data into knowledge graphs, enabling models to traverse and synthesize scattered information for complex reasoning [55]. Recent innovations further optimize through document-centric graph construction, linear-complexity scaling [56], and adaptive frameworks that dynamically build reasoning structures or perform iterative retrieval [8]. However, these methods are predominantly designed for unstructured textual corpora. Our work differs in that it targets domain-specific CAN analytics, which relies on retrieval over definitions of signal, bit, and function specifications rather than narrative text. We propose a specialized graph-construction strategy that identifies implicit signals in high-level analytical intents, addressing the limitations of general-purpose retrieval methods.

9 CONCLUSION

We introduce NIO-Weaver, a multi-agent framework for natural-language analytics on CAN databases. To bridge the semantic gap between high-level user intents and opaque signal columns that hinders traditional NL2SQL methods, NIO-Weaver uses a domain-specific knowledge graph and hybrid retrieval to accurately identify required signals. Additionally, its hierarchical SQL synthesis engine uses taxonomy-guided self-correction to automate the generation of complex, multi-step pipelines. Empirical results show that NIO-Weaver significantly outperforms state-of-the-art baselines. In the future, we will introduce stronger CTE-level verification and execution-guided repair, ensuring that each intermediate step of the event logic can be validated before producing the SQL.

ACKNOWLEDGMENTS

Xuanhe Zhou is the corresponding author. This work was supported in part by National Key R&D Program of China (No. 2023YFB4502400), in part by China NSF grant (No. 62441236, 62372296, 62432007, U25A6024, U25A20437, 62525202, 62232009, 62502304), in part by Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM103), in part by Alibaba Group through Alibaba Innovation Research Program, in part by Tencent Rhino Bird Key Research Project, Shenzhen Project (CJGJZD20230724093403007), Zhongguancun Lab, and Beijing National Research Center for Information Science and Technology (BNRist), CCF Populus Grove Fund, ByteDance, Tencent, Ant Group through CCF-Ant Research Fund, Shanghai Jiao Tong University AI for Engineering Initiative.

REFERENCES

[1] DeepSeek-V3.2. *(DeepSeek AI).* https://huggingface.co/deepseek-ai/DeepSeek-V3.2

[2] Gemini 3 Pro. *(Google DeepMind).* https://deepmind.google/models/gemini/pro/

[3] GPT-5.2. *(OpenAI).* https://developers.openai.com/api/docs/models/gpt-5.2

[4] GraphRAG. *(Microsoft).* https://www.microsoft.com/en-us/research/publication/from-local-to-global-a-graph-rag-approach-to-query-focused-summarization/

[5] Qwen3. *(Qwen Team).* https://huggingface.co/collections/Qwen/qwen3

[6] Arian Askari, Christian Pölitz, and Xinye Tang. 2025. MAGIC: Generating Self-Correction Guideline for In-Context Text-to-SQL. In *AAAI*. AAAI Press, 23433–23441.

[7] Andrew Brown, Muhammad Roman, and Barry Devereux. 2025. A Systematic Literature Review of Retrieval-Augmented Generation: Techniques, Metrics, and Challenges. *Big Data Cogn. Comput.* 9, 12 (2025), 320.

[8] Shengyuan Chen, Chuang Zhou, Zheng Yuan, Qinggang Zhang, Zeyang Cui, Hao Chen, Yilin Xiao, Jiannong Cao, and Xiao Huang. 2026. You Don't Need Pre-Built Graphs for RAG: Retrieval Augmented Generation with Adaptive Reasoning Structures. In *AAAI*. AAAI Press, 30270–30278.

[9] Jindong Cui, Yuqing Wang, Zengchen Zhu, and Ruotong Li. 2025. A task decomposition and scheduling model for power IoT data acquisition with overlapping data efficiency optimization. *Scientific Reports* 15, 1 (2025), 17563.

[10] Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. 2025. Reforce: A Text-to-SQL agent with self-refinement, format restriction, and column exploration. In *ICLR 2025 Workshop: VerifAI: AI Verification in the Wild*. OpenReview.net.

