Automating Database-Native Function Code Synthesis with LLMs

WEI ZHOU, Shanghai Jiao Tong University, China
XUANHE ZHOU*, Shanghai Jiao Tong University, China
QIKANG HE, Shanghai Jiao Tong University, China
GUOLIANG LI, Tsinghua University, China
BINGSHENG HE, National University of Singapore, Singapore
QUANQING XU, Ant Group, China
FAN WU, Shanghai Jiao Tong University, China

Database systems incorporate an ever-growing number of functions built in their kernels (a.k.a., database native functions) for scenarios like new application support and business migration. This growth causes an urgent demand for automatic database native function synthesis. While recent advances in LLM-based code generation (e.g., Claude Code) show promise, existing approaches are too generic for database-specific development. They often hallucinate or overlook critical context because database function synthesis is inherently complex and error-prone, where synthesizing a single database function may involve registering multiple function units (e.g., for different input types), placing code in the correct source files, linking internal references, and implementing logic correctly.

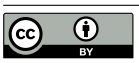
To this end, we propose ***DBCooker***, an LLM-based system for automatically synthesizing database native functions. The system consists of three key components. First, the *function characterization module* aggregates multi-source declarations, identifies function units that require specialized coding through hierarchical analysis, and traces cross-unit dependencies via static analysis. Second, we design operations to address the main synthesis challenges: (1) a *pseudo-code–based coding plan generator* that constructs structured implementation skeletons by identifying key elements such as reusable referenced functions; (2) a *hybrid fill-in-the-blank model* guided by probabilistic priors and component awareness to integrate core logic with reusable routines; and (3) *three-level progressive validation*, including syntax checking, standards compliance, and LLM-guided semantic verification. Finally, an *adaptive orchestration strategy* unifies these operations with existing database tools and dynamically sequences them based on the orchestration history of similar functions. Results show that our system outperforms state-of-the-art methods on SQLite, PostgreSQL, and DuckDB (34.55% higher accuracy on average), and can synthesize four categories of new functions absent in the latest SQLite (v3.50). The code is available at *https://github.com/weAIDB/DBCooker*.

CCS Concepts: • **Information systems → Data management systems**.

Additional Key Words and Phrases: Database Function, Function Code Synthesis, Large Language Models

*Xuanhe Zhou is the corresponding author.

Authors' Contact Information: Wei Zhou, Shanghai Jiao Tong University, China, weizhoudb@sjtu.edu.cn; Xuanhe Zhou, Shanghai Jiao Tong University, China, zhouxuanhe@sjtu.edu.cn; Qikang He, Shanghai Jiao Tong University, China, hqk0205@sjtu.edu.cn; Guoliang Li, Tsinghua University, China, liguoliang@tsinghua.edu.cn; Bingsheng He, National University of Singapore, Singapore, dcsheb@nus.edu.sg; Quanqing Xu, Ant Group, China, xuquanqing.xqq@oceanbase.com; Fan Wu, Shanghai Jiao Tong University, China, fwu@cs.sjtu.edu.cn.



ACM 2836-6573/2026/6-ART141
https://doi.org/10.1145/3802018

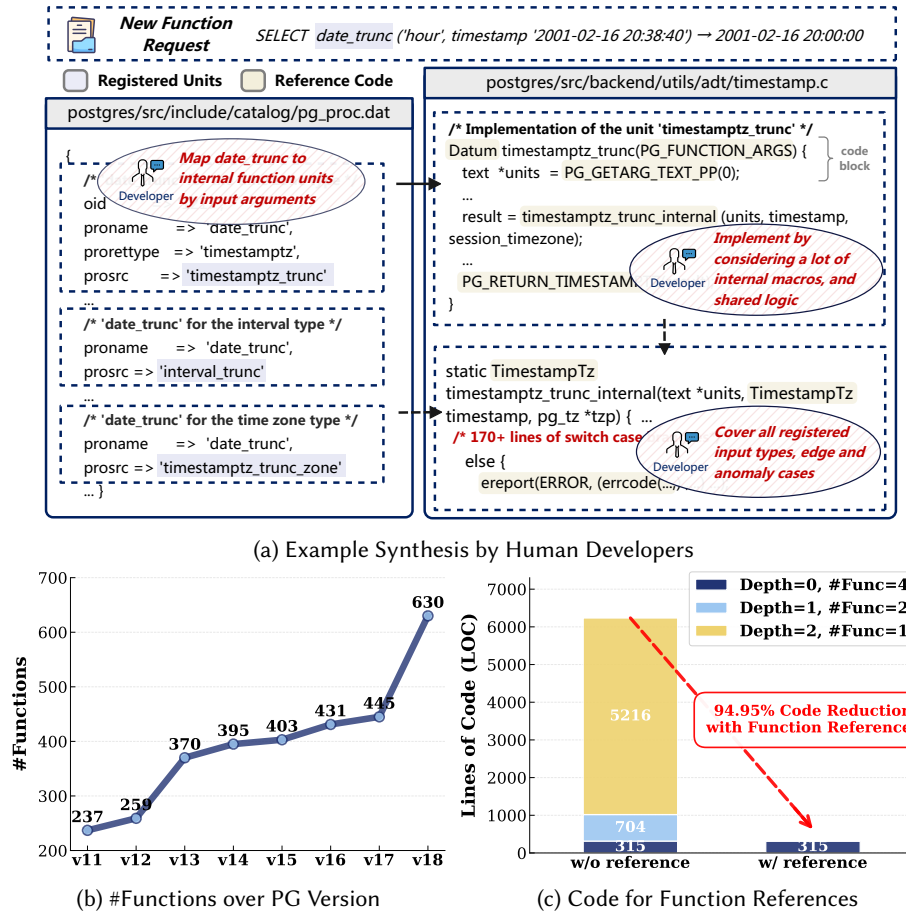
ACM Reference Format:
Wei Zhou, Xuanhe Zhou, Qikang He, Guoliang Li, Bingsheng He, Quanqing Xu, and Fan Wu. 2026. Automating Database-Native Function Code Synthesis with LLMs. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 141 (June 2026), 26 pages. https://doi.org/10.1145/3802018

1 Introduction

Databases offer numerous native SQL functions in their kernels. For instance, PostgreSQL v18 includes 27 date functions such as *date_trunc()*, and DuckDB v1.4.0 offers 57 numeric functions such as *sqrt()*. The number of native SQL functions has been steadily increasing in modern database systems. For example, as shown in Figure 1 (b), PostgreSQL functions nearly tripled from 237 (v11) to 630 (v18) [14], DuckDB grew from 60 (v0.3.3) to 666 (v1.4.0) [6], and SQLite increased from 52 (v3.8.0) to 143 (v3.50.0) [16]. This expansion is driven by new scenario support (e.g., BI analysis [18, 52, 54, 55] and geometric processing [35]) and business migration [24, 49, 53]. Specifically, in legacy migration scenarios (e.g., Oracle to PostgreSQL), implementing proprietary functions is a major bottleneck [11, 12, 56], with code refactoring accounting for 30%–60% of migration budgets and requiring 40–80 hours per 1,000 code lines [10].

Synthesizing database native functions is a critical task for extending system capabilities, as evidenced by the official development guides provided by major systems such as PostgreSQL [5, 13]. However, it is a tightly coupled, error-prone process that demands extensive human expertise [22] and a deep understanding of internal dependencies and differences across major database updates [4, 8], imposing a hard-to-sustain burden on developers. The complexity is substantial: PostgreSQL native function codes span 119,161 lines (within two dependency hops) from v11 to v18, and the DuckDB GitHub repository reports 3,791 function-related issues [7]. As shown in Figure 1 (a), implementing the SQL function *date_trunc()* in PostgreSQL requires developers to (1) register appropriate function units (e.g., *timestamptz_trunc* and *interval_trunc*) based on factors such as argument types specified in the *prorettype* attribute; (2) search for the correct source files to implement the function units (e.g., *timestamp.c* for *timestamptz_trunc*); and (3) reference the correct units (e.g., *PG_GETARG_TEXT_PP*) to complete the code blocks in these function units. Failing to utilize these references, such as misusing some internal data types, can result in standard violations (causing synthesis failures) or re-implementation waste. As shown in Figure 1 (c), implementing only the four registered units of *date_trunc()* using reference units reduces the required code lines by approximately 94.95% compared to implementing it from scratch, which would require 6,235 lines of code and 225 functions across two-hop invocations.

Limitations of Existing Methods. Despite recent advances in LLM-based general-purpose code generation [27], to our knowledge, there is no publicly available tool or framework that provides a fully automated pipeline for database native functions. Existing code synthesis methods, including prompt-based approaches [20, 45], agent-based systems [31, 43], and training-based models [21, 29, 34], demonstrate significant limitations in database native function synthesis. First, these general-purpose methods fail to capture the specific characteristics of database native functions, such as required function units and fine-grained references within the repository, resulting in incorrect or incomplete implementations. For example, Qwen Code [26] may omit a function unit needed to handle interval inputs in *date_trunc()*. Second, they rely heavily on coarse-grained file search operations without domain-specific heuristics to guide code placement, and overlook essential verification steps. For instance, Claude Code [1] spends most of its time scanning unrelated files to locate where to implement *date_trunc()*, rather than validating its correctness across different input values. Third, existing methods use a static synthesis strategy that starts with file exploration and implements functions from scratch, without accounting for differences in function complexity or leveraging similarities across functions. For example, they treat a simple math function *sqrt()*



(a) Example Synthesis by Human Developers

(b) #Functions over PG Version

(c) Code for Function References

Fig. 1. Database native function code synthesis is a complex problem – (a) Example synthesis workflow by human developers; (b) Increasing tendency of function number across versions; (c) Code size comparison with vs. without reference usage.

and a complex aggregate function *json_agg()* identically, and fail to reuse related implementations such as *date_part()* of the same date function category (see analysis in Section 2.3).