[11] Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. 2024. Combining Small Language Models and Large Language Models for Zero-Shot NL2SQL. *Proc. VLDB Endow.* 17, 11 (2024), 2750–2763.

[12] Mohammad Farsi, Karl Ratcliff, and Manuel Barbosa. 1999. An overview of controller area network. *Computing and Control Engineering* 10, 3 (1999), 113–120.

[13] Li Feng, Le Liu, Hongsheng Xu, Sumeng Gao, Hai Tang, Jianing Cui, Bin Zhang, and Yufeng Rao. 2025. Research on engine power-loss fault diagnosis method based on time-series data mining. *Scientific Reports* (2025).

[14] 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. *Proc. VLDB Endow.* 17, 5 (2024), 1132–1145.

[15] Satya Krishna Gorti, Ilan Gofman, Zhaoyan Liu, Jiapeng Wu, Noël Vouitsis, Guangwei Yu, Jesse C. Cresswell, and Rasa Hosseinzadeh. 2025. MSc-SQL: Multi-Sample Critiquing Small Language Models For Text-To-SQL Translation. In *NAACL (Long Papers)*. Association for Computational Linguistics, 2145–2160.

[16] Bernal Jiménez Gutiérrez, Yiheng Shu, Weijian Qi, Sizhe Zhou, and Yu Su. 2025. From RAG to Memory: Non-Parametric Continual Learning for Large Language Models. In *ICML (Proceedings of Machine Learning Research)*. PMLR / OpenReview.net.

[17] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2025. Next-Generation Database Interfaces: A Survey of LLM-Based Text-to-SQL. *IEEE Trans. Knowl. Data Eng.* 37, 12 (2025), 7328–7345.

[18] Tengjun Jin, Yoojin Choi, Yuxuan Zhu, and Daniel Kang. 2026. Text-to-SQL Benchmarks are Broken: An In-Depth Analysis of Annotation Errors. In *CIDR*. www.cidrdb.org.

[19] George Katsogiannis-Meimarakis, Katsiaryna Mirylenka, Paolo Scotton, Francesco Fusco, and Abdel Labbi. 2026. In-depth Analysis of LLM-based Schema Linking. In *EDBT*. OpenProceedings.org, 117–130.

[20] Shreeram V Kulkarni, G Arjun, Sandeep Gupta, Raj Sinha, and Anand Shukla. 2025. Advanced battery diagnostics for electric vehicles using CAN based BMS data with EKF and data driven predictive models. *Scientific Reports* 15, 1 (2025), 32848.

[21] Brooke Lampe and Weizhi Meng. 2024. can-train-and-test: A curated CAN dataset for automotive intrusion detection. *Comput. Secur.* 140 (2024), 103777.

[22] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *NeurIPS*.

[23] Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. 2025. Alpha-SQL: Zero-Shot Text-to-SQL using Monte Carlo Tree Search. In *ICML (Proceedings of Machine Learning Research)*. PMLR / OpenReview.net.

[24] Guoliang Li, Ji Sun, James Pan, Jiang Wang, Yongqing Xie, Ruicheng Liu, and Wen Nie. 2025. GaussDB-Vector: A Large-Scale Persistent Real-Time Vector Database for LLM Applications. *Proc. VLDB Endow.* 18, 12 (2025), 4951–4963.

[25] Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. 2025. OmniSQL: Synthesizing High-quality Text-to-SQL Data at Scale. *Proc. VLDB Endow.* 18, 11 (2025), 4695–4709.

[26] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan 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 *NeurIPS*.

[27] Hongao Liu, Chang Li, Xiaosong Hu, Jinwen Li, Kai Zhang, Yang Xie, Ranglei Wu, and Ziyou Song. 2025. Multi-modal framework for battery state of health evaluation using open-source electric vehicle data. *Nature Communications* 16, 1 (2025), 1137.

[28] Kyle Luoma and Arun Kumar. 2025. SNAILS: Schema Naming Assessments for Improved LLM-Based SQL Inference. *Proc. ACM Manag. Data* 3, 1 (2025), 77:1–77:26.

[29] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan Ö. Arik. 2025. CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL. In *ICLR*. OpenReview.net.

[30] Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In *NeurIPS*.

[31] Mohammadreza Pourreza and Davood Rafiei. 2024. DTS-SQL: Decomposed Text-to-SQL with Small Large Language Models. In *EMNLP (Findings) (Findings of ACL)*. Association for Computational Linguistics, 8212–8220.

[32] Kiran Pradeep, Kirushikesh D. B., Nishtha Madaan, Sameep Mehta, and Pushpak Bhattacharyya. 2025. Divide, Link, and Conquer: Recall-oriented Schema Linking for NL-to-SQL via Question Decomposition. In *EMNLP (Industry Track)*. Association for Computational Linguistics, 1727–1743.

[33] Ritu Rai, Jyoti Grover, Prinkle Sharma, and Ayush Pareek. 2025. Securing the CAN bus using deep learning for intrusion detection in vehicles. *Scientific Reports* 15, 1 (2025), 13820.

[34] Liang Shi, Zhengju Tang, Nan Zhang, Xiaotong Zhang, and Zhi Yang. 2026. A Survey on Employing Large Language Models for Text-to-SQL Tasks. *ACM Comput. Surv.* 58, 2 (2026), 54:1–54:37.

[35] S Sivabalan and R Minu. 2022. Edge AI for industrial IoT applications. In *Applied Edge AI*. Auerbach Publications, 147–170.

[36] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive - A Warehousing Solution Over a Map-Reduce Framework. *Proc. VLDB Endow.* 2, 2 (2009), 1626–1629.

[37] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. In *COLING*. Association for Computational Linguistics, 540–557.

[38] Fabian Wenz, Omar Bouattour, Devin Yang, Justin Choi, Cecil Gregg, Nesime Tatbul, and Çagatay Demiralp. 2026. BenchPress: A Human-in-the-Loop Annotation System for Rapid Text-to-SQL Benchmark Curation. In *CIDR*. www.cidrdb.org.

[39] Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. OpenSearch-SQL: Enhancing Text-to-SQL with Dynamic Few-shot and Consistency Alignment. *Proc. ACM Manag. Data* 3, 3 (2025), 194:1–194:24.

[40] Yicun Yang, Zhaoguo Wang, Yu Xia, Zhuoran Wei, Haoran Ding, Ruzica Piskac, Haibo Chen, and Jinyang Li. 2025. Automated Validating and Fixing of Text-to-SQL Translation with Execution Consistency. *Proc. ACM Manag. Data* 3, 3 (2025), 134:1–134:28.

[41] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *EMNLP*. Association for Computational Linguistics, 3911–3921.

[42] Xiang Zhang, Le Zhou, Hongming Xu, Wei Zhou, Xuanhe Zhou, Guoliang Li, Yuyu Luo, Changdong Liu, Guorun Chen, Jiang Liao, and Fan Wu. 2026. Dial: A Knowledge-Grounded Dialect-Specific NL2SQL System. *Proc. VLDB Endow.* 19, 11 (2026), 3718–3731.

[43] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. 2024. Chat2Data: An Interactive Data Analysis System with RAG, Vector Databases and LLMs. *Proc. VLDB Endow.* 17, 12 (2024), 4481–4484.

[44] Wei Zhou, Yuyang Gao, Xuanhe Zhou, and Guoliang Li. 2025. Cracking SQL Barriers: An LLM-based Dialect Translation System. *Proc. ACM Manag. Data* 3, 3 (SIGMOD) (2025).

[45] Wei Zhou, Yuyang Gao, Xuanhe Zhou, and Guoliang Li. 2026. CrackSQL: A Hybrid Dialect Translation System Powered by LLM. In *SIGMOD Companion*. ACM, 154–157.