Challenges. It presents three main challenges. First, a high-level SQL function may correspond to multiple underlying function units with distinct names and responsibilities, making it non-trivial to determine which units need to be synthesized. For example, in PostgreSQL, *date_trunc()* needs to register functions like *timestamptz_trunc* and *interval_trunc* separately for input arguments in timezone and interval (**C1**). Second, implementing a database native function requires referencing a vast number of existing function units, without which the implementation procedure cannot be correctly synthesized, because (1) the references are complex and cannot easily generate from scratch (e.g., the lines of code for *data_trunc()* references within two hops increase from 315 to 6,235 lines in total); and (2) some references are essential for functionality (e.g., output formatting using macros *PG_RETURN_NUMERIC* in PostgreSQL) (**C2**). Third, generalizing synthesis across diverse functions is difficult. Simple functions such as *sqrt()* can be implemented by directly wrapping

standard libraries, whereas high-complexity aggregate functions, such as *json_agg()*, require custom logic, necessitating adaptive synthesis strategies (**C3**).

Our Methodology. To address these challenges, we propose ***DBCooker***, an automatic LLM-based database function synthesis system. First, to accurately capture function composition and key information, we introduce the *function characterization* module with (1) multi-source function declaration collection (e.g., documents and system catalogs), (2) hierarchical function unit identification to isolate distinctive units (require implementation), and (3) context-aware cross-unit reference analysis using static dependency graphs and category-specific pruning (**for C1**). Second, to improve synthesis accuracy and correctly utilize references, we design *synthesis operations*, including (1) format-based pseudo-plan generation, creating and scoring structured code skeletons to guide generation, (2) a hybrid fill-in-the-blank coding model enhanced with probabilistic priors and component awareness for precise incorporation of distinctive and reusable reference units, and (3) a *three-level progressive validation* module, covering syntax, compliance, and LLM-guided checking (**for C2**). Finally, to handle functions of varying complexity, we propose an *adaptive synthesis operation orchestration* module that abstracts operations as tools and dynamically optimizes the tool calling workflow using a combination of LLM-driven decision making and similar workflow trajectories (**for C3**).

Contributions. In summary, we make the following contributions.

- We propose an LLM-based system for automatic synthesis of database function code. *To our best knowledge, it is the first system that analyzes function composition, generates required internal references and linkages across functional units, and adaptively synthesizes various database functions* (Section 3).
- We propose a function characterization module that captures key implementation information via declaration extraction, distinct unit identification, and cross-unit reference analysis (Section 4).
- We propose a distinctive function unit code synthesis strategy via format-based pseudo-plan generation, hybrid fill-in-the-blank model guided coding, and progressive code validation (Section 5).
- We employ an adaptive diverse-function code synthesis mechanism that dynamically orchestrates synthesis operations via a hybrid optimization strategy combining LLM-based decision making and trajectory-based workflow reuse (Section 6).
- We conduct extensive experiments to assess different synthesis methods over three mainstream databases and ***DBCooker*** outperforms state-of-the-art methods (e.g., Claude Code [1]) with 34.55% higher accuracy on average and can add completely new functions absent in the latest SQLite (v3.50) (Section 7). The code is available at *https://github.com/weAIDB/DBCooker*.

2 Preliminary

In this section, we first introduce database native functions, and then formalize the problem of database function code synthesis.

2.1 Database Native Functions

Database native functions provide a wide range of functionalities (e.g., string manipulation, numeric computation) and can be decomposed into three parts: (1) *function declaration*, (2) *new function units to be implemented*, and (3) *existing function units used as references*.

Function Declaration. Given a database engine $\mathcal{D}$, a *function declaration* $\boldsymbol{f_{dec}}$ formally specifies the function's existence and usage, including the name, description, argument types, and return type. The declaration is typically stored in the system catalog to support consistent registration and invocation. For example, the declaration of *date_trunc()* describes its functionality "truncate timestamp", and specifies the two input arguments (*text, timestamp*) in *pg_proc.dat*.

Function Unit. A function unit f_{unit} is a self-contained executable component with one or multiple code blocks $\{c_{block}\}$ defined in specific database files. Each code block (i.e., continuous code lines) encapsulates a specific aspect of the function's behavior, ranging from its main computational logic to auxiliary tasks (e.g., process the input arguments or format the output results).

For function code synthesis, given an SQL function, we distinguish between the newly synthesized function units f_{unit}^{new} (which need to be implemented) and the reference function units f_{unit}^{ref} (which are already implemented, such as library functions or macros).

For example, the *date_trunc()* function owns newly synthesized function units (e.g., *timestamptz_trunc*) defined in *timestamp.c*, and multiple reference function units (e.g., *PG_GETARG_TEXT_PP*).

Database Native Function. Different from functions like UDFs defined in SQL level, the database native function f is fully implemented by one or multiple function units and directly integrated into the database $\mathcal{D}$, which can be formally represented as below.

DEFINITION 2.1 (Database Native Function). *A database native function f defined in a database $\mathcal{D}$ is represented as $\langle f_{dec}, \{\langle f_{unit}^{new}, f_{unit}^{ref}\}\rangle\rangle$, which is composed of one or multiple functions units $\{f_{unit}^{new}\}$, where each f_{unit}^{new} can invoke a set of external referenced function units $\{f_{unit}^{ref}\}$ (e.g., internal modules, macros), and is exposed through a unified SQL-level function interface defined in function declaration f_{dec}.*

Example 2.1. Database native function *date_trunc()* in PostgreSQL includes: (1) f_{dec} that specifies the signature with two input arguments (*text, timestamp*) and one output argument *timestamp*; (2) f_{unit} with four newly synthesized function units (i.e., *timestamp_trunc*, *interval_trunc*, *timestamptz_trunc*, and *timestamptz_trunc_zone*) defined in *src/backend/utils/adt/timestamp.c* to handle different inputs, and multiple reference function units (e.g., *PG_GETARG_TEXT_PP* to get the input arguments, and *PG_RETURN_TIMESTAMPTZ* to format output results).

2.2 Database Native Function Code Synthesis

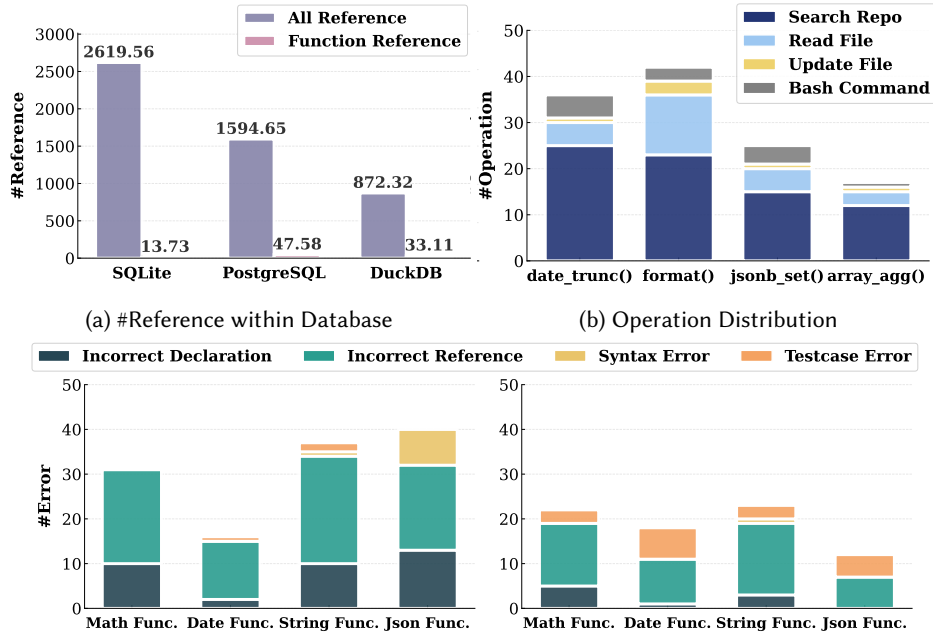
We next define the problem of *database native function code synthesis*. Given a precise *function specification* (e.g., function name, description, input arguments, output types, and SQL examples), the goal is to automatically generate and implement executable function codes within the database kernel such that the synthesized code correctly realizes the intended functionality.

DEFINITION 2.2 (Database Native Function Synthesis). *Given a SQL-level function specification $\mathcal{S}$ for the target database $\mathcal{D}$, database native function synthesis aims to generate the codes of necessary function units $\{f_{unit}^{new}\}$ that satisfy $\mathcal{S}$ and can be successfully integrated into $\mathcal{D}$ with all the essential references $\{f_{unit}^{ref}\}$, i.e., without any compliance error and passing all the test-cases $\mathcal{T}$ in database $\mathcal{D}$ with expected results.*

Example 2.2. To synthesize the date_trunc() *function, the input declaration includes (1) the code-style function declaration with the name and the input arguments, (2) the textual declaration with the usage descriptions, and SQL examples. Accordingly, we first identify the reference function units such as* PG_FUNCTION_ARGS *for processing input arguments. We then implement multiple new function units of key processing, such as* timestamptz_trunc() *and* timestamp_trunc(), *which handle input validation and output construction. Once all the function units are correctly implemented, the function can be executed using the SQL "*SELECT date_trunc('hour', timestamp '2001-02-16 20:38:40')*", producing the result "2001-02-16 20:00:00".*

2.3 Pilot Study

We characterize database native function codes and investigate the limitations of existing methods.



(a) #Reference within Database

(b) Operation Distribution

(c) Error Distribution (Claude Sonnet 4.5 [3] and Claude Code [1])

Fig. 2. Native database function synthesis is hindered by dense but sparsely used references, inefficient operations, and diverse errors – (a) All references versus those in native functions; (b) Operation distribution of SOTA agent-based methods (i.e., Claude Code [1]); (c) Error distribution of LLM-based (Claude Sonnet 4.5 [3]) and agent-based methods (Claude Code [1]).

O1. Databases contain a large number of references due to their inherent complexity, while native functions depend on a small set of relevant references for their own implementation. We analyze the total number of references across the entire database repository and compare them with the references directly utilized by native functions. As shown in Figure 2 (a), we notice that although database repositories contain a vast number of file and function references, native functions depend on only a limited portion of them. Specifically, the average number of references per file in the three database repositories is 2619.56, 1594.65, and 872.32, respectively, while the corresponding averages for a single function unit are only 13.73, 47.58, and 33.11. This high reference density underscores the need for effective methods to accurately identify the correct reference functions during the synthesis process.

O2. Current agent-based frameworks employ a generic design with limited focus on database-specific operations, leading to inefficient exploration and synthesis. Most existing code synthesis frameworks, including state-of-the-art agent-based approaches, are general-purpose systems designed for diverse code repositories [28]. However, they often overlook the structural characteristics of database projects (e.g., function definitions in files such as *pg_proc.dat* in PostgreSQL), which hinder the efficient synthesis of database native functions. As shown in Figure 2 (b), Claude Code [1] spends most of its time on file-level searches (63.70% on *Search Repo* and *Read File*) instead of code generation tasks (4.95% on *Update File*), which implement the function logic. This imbalance highlights that current frameworks focus too much on repository traversal, neglecting code construction and validation. Therefore, there is a need for database-aware synthesis strategies that can efficiently navigate repository structures while maintaining a balanced focus on both code generation and correctness.

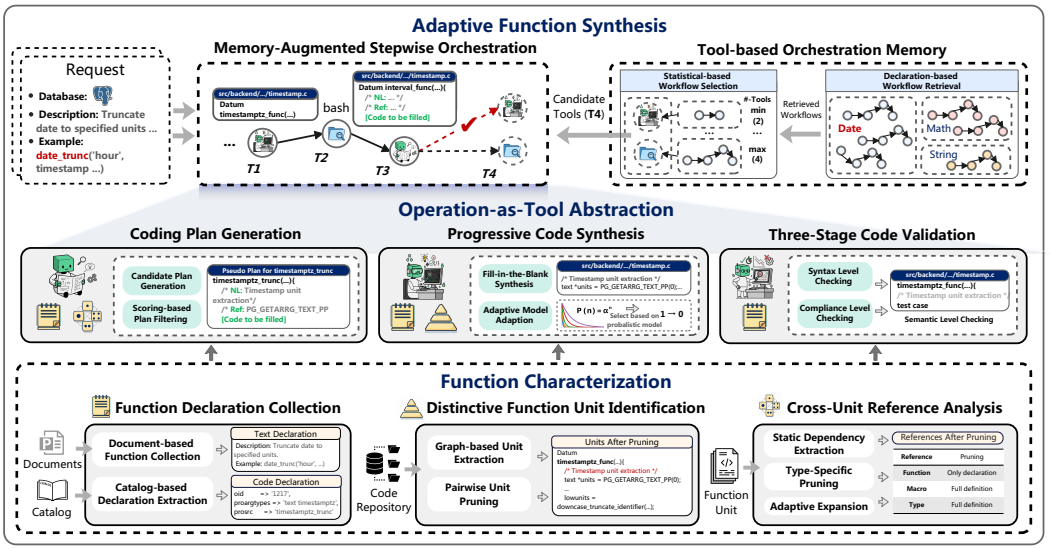


Fig. 3. Technique Overview of ***DBCooker***.

O3. Existing synthesis methods exhibit various errors, with declaration errors being the most frequent in synthesizing database functions. We analyze the error types and frequencies produced by an advanced LLM (Claude Sonnet 4.5 [3]) and an agent-based framework (Claude Code [1]) when synthesizing PostgreSQL functions. As shown in Figure 2 (c), both methods, especially the advanced LLM, produce many declaration-related errors. These include redeclaration within the same file (e.g., redefining *int4div* in *div*) and incorrect function references with mismatched arguments (e.g., too few arguments to *pg_regcomp* in *regexp_matches*). On average, declaration errors account for 81.76% of all errors in both methods. We also observe test-case errors in which generated functions fail to produce the expected outputs. These results highlight the need for improved module designs to reduce declaration issues, unresolved references, and test case failures during synthesis.

3 *DBCooker* Overview

In this section, we introduce the system overview (see Figure 3). In contrast to general-purpose code synthesis frameworks [1, 23], ***DBCooker*** is designed with explicit considerations of database codebase characteristics, enabling robust handling of their unique structural organizations (e.g., the strict mapping between SQL-level catalog definitions and their underlying implementation units).

Function Characterization. This module is designed around how database functions are implemented, forming structural templates and constraints that capture the internal organization to guide synthesis. Taking the database codebase as input, we introduce automatic strategies to extract key metadata as output (defined in Definition 2.1). *(1) Function Declaration Collection.* Function declarations are useful (e.g., function name, arguments) for code reuse (e.g., referenced units across similar functions) during synthesis. To address the SQL-level declaration of database functions, we take official documentation and system catalogs (e.g., *pg_proc*) as input and collect declarations in two ways: by parsing official documentation to retrieve structured function data and by querying system catalogs (e.g., *pg_proc* in PostgreSQL) to gather authoritative declarations directly from the database engine; *(2) Distinctive Function-Unit Identification.* To address the database-specific mappings from a single SQL function to multiple internal units (e.g., 4 date types for the PostgreSQL

date_trunc() function), we adopt a graph-based analysis strategy. Using the function name as input, we identify reusable units by locating potential registration points via keyword matching, then traversing code paths in the dependency graph (e.g., constructed by static analysis tools [17]) to collect related units, and distinguishing shared auxiliary units (reusable as templates) from distinctive ones through pairwise comparisons; *(3) Cross-Unit Reference Analysis.* To address the complex cross-unit dependencies within the database codebase and reuse these references for accurate unit synthesis, we propose a context-aware method. Taking the initial dependency graph as input, we first collect dependencies and then refine them using category-specific rules to retain only essential content (e.g., keep declarations, omit full method bodies), which can later be restored during synthesis.

Function Synthesis Operations. This module builds on the extracted function characters and includes specialized synthesis operations to ensure database correctness. It involves three operations: planning, coding, and testing for native function synthesis. *(1) Pseudo-based Coding Plan Generation.* To guide native function synthesis that follows registration rules and internal structures, we propose pseudo-plan generation to create skeletons. Taking function metadata and database hierarchy as input, we identify relevant reference units via meta-information matching from similar functions. Then, we instruct LLMs to generate multiple candidate plans (i.e., structured code skeletons) using these units, with placeholders for variables, function bodies, and return expressions. A score-based pruning mechanism ranks these candidates to output the optimal plan based on database code brevity and convention adherence; *(2) Progressive Code Synthesis.* We model function synthesis as a fill-in-the-blank problem, where each blank corresponds to a required code unit. A two-step strategy is employed: (a) a template instantiation mechanism retrieves partially matched templates based on declaration types; (b) in cases of failures, a probabilistic adaptation module controls the synthesis, allowing either blank-filling or generation from scratch to output the synthesized code; *(3) Three-Stage Code Validation.* To ensure the compilation and correct database execution of synthesized functions, we introduce a three-tier progressive validation framework that takes the synthesized code as input: *(a) Syntax validation* uses grammar parsers (e.g., ANTLR) to ensure parseability under the target database; *(b) Compliance checking* uses database-native compilation tools to verify the function adheres to internal conventions (e.g., registration consistency); *(c) Semantic validation* augments the database's test suite with auto-generated test cases across input types and edge cases.

Adaptive Function Synthesis. To address the limitations of rigid "coding → testing" workflows for functions of varying complexity, we first unify diverse synthesis steps, including both common utilities (e.g., *bash*) and LLM-based tools, by abstracting them into a set of callable tools. Then, taking the synthesis task as input, we develop a hybrid orchestration strategy that combines an LLM-based controller, which adaptively determines the next tool through context-aware reasoning, with a global trajectory memory that accumulates historical workflows to guide and accelerate synthesis with similar functions to output the final functions.

4 Function Characterization

In this section, we introduce the *Function Characterization* module. Unlike general-purpose agents that rely on inefficient file search [1, 15], this module performs deterministic graph-based analysis to decompose SQL functions into essential structural templates rather than raw snippets. It is designed to address three challenges inherent to database kernels. First, constructing a concise and informative overview of a function is non-trivial, as its definition is distributed across heterogeneous textual and code-level declarations. Second, identifying the essential function units requires understanding the database's internal architectural conventions (e.g., type-specific components

and modular execution pipelines). Third, these function units often exhibit structural reuse and cross-unit dependencies (e.g., database-specific macros) to facilitate consistent implementations across similar functions.

4.1 Function Declaration Collection

To effectively represent functions with key information from the SQL keyword, we design a dual-source strategy to extract both the textual and the code-style function declarations f_{dec} systematically.

Document-based Function Collection. To capture the SQL-level semantics of native functions, we automatically parse and analyze official database documentation to collect textual function declarations. We identify sections describing native functions by examining the hierarchical document structure and employ automated scripts to parse and normalize the extracted content into a unified JSON format containing function-related information. For instance, from PostgreSQL's "*Chapter 9. Functions and Operators*", we extract the function *date_trunc*, described as "*truncate timestamp to specified precision*", with examples such as *date_trunc('hour', timestamp '2001-02-16 20:38:40')* → *2001-02-16 20:00:00*.

Catalog-based Declaration Extraction. To accurately capture implementation-level specifications, we extract definitive code-level declarations from the internal system catalogs and registration files (e.g., the *pg_proc* table and the *pg_proc.dat* file in PostgreSQL). We query and consolidate catalog entries, systematically retrieving key attributes such as input arguments and return types, and normalize them into the same JSON format. For example, querying PostgreSQL's system catalog for *date_trunc* returns an entry indicating two input arguments (*proargtypes = text timestamptz*) and a result type of *timestamptz* (*prorettype = timestamptz*).

4.2 Distinctive Function Unit Identification

To follow the implicit conventions for new database function synthesis, we design two modules to identify the essential function units f_{unit} that require implementation across multiple files.

Graph-based Unit Extraction. To uncover the underlying implementation of an encapsulated SQL function interface, we propose a graph-based unit extraction method that systematically identifies all associated function units f_{unit}. For a native database function f with SQL-level declaration f_{dec}, we first perform keyword-based retrieval to locate the corresponding function entry and its registration unit f_{unit}^{reg}. Starting from f_{unit}^{reg}, we apply automated static analysis to construct a reference graph $\mathcal{G}_f = (\mathcal{V}_f, \mathcal{E}_f)$, where $\mathcal{V}_f$ denotes the set of function units f_{unit} and $\mathcal{E}_f$ encodes references through function invocations or class inheritance relations. The graph expansion proceeds recursively and terminates when no additional function units are discovered or when the number of current units being referenced is larger than a predefined threshold.

As shown in Figure 4 (a), we initialize a dependency graph in DuckDB from the SQL keyword *date_part* to locate the registration entry in *function_list.cpp*. The graph is then expanded via reference traversal to identify key functional units, such as *DatePartFun* in *date_functions.hpp*, while *function_set.hpp* is excluded since the involved component *ScalarFunctionSet* is a reference unit widely utilized across scalar functions.

Pairwise Unit Pruning. To identify common code patterns across function units and facilitate the synthesis of similar native functions, we introduce a pairwise unit pruning strategy that extracts pruned fixed patterns from the set of extracted function units $\{f_{unit}\}$.

(1) Grouping by Function Declarations: To facilitate the identification of pruned implementation patterns, the extracted function units f_{unit} are first grouped according to their SQL-level declarations f_{dec}, considering input-output argument types and functional categories. For example, the functions

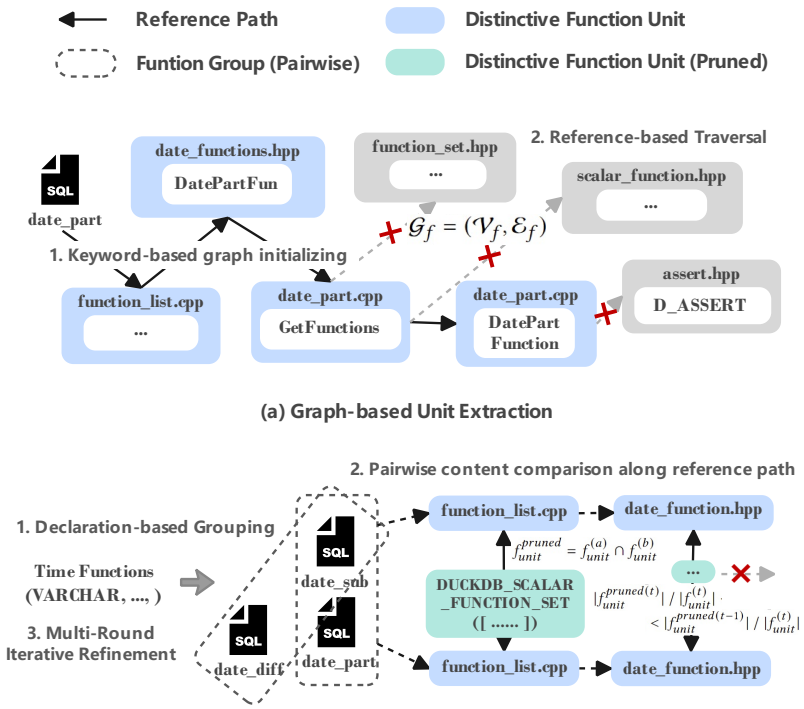


Fig. 4. Example Function Unit Identification.

timestamptz_part and *extract_timestamptz* are grouped together because they belong to the same datetime category and accept timestamp arguments (Figure 4).

(2) Pairwise Content Comparison: To effectively derive candidate function units with common patterns, units within each group are randomly selected in pairs and compared along their invocation paths in the reference graphs $\mathcal{G}_f = (\mathcal{V}_f, \mathcal{E}_f)$. For a pair of units $f_{unit}^{(a)}, f_{unit}^{(b)}$, identifiers such as variable names are first abstracted to normalize the code. The pruned code blocks are then extracted as $f_{unit}^{pruned} = f_{unit}^{(a)} \cap f_{unit}^{(b)}$ based on the exact keyword matching. Non-pruned blocks are replaced with placeholders to represent variability across implementations.

(3) Multi-Round Refinement: To robustly determine representative pruned units f_{unit}^{pruned}, multiple rounds of pairwise comparison are performed within each group, bounded by the group size. Each round t stops when the proportion of pruned components along the code reference paths decreases compared to the previous round $t-1$, i.e., $|f_{unit}^{pruned(t)}| \,/\, |f_{unit}^{(t)}| < |f_{unit}^{pruned(t-1)}| \,/\, |f_{unit}^{(t)}|$.

As shown in Figure 4 (b), we first classify *date_sub()* function based on its declaration of *Time Functions*. We then randomly select *date_sub()* and *date_part()* from the same group for pairwise comparison. Along their graph reference paths, we prune units to derive *DUCKDB_SCALAR_FUNCTION_SET(...)* in *function_list.cpp*, terminating at *date_functions.hpp* when pruned units become significantly smaller. Multi-round comparisons with other function pairs, such as *date_sub()* and *date_diff()*, are conducted to robustly extract diverse pruned units. Finally, the top-k most frequently occurring pruned units f_{unit}^{pruned} across comparisons are selected as representative pruned function units, capturing stable and reusable structures for the fill-in-the-blank synthesis model in Section 5.2.

4.3 Cross-Unit Reference Analysis

To systematically capture dependencies among function units within complex database repositories, we propose cross-unit reference analysis to extract reference units f_{unit}^{ref} for each native function f.

(1) Static Dependency Extraction: Given the reference graph $\mathcal{G}_f = (\mathcal{V}_f, \mathcal{E}_f)$, we iteratively perform automated static analysis to identify the reference units $\{f_{unit}^{ref}\}$ for each function unit $f_{unit} \in \mathcal{V}_f$ along the graph paths, including assertion references *D_ASSERT* and executor references *BinaryExecutor* in *DatePartFunction*.

(2) Type-Specific Pruning: To address the redundancy or excessive details introduced by directly including all reference units (e.g., complete class definitions), we apply predefined type-specific pruning rules that retain only the essential contextual information while preserving core structural elements. For example, the *ScalarFunctionSet* class retains its declaration list, while detailed method implementations, such as *GetFunctionByArguments*, are omitted.

(3) Adaptive Expansion: The resulting reference set provides an initial representation of f_{unit}^{ref}, which can be incrementally expanded during function synthesis. Additional references can be added on demand by using automated static analysis tools to retrieve the full content of reference units, such as the complete implementation of the *GetFunctionByArguments* method.

5 Function Code Synthesis Operations

Existing code synthesis frameworks [1, 15, 23] have limitations in utilizing the characterized database function information. First, they rely on LLMs to directly generate code using bash commands and file-related tools, which are difficult to identify complex function relations, such as referenced macros and internal units in existing functions (Section 4.3). Second, they generate code from scratch, whereas many database functions are large and complex (e.g., 6,235 lines across 225 functions for *data_trunc()*) and could be partially reused to reduce synthesis effort (Section 4.2). Third, they depend on LLM's internal knowledge for validation, but database functions often take diverse arguments whose edge cases require careful handling to ensure test coverage. To address these, we enhance database function synthesis with three key LLM-enhanced operations.

5.1 Pseudo-based Coding Plan Generation

As illustrated in Figure 1, human developers invest significant effort to map a SQL-level function to its internal units and determine their implementation logic and locations across files via repository exploration. Rather than relying on invisible LLM reasoning [46, 47], ***DBCooker*** defers code synthesis to an informative plan. We generate skeletons based on registered function units (for SQL-level functionality) and internal database hierarchy information. Each plan explicitly comprises: (1) the decomposed function units, and (2) the referenced units within the database.

5.1.1 Candidate Plan Generation

To identify relevant function units for an informative plan, we adopt a metadata-based matching strategy. Specifically, we first collect potential referenced function units from all the existing functions with the same function category. The referenced function units are then filtered to remove duplicates and grouped by unit type.

Considering the diverse processing logic required for different input types and the frequent reuse of existing units in database native functions, we define and instruct LLM to generate the candidate coding plans with: (1) the decomposed function units for each input type, as well as the structured processing logic within each unit, and (2) the potential referenced units in the database that support the implementation of each code block in the decomposed function units. As shown in Figure 5, the coding plan enumerates the required function units along with their file paths (e.g., *Datum*

src/backend/utils/adt/timestamp.c

```
Datum
timestamptz_trunc(PG_FUNCTION_ARGS) {
    /* Step 1: Extract function arguments
       Potential code elements: PG_GETARG_TEXT_PP(), PG_GETARG_TIMESTAMPTZ() */
    [ code to be filled ]

    /* Step 2: Call internal truncation function with session timezone
       Potential code elements: timestamptz_trunc_internal(), session_timezone */
    [ code to be filled ]

    /* Step 3: Return the truncated result
       Potential code elements: PG_RETURN_TIMESTAMPTZ() */
    [ code to be filled ]
}
```

Fig. 5. Example Pseudo-based Coding Plan.

timestamptz_trunc(PG_FUNCTION_ARGS) in *src/backend/utils/adt/timestamp.c*). Furthermore, each unit is decomposed into a sequence of logical code blocks with certain functionality. For each block, the LLM-generated plan includes a natural language description (e.g., *Step 1: Extract function arguments*) and a list of potential referenced units (e.g., *PG_GETARG_TEXT_PP()*).

While these plans outline essential details for code implementations, they might incur errors in plan generation, such as invalid reference units or incomplete processing logic. For instance, when synthesizing PostgreSQL's *date_trunc()* function, an incorrect plan might use a non-existent macro (e.g., *PG_GETARG_TXT()*) or omit the essential processing branches for the *interval* input type. To further improve the quality of generated plans, we employ an ensemble strategy that generates multiple pseudo-plans in parallel and refines them through a scoring-based filtering process.

5.1.2 Scoring-based Plan Filtering

To reduce the risk that the generated low-quality plans might mislead the synthesis process, we introduce scoring-based plan filtering based on a metric-based scoring function. Unlike existing methods that rely on LLM to produce uncertain evaluation results [15], this function deterministically assesses the generated plans, considering both faithfulness (i.e., the correctness of the listed referenced units) and simplicity (i.e., the number of code blocks). Specifically, given n implementation plans, it computes a normalized score for each plan based on three criteria. The score for plan k is computed as:

$$R_k = \alpha \cdot N(v_1^k) + \beta \cdot N(v_2^k) + \gamma \cdot N(v_3^k)$$

where the first two items characterize faithfulness, and the last item captures simplicity. v_1^k is the number of incorrectly referenced function units (i.e., the ones listed in "*Potential code elements*"). v_2^k is the number of incorrectly specified file locations for each function unit (e.g., the ones outside the specified database directory). v_3^k is the number of the listed function units. $N(x)$ denotes min-max normalization $N(x) = 1 - \frac{x - \min}{\max - \min}$ (or 1.0 if min = max) to scale the value among the n plans. α, β, and γ are the weight coefficients to trade between criteria, which are 0.4, 0.4, and 0.2, respectively.

We remove incorrect function units and referenced function units from the plan while computing this score, and filter out plans whose score is lower than a predefined threshold (e.g., 0.5).

5.2 Progressive Code Synthesis

To alleviate the risk of introducing errors during function synthesis, we propose progressive code synthesis that formulates function synthesis as a fill-in-the-blank task. Unlike traditional methods that require LLMs to synthesize the entire functions from scratch, we employ a probabilistic, self-correcting framework utilizing templates and scored pseudo-plans. This approach progressively

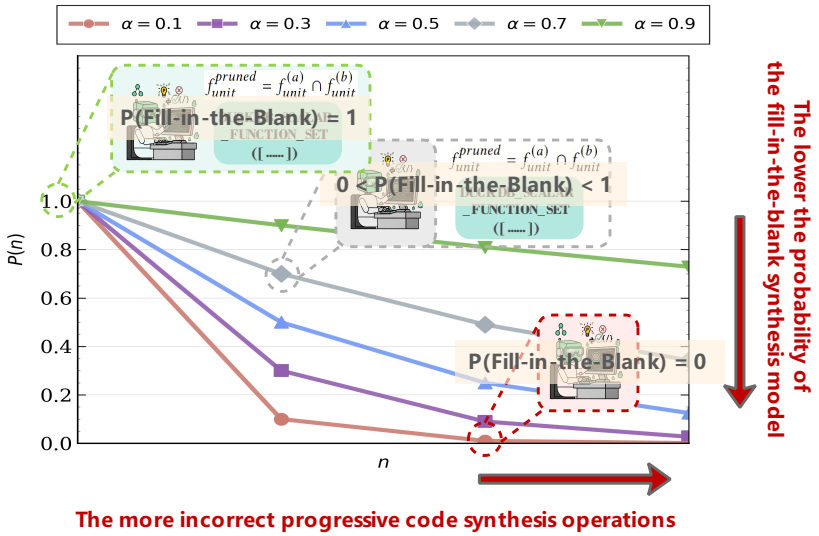


Fig. 6. Synthesis Model Adaption Example.

refines code by focusing on key function units and is uniquely capable of triggering a semantic rollback (i.e., switching from template-based to from-scratch synthesis) based on validation feedback.

Fill-in-the-Blank Synthesis Model. Building on the progressive synthesis paradigm, we introduce a fill-in-the-blank synthesis model that directs LLM's attention to critical function units within native database functions. Given the metadata (e.g., function category) specified in a function declaration, we first retrieve native functions that share the same metadata value. From these retrieved functions, we extract the top-k representative function units with placeholders in Section 4.2 which contain the most relevant implementation logic that can be reused. For example, in DuckDB, the LLM only needs to generate the distinctive function units for ten type-specific functions and add them to *DatePartFun::GetFunctions*, without needing to implement all function units, such as those for registrations in *DatePartFun*. To further enhance synthesis reliability, we invoke LLM to generate multiple candidate function units in parallel. A self-consistency strategy is then applied to merge these candidates by selecting units that appear most frequently across the generated results, ensuring coherent and high-quality synthesis.

Synthesis Model Adaption. To address the issue of incorrectly identified function units in Section 4.2 and enhance overall synthesis flexibility, we introduce an automatic adaptation mechanism that dynamically regulates the application of the fill-in-the-blank model based on historical synthesis. As shown in Figure 6, we formulate the adoption of the fill-in-the-blank model as a probabilistic problem, where the initial probability of using the fill-in-the-blank model is set to $P_0 = 1$. This probability decays exponentially with each failed synthesis attempt according to: $P(n) = \alpha^n$, where n denotes the number of code synthesis failures and α ($0 < \alpha < 1$) is the decay factor that determines the probability reduction rate. A smaller value of α corresponds to a faster decrease in the model adoption probability. Once the probability reaches zero (i.e., $P_0 = 0$), the LLM shifts to a fully exploratory synthesis mode, autonomously identifying and implementing all required function units from scratch. This transition serves as a rollback mechanism for handling errors such as incorrect coding plans. For example, when a plan indicates a non-existent macro or an incorrect file path, ***DBCooker*** can entirely ignore the flawed plan after several failed attempts and eventually implement the logic from scratch by searching the repository.

5.3 Three-Stage Code Validation

Existing approaches rely on the *bash* tools and LLM-generated *make* commands to validate function codes, which might introduce instability in the validation process. Unlike these methods that use generic syntax checks or static external tests [28], we propose a progressive validation strategy that delivers timely and reliable feedback by assessing generated function units at three hierarchical levels, from single-file correctness to multi-file integration. This strategy integrates with the databases, employs kernel-level compliance checks, and an LLM-augmented test generator to ensure comprehensive correctness. *(1) Syntax Level:* This level performs basic syntax checks within a single file to ensure variables are properly declared and referenced, using language-specific parsers like ANTLR [38]; *(2) Compliance Level:* This level verifies adherence to database-specific conventions across multiple files by customizing compliance tools with commands (e.g., *make install* in PostgreSQL), validating correct integration and build compliance within a specified timeout; *(3) Semantic Level:* This level evaluates whether generated functions produce expected outputs on representative test cases. We use LLM to generate SQLs to assess runtime behavior, maximizing coverage with three contextual sources: (a) Expertise instructions to guide LLM on error-prone areas like input types; (b) Existing database test suites for reusable examples and format conventions; (c) Decomposed internal code blocks to generate test cases relevant to the new implementation.

6 From Static Function Synthesis to Adaptive Tool Orchestration

Conventional synthesis workflows typically adopt pre-defined pipelines with fixed stages (e.g., *coding* → *testing*) [23]. However, database native functions exhibit diverse synthesis complexities. For example, PostgreSQL implements mathematical functions (e.g., *sin*) through concise wrappers around the standard library, while others require the implementations of specific processing logic. To overcome the limited flexibility of pre-defined workflows in handling function-specific synthesis, we shift from static synthesis workflows to an adaptive tool orchestration framework. Unlike rigid pipelines with fixed workflows [25, 44], we employ a memory-augmented strategy that dynamically sequences operations using retrieved similar trajectories and synthesis context.

6.1 Operation-as-Tool Abstraction

To seamlessly integrate synthesis operations with common tools (e.g., file search) into a unified workflow, we introduce operation-as-tool abstraction, which encapsulates each synthesis operation as a callable tool O. Each tool is implemented with a unified interface that standardizes invocation and result handling across different operations. The tool interface comprises three modules.

- *(1) Metadata Module:* Define the tool's name, the arguments, and the functionality descriptions.
- *(2) Core Logic Module:* Implement the designed operations via the processing logic module, such as coding plan generation.
- *(3) Routing Module:* Forward the input to the processing logic module and return the outputs in a standardized format that can be directly fed into the context of LLM.

For example, we encapsulate the coding plan operation as a tool with: (1) the *metadata module* including the name called "plan_agent", the input argument including "plan_num" for the expected plan number, and the illustration describing "Generate pseudo-based plans to outline and instruct synthesis"; (2) the *core logic module* to instruct LLM to generate multiple pseudo-based plans, and filter these plans based on pre-defined scoring functions; (3) the *routing module* to return the plans in JSON contents to input to LLM.

We integrate a hybrid tool set with: (1) *Common utilities* including bash command interfaces and static analysis tools for function characterization; (2) *LLM-based operations* covering proposed

Algorithm 1: Tool-based Stepwise Function Synthesis

Input: Function Declaration f_{dec}, Synthesis Tool Set $\mathcal{O}$, Synthesis Memory Pool $\mathcal{M}$
Output: Synthesized Function Units f_{unit}

1 / * 1. Get trajectory of similar functions as reference */
2 Retrieve items
$\mathcal{M}'_{\text{ref}} = \{m \in \mathcal{M} \mid \text{category}(m) = \text{category}(f_{dec}) \wedge \text{count\_op}(m) \in \{\min, \text{median}, \max\}\}$
3 / * 2. Orchestrate operation tool on-the-fly */
4 Initialize function unit $f_{unit} \leftarrow \emptyset$, trajectory $C \leftarrow \emptyset$, tool
$op \leftarrow op_{coding} \in \mathcal{O} = \{op_{plan}, op_{code}, ..., op_{validation}\}$
5 **while** *True* **do**
6 Update $f_{unit} \leftarrow \text{execute}(op)$ and update $C \leftarrow C \cup op$
7 Determine next tool $op' \leftarrow \text{LLM}(C, \mathcal{M}'_{ref})$
8 **if** $op' = op_{stop}$ **then**
9 Generate trajectory summary $s \leftarrow \text{LLM}(C)$
10 **break**
11 **else**
12 $op \leftarrow op'$
13 / * 3. Save trajectory and return synthesized units */
14 Store (f_{dec}, C, s) into memory pool $\mathcal{M}$
15 **return** f_{unit}

synthesis operations (e.g., pseudo-based plan generation) as well as LLM-optimization strategies such as majority-vote sampling and post-synthesis self-reflection.

6.2 Tool-based Stepwise Function Synthesis

To flexibly handle the synthesis of different functions, we introduce a tool-based stepwise synthesis framework that decomposes full implementations into the adaptive invocation of various tools.

Tool-based Orchestration Memory. To move beyond the limitations of LLM-only orchestration, we develop a tool-based orchestration memory that provides external and structured guidance for adaptive synthesis. The memory pool $\mathcal{M} = \{f_{dec}, C, s\}$ is built to capture orchestrations with: (1) the function metadata, such as the category in the function declaration f_{dec}; (2) the tool invocation trajectory C, which captures the tool sequence and associated statistics (e.g., number of each invoked tool); (3) a concise synthesis summary s describing the overall orchestration process as generated by LLM. To ensure the trajectory pool remains compact and representative, we implement a distribution-aware quality control mechanism. Trajectories are organized by function category (e.g., *math* and *date*) to enable efficient retrieval of orchestration patterns for similar operations. A new trajectory is inserted only when it provides new statistical evidence for its category. Specifically, insertion occurs when a trajectory's tool-invocation profile (i.e., the number and types of invoked tools) updates the maintained statistics of the pool (e.g., minimum, median, maximum tool usage).

Memory-Augmented Stepwise Orchestration. As illustrated in Algorithm 1, the synthesis begins by retrieving relevant entries from $\mathcal{M}$ via metadata matching (e.g., same category). To ensure balanced reference coverage, the entries with minimum, median, and maximum tool counts are selected to form the orchestration reference $\mathcal{M}'_{ref}$ (**line 2**). Starting from the LLM-based tool op

(e.g., code synthesis operation), we invoke the selected tools to iteratively update the synthesized function unit f_{unit} and the orchestration trajectory C. LLM dynamically determines the next tool op' (e.g., pseudo-plan generation for initial implementation outlining or a new plan generation to fix errors identified by code validation) based on the evolving synthesis context (**line 7**). If the determined next tool op_t aims to stop the orchestration process, we invoke LLM to produce the final trajectory summary s and store the complete orchestration information (f_{dec}, C, s) into the memory pool $\mathcal{M}$, and return the synthesized function units f_{units} (**line 15**).

7 Experiments

We conduct experiments to validate the effectiveness of ***DBCooker***.

7.1 Experiment Settings

Tested Databases. We synthesize native functions over three mainstream databases: (1) SQLite: A lightweight database engine implemented in C; (2) PostgreSQL: A full-featured, standards-compliant object-relational database written in C; (3) DuckDB: A high-performance in-process database for analytical processing written in C++. These databases coherently vary in complexity, where SQLite is more lightweight than the other two databases.

Tested Functions. We synthesize functions in two types: (1) those with ground-truth implementations in the official repositories, and (2) those currently absent from these repositories. To ensure representative coverage across diverse operations and complexity, we first classify functions based on categories defined in the official documents (e.g., date and time). We then select functions from each category based on their implementation complexity (e.g., the code length and the number of reference units). In total, we test 75, 145, and 128 functions on SQLite, PostgreSQL, and DuckDB, respectively. For functions absent from a target database (e.g., SQLite), we collect distinct function declarations from the other two databases (e.g., PostgreSQL and DuckDB) as synthesis inputs. The complete list of tested functions is included in our [artifact].

Evaluation Methods. We assess three types of methods. (1) LLM-based: We evaluate advanced LLMs including GPT-5 [9], Claude Opus 4.1 [2], Claude Sonnet 4.5 [3], and Qwen3 Coder Plus [42]. Each model is prompted to specify the exact file path for the generated code, which is then integrated into the repository using empirical placement rules (e.g., inserting function units before the *aBuiltinFunc[]* definition in SQLite). (2) RAG-based: We enhance LLM-based methods with CodeRAG, which retrieves ground-truth reference function units via static analysis and provides them as the context input to LLMs. (3) Agent-based: We assess state-of-the-art coding agents, including Claude Code [1], Qwen Code [15], and TRAE [23] (top-1 on the SWE-bench Verified leaderboard [28]).

Evaluation Metrics. We calculate the synthesis accuracy with two metrics. (1) Compliance Accuracy (Acc_{EXE}): The ratio of the synthesized functions that successfully compile and integrate into the database (e.g., the *./configure && make && make install* command in PostgreSQL); (2) Result Accuracy (Acc_{RES}): The ratio of the synthesized functions that pass all the testcases (e.g., those under *src/test/regress/sql* in PostgreSQL) and yield expected results.

Implementation. Experiments are performed on a workstation with 2 Intel Xeon E5-2678 v3 CPUs (2.50 GHz), 256 GB RAM, and 4 NVIDIA RTX 4090 Ti GPUs. The default LLM of agent-based methods is Qwen3 Coder Plus with the temperature set to 0.1. Each synthesis has a maximum timeout of 300 seconds. The parameters in Section 5.1 and 5.2 are lightweight heuristics and remain identical across databases in our experiments and require no per-database tuning. Following the quality control mechanism in Section 6.2, the resulting trajectory memory pool contains 90 entries

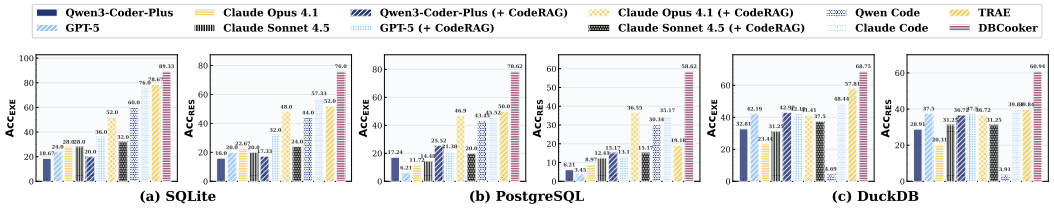


Fig. 7. Overall Code Synthesis Accuracy (%) of Different Synthesis Methods.

for SQLite (filtered from 550 initial trajectories across 75 functions), 174 for PostgreSQL (from 830 trajectories across 145 functions), and 154 for DuckDB (from 762 trajectories across 128 functions).

7.2 Overall Performance

We evaluate the synthesis accuracy and analyze the error distributions of different methods across three mainstream databases.

Synthesis Accuracy. According to the results in Figure 7, we have the following two observations. *First, **DBCooker** outperforms all the other methods with the highest synthesis accuracy across different databases.* Specifically, it achieves Acc_{EXE} and Acc_{RES} scores of 78.90% and 65.19%, respectively, outperforming other methods by an average margin of 124.37% and 149.68%, respectively. This improvement stems from the three database-aware modules integrated in ***DBCooker***. The *function characterization* module enables LLM to recognize system-specific specifications during synthesis, such as declaring function items in *aBuiltinFunc[]* for implementations in *src/func.c* of SQLite. The *adaptive orchestration* module coordinates the synthesis operations, decomposes function units, and identifies relevant reference units (e.g., adding *array_cross_product* in *ScalarFunctionSet* for functions implemented as *ArrayFixedCombine<float, CrossProductOp, 3>* and *ArrayFixedCombine<double, CrossProductOp, 3>*). In contrast, LLM-based methods mostly depend on their internal knowledge, often resulting in incorrect file placements (e.g., misplacing *#include float8.h* in PostgreSQL). Agent-based methods are inefficient due to the exploration of irrelevant parts of the repository. For instance, Claude Code and TRAE spend excessive time performing numerous search and read operations on unrelated files. Similarly, Qwen Code underperforms on DuckDB, repeatedly attempting full compilations with incorrect function implementations.

Second, RAG-based methods, which enrich the LLM context with reference function units, improve the synthesis accuracy of LLM-based approaches. Specifically, integrating CodeRAG results in an average accuracy improvement of 50.56% over LLM-based methods. This improvement arises because native functions rely on multiple reference units within complex database repositories, which LLMs often fail to identify on their own, despite being extensively pre-trained on the database repository. CodeRAG addresses this by efficiently retrieving relevant function units from the large search space and providing them to the LLM, facilitating accurate integration of these units into each function's implementation (e.g., 4 structs and 18 functions for the *array_cross_product* function in DuckDB).

Synthesis Error Distribution. We analyze the four error type distribution from different synthesis methods. For instance, declaration-related errors, including *incorrect declaration* and *incorrect reference*, indicate whether a method correctly identifies declaration locations and incorporates the appropriate reference function units within the database. As shown in Figure 8, agent-based methods produce fewer declaration-related errors, and ***DBCooker*** achieves the lowest declaration-related error. The reduction stems from the ability of agent-based methods to actively search the repository and verify the existence of required declarations (e.g., checking whether *repeat()* is registered in *pg_proc.dat*). ***DBCooker*** further optimizes by identifying all necessary units for each function (e.g., registering *repeat* in *pg_proc.dat* and implementing key units in *oracle_compat.c*)

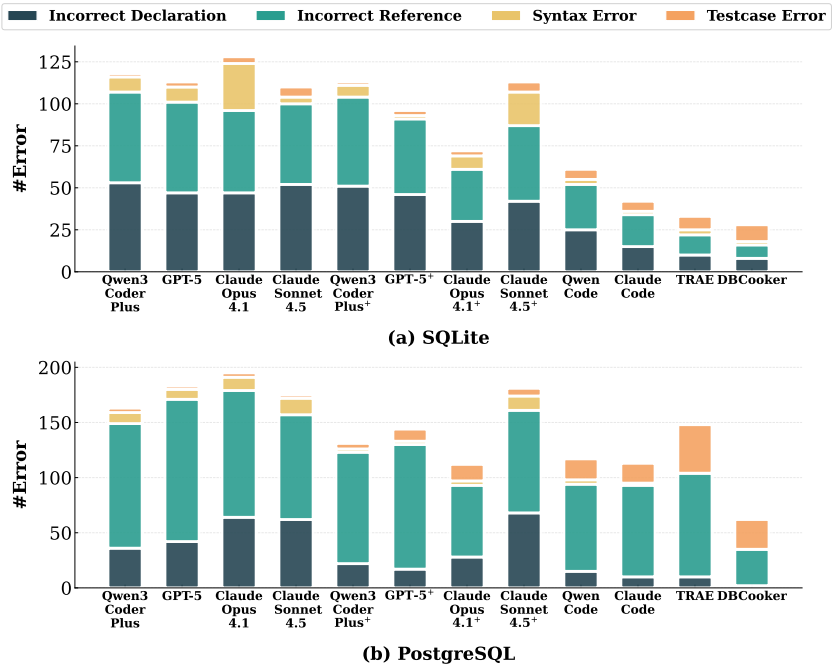


Fig. 8. Error Distribution of Different Synthesis Methods (+ indicates the usage of CodeRAG, and the codes of a native function might contain multiple errors).

and signaling absent references such as 'JsonParse' during progressive code validation. In contrast, LLM-based methods rely mainly on internal knowledge and the input context, which often fails to capture all relevant function units, leading to missing definitions (e.g., 'repeat' in PostgreSQL) or incorrect references to absent units (e.g., 'JsonParse' in SQLite's *json_remove()* function).

To further isolate the errors introduced by file search, we conduct additional experiments where agentic baselines were explicitly given the full context (+ Hint) of correct file paths, function declarations, and reference units to estimate their performance outside the file search problem. As shown in Figure 9, even with this complete context, these agents (+ Hint) still generate code containing undefined references and type-related semantic errors (e.g., introducing numeric corruption by using *int64* instead of *double*), with an average 22.56% lower accuracy than ***DBCooker***. This demonstrates that, beyond the challenge of extensive file search, failures of agentic baselines can also arise from issues such as incorrect declarations and references in their generated code, whereas ***DBCooker*** benefits from its integrated components, such as progressive validation.

Summary: *(1) Consistent Performance Improvement.* ***DBCooker*** outperforms both general agent-based methods (e.g., Claude Code), which struggle with complex database codebases by performing excessive searches and producing incorrect compilations, and the +Hint variants by 22.56%, which effectively eliminate retrieval bottlenecks and serve as the upper bound for retrieval-based systems. Even provided with the correct files, these methods still persist in generating code with undefined references and type-related semantic errors; *(2) Synthesis Goes Beyond Retrieval:* These results show that database native function synthesis goes beyond retrieval. While retrieval helps to identify the correct context (e.g., file paths and references), effective synthesis further requires enforcing database-specific correctness, such as generating type-dependent execution branches (e.g., for *date_trunc()*) and refining code under kernel-level constraints (e.g., successful compilation and passing SQL tests).

Table 1. Code Synthesis Accuracy (%) over Varying Function Difficulty.

Method	SQLite						PostgreSQL						DuckDB					
	EASY		MEDIUM		HARD		EASY		MEDIUM		HARD		EASY		MEDIUM		HARD	
	Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}
Qwen3-Coder-Plus	22.58	22.58	16.67	12.5	15.0	10.0	12.82	7.69	20.45	2.27	26.09	8.7	41.43	37.14	20.45	18.18	28.57	21.43
GPT-5	29.03	29.03	33.33	20.83	5.0	5.0	7.69	6.41	0.0	0.0	13.04	0.0	52.86	48.57	29.55	25.0	28.57	21.43
Claude Opus 4.1	35.48	35.48	33.33	16.67	10.0	10.0	11.54	8.97	6.82	4.55	21.74	17.39	32.86	30.0	11.36	9.09	14.29	7.14
Claude Sonnet 4.5	35.48	32.26	37.5	20.83	5.0	0.0	14.1	10.26	13.64	13.64	17.39	17.39	40.0	37.14	18.18	18.18	28.57	21.43
Qwen3-Coder-Plus (+ CodeRAG)	32.26	32.26	16.67	8.33	5.0	5.0	26.92	20.51	25.0	6.82	21.74	13.04	52.86	47.14	34.09	27.27	21.43	14.29
GPT-5 (+ CodeRAG)	54.84	51.61	33.33	29.17	10.0	5.0	24.36	21.79	15.91	2.27	21.74	4.35	52.86	47.14	25.0	22.73	42.86	35.71
Claude Opus 4.1 (+ CodeRAG)	64.52	61.29	50.0	45.83	35.0	30.0	43.59	38.46	56.82	40.91	39.13	21.74	51.43	45.71	25.0	22.73	42.86	35.71
Claude Sonnet 4.5 (+ CodeRAG)	45.16	41.94	37.5	20.83	5.0	0.0	15.38	12.82	20.45	15.91	34.78	21.74	47.14	45.71	22.73	18.18	35.71	35.71
Qwen Code	58.06	51.61	62.5	45.83	60.0	30.0	39.74	28.21	50.0	38.64	43.48	21.74	5.71	5.71	4.55	2.27	0.0	0.0
Claude Code	87.1	67.74	75.0	50.0	60.0	50.0	41.03	30.77	56.82	40.91	43.48	30.43	57.14	50.0	38.64	27.27	35.71	28.57
TRAE	87.1	58.06	79.17	41.67	65.0	55.0	46.84	22.78	54.55	18.18	52.17	8.7	65.71	44.29	54.55	36.36	28.57	28.57
DBCooker	**96.77**	**83.87**	**79.17**	**58.33**	**90.0**	**85.0**	**71.79**	**65.38**	**90.91**	**56.82**	**78.26**	**39.13**	**75.71**	**70.0**	**59.09**	**47.73**	**64.29**	**57.14**

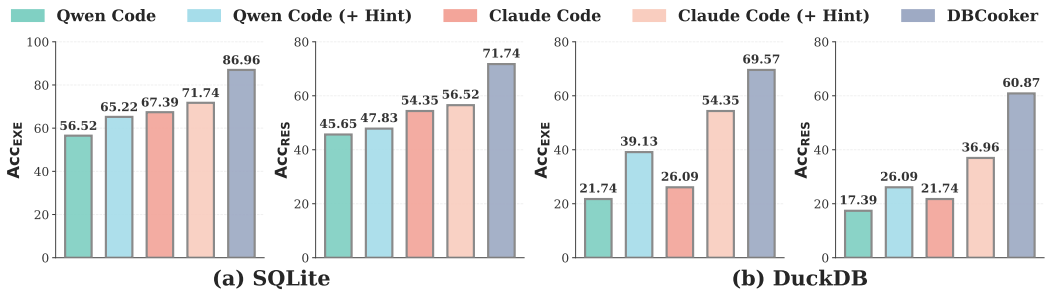


Fig. 9. Code Synthesis Accuracy (%) with Full Context.

7.3 Fine-Grained Analysis

To further evaluate the adaptability of ***DBCooker*** across diverse functions, we perform a fine-grained analysis in two dimensions.

Performance by Synthesis Difficulty. We classify functions into groups of three synthesis difficulty (i.e., EASY, MEDIUM, and HARD), and assess the synthesis accuracy within each group. We determine the synthesis difficulty of each function based on two code characteristics: (1) the number of involved function units and code tokens, and (2) the number of utilized reference function units. The boundaries of the three groups are automatically determined using the k-means algorithm based on the derived code statistics. As shown in Table 2, we have two observations.

*First, **DBCooker** consistently outperforms other methods across three difficulty levels.* Specifically, it achieves synthesis accuracies Acc_{EXE} and Acc_{RES} of 78.44% and 62.60%, which are 133.06% and 150.80% higher than other methods on average. Notably, for HARD functions, ***DBCooker*** reaches an accuracy of 68.97%, outperforming other methods by 197.10%. *Second, the performance of LLM-based and RAG-based methods degrades severely as synthesis difficulty increases from EASY to HARD.* Specifically, these methods achieve Acc_{EXE} and Acc_{RES} of 35.30% and 32.16% for EASY functions, while their accuracy decreases to 22.02% and 18.90% for HARD functions on average. The superior performance of ***DBCooker*** is attributed to its adaptive orchestration, which efficiently navigates the database repository and manages function synthesis across varying complexities. For instance, ***DBCooker*** implements EASY math functions in DuckDB by wrapping standard math library, such as *std::cbrt* for the *CbRtOperator* struct and *std::pow* for the *PowOperator* struct. For HARD functions, such as *bit_count()* in DuckDB, ***DBCooker*** uses six overloaded functions, each specialized for a distinct input type (TINYINT, SMALLINT, INTEGER, BIGINT, HUGEINT, and BIT), employing appropriate operators to efficiently count the number of set bits in the binary representation. In

Table 2. Code Synthesis Accuracy (%) over Function Category.

Method	Math Func.		Date Func.		String Func.		JSON Func.	
	Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}
Qwen3-Coder-Plus	6.67	6.67	23.53	23.53	51.35	8.11	0.0	0.0
GPT-5	6.67	6.67	17.65	11.76	10.81	2.7	0.0	0.0
Claude Opus 4.1	6.67	6.67	29.41	23.53	24.32	16.22	4.55	4.55
Claude Sonnet 4.5	16.67	16.67	29.41	23.53	24.32	18.92	9.09	9.09
Qwen3-Coder-Plus (+ CodeRAG)	40.0	6.67	35.29	29.41	27.03	16.22	40.91	40.91
GPT-5 (+ CodeRAG)	16.67	10.0	41.18	29.41	18.92	8.11	54.55	36.36
Claude Opus 4.1 (+ CodeRAG)	83.33	76.67	41.18	17.65	54.05	35.14	72.73	63.64
Claude Sonnet 4.5 (+ CodeRAG)	23.33	23.33	41.18	29.41	35.14	21.62	9.09	9.09
Qwen Code	56.67	43.33	58.82	41.18	45.95	32.43	86.36	54.55
Claude Code	56.67	46.67	58.82	23.53	51.35	45.95	90.91	**68.18**
TRAE	53.33	23.33	64.71	29.41	63.16	18.42	**95.45**	40.91
DBCooker	**96.67**	**80.0**	**94.12**	**52.94**	**89.19**	**62.16**	**95.45**	63.64

contrast, LLM-based and RAG-based methods primarily rely on the provided inputs and their pretrained internal knowledge, which limits their ability to accurately integrate multiple function units and manage numerous references, making them prone to errors when synthesizing HARD functions with extensive code tokens and complex references.

Performance by Function Category. We classify functions into category groups and assess accuracy within each group. Table 2 reports the results for four primary categories in PostgreSQL. We observe that ***DBCooker*** consistently achieves higher and more stable synthesis accuracy across different categories. Specifically, Acc_{EXE} ranges from 89.19% to 96.67% across the four categories, with an overall accuracy improvement of 151.11% over other methods. This advantage arises from ***DBCooker*** explicitly leveraging function category information. Unlike other methods that apply a uniform framework from scratch, ***DBCooker*** identifies distinctive function units through pairwise code comparison and retrieves relevant reference units within the same category. For example, it provides LLM with string-related reference units, including *pg_database_encoding_max_length*, *PG_GETARG_TEXT_PP*, and *PG_RETURN_TEXT_P*, to implement string functions such as *text_substr()* for SQL-level function *substr()* and *text_reverse()* for SQL-level function *reverse()* in *src/backend/utils/adt/varlena.c*. In contrast, LLM-based methods rely exclusively on pretrained knowledge, preventing them from fully exploiting category-specific patterns. Similarly, agent-based methods are inefficient because they repeatedly search the repository from scratch (e.g., scanning files based on keywords) without reusing information from functions in the same category, reducing their overall effectiveness.

7.4 Ablation Study

We assess the synthesis accuracy of five variants over the three modules in ***DBCooker***. *(1) Function Code Characterization:* We remove the collected information in the LLMs context, including the identified distinct function units, and the potential reference function units; *(2) Function Code Synthesis Operation:* We alternatively remove two operations (i.e., Pseudo-based Plan Generation,

Table 3. Code Synthesis Accuracy (%) of ***DBCooker*** Variants.

Component		SQLite		PostgreSQL		DuckDB	
Function Code Characterization		Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}
×		68.0	54.67	31.25	16.07	44.90	28.57
Function Code Synthesis Operation		Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}
Pseudo-based Plan Generation	**Three-Stage Code Validation**						
×	✓	74.67	57.33	37.04	20.37	48.98	30.61
✓	×	38.67	32.0	6.9	5.52	34.69	28.57
×	×	18.67	17.33	9.66	7.59	26.53	14.29
Adaptive Tool Orchestration		Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}	Acc_{EXE}	Acc_{RES}
×		65.33	49.33	29.76	21.19	51.02	30.61
***DBCooker* (Ours)**		**81.33**	**69.33**	**78.62**	**58.62**	**83.67**	**67.35**

and Progressive Code Validation); *(3) Adaptive Tool Orchestration:* We replace adaptive tool orchestration with a fixed multi-LLM collaboration without repository exploration. Table 3 reports the results on three databases and we have three observations below.

First, all proposed components are essential for accurate synthesis, with their removal leading to different degrees of accuracy drops. ***DBCooker*** obtains the improvement of Acc_{EXE} and Acc_{RES} by 42.14% and 37.50% with these components on average. For example, the function code characterization module successfully pinpoints the three required function units (i.e., *sumStep*, *totalFinalize*, and *sumInverse*) for the *total()* window aggregate function in SQLite. In the synthesis operation module, pseudo-based plan generation highlights potential reference units to guide function synthesis, preventing errors such as "unknown type name MultiRangeType" in PostgreSQL's *lower()* function. The adaptive tool orchestration enables progressive correction of errors through iterative code synthesis operations, resolving issues