[46] Wei Zhou, Guoliang Li, Haoyu Wang, Yuxing Han, Xufei Wu, Fan Wu, and Xuanhe Zhou. 2025. PARROT: A Benchmark for Evaluating LLMs in Cross-System SQL Translation. In *NeurIPS*.

[47] Wei Zhou, Chen Lin, Xuanhe Zhou, and Guoliang Li. 2024. Breaking It Down: An In-depth Study of Index Advisors. *Proc. VLDB Endow.* 17, 10 (2024), 2405–2418.

[48] Wei Zhou, Peng Sun, Xuanhe Zhou, Qianglei Zang, Ji Xu, Tieying Zhang, Guoliang Li, and Fan Wu. 2026. DBAIOps: A Reasoning LLM-Enhanced Database Operation and Maintenance System using Knowledge Graphs. *Proc. VLDB Endow.* 19, 6 (2026), 1319 – 1331.

[49] Wei Zhou, Jun Zhou, Haoyu Wang, Zhenghao Li, Qikang He, Shaokun Han, Guoliang Li, Xuanhe Zhou, Yeye He, Chunwei Liu, Zirui Tang, Bin Wang, Shen

Tang, Kai Zuo, Yuyu Luo, Zhenzhe Zheng, Conghui He, Jingren Zhou, and Fan Wu. 2026. Can LLMs Clean Up Your Mess? A Survey of Application-Ready Data Preparation with LLMs. *arXiv preprint* https://arxiv.org/abs/2601.17058 (2026).

[50] Wei Zhou, Xuanhe Zhou, Shaokun Han, Hongming Xu, Guoliang Li, Zhiyu Li, Feiyu Xiong, and Fan Wu. 2026. Are We Ready For An Agent-Native Memory System? *arXiv preprint* (2026). https://arxiv.org/abs/2606.24775

[51] Wei Zhou, Xuanhe Zhou, Qikang He, Guoliang Li, Bingsheng He, Chao Peng, Yun Lin, and Fan Wu. 2026. DBCooker: A Database Kernel-Specialized Coding Agent Harness. *Proc. VLDB Endow.* 19, 12 (2026), 4842–4845.

[52] Wei Zhou, Xuanhe Zhou, Qikang He, Guoliang Li, Bingsheng He, Quanqing Xu, and Fan Wu. 2026. Automating Database-Native Function Code Synthesis with LLMs. *Proc. ACM Manag. Data* 4, 3 (2026), 141:1–141:26.

[53] Xuanhe Zhou, Junxuan He, Wei Zhou, Haodong Chen, Zirui Tang, Haoyu Zhao, Xin Tong, Guoliang Li, Youmin Chen, Jun Zhou, Zhaojun Sun, Binyuan Hui, Shuo Wang, Conghui He, Zhiyuan Liu, Jingren Zhou, and Fan Wu. 2025. A Survey of LLM x DATA. *arXiv preprint* https://arxiv.org/abs/2505.18458 (2025).

[54] Xuanhe Zhou, Wei Zhou, Liguo Qi, Hao Zhang, Dihao Chen, Bingsheng He, Mian Lu, Guoliang Li, Fan Wu, and Yuqiang Chen. 2025. OpenMLDB: A Real-Time Relational Data Feature Computation System for Online ML. In *SIGMOD Conference Companion*. ACM, 729–742.

[55] Yingli Zhou, Yaodong Su, Youran Sun, Shu Wang, Taotao Wang, Runyuan He, Yongwei Zhang, Sicong Liang, Xilin Liu, Yuchi Ma, and Yixiang Fang. 2025. In-depth Analysis of Graph-based RAG in a Unified Framework. *Proc. VLDB Endow.* 18, 13 (2025), 5623–5637.

[56] Luyao Zhuang, Shengyuan Chen, Yilin Xiao, Huachi Zhou, Yujing Zhang, Hao Chen, Qinggang Zhang, and Xiao Huang. 2026. LinearRAG: Linear Graph Retrieval Augmented Generation on Large-scale Corpora. In *ICLR*. OpenReview.net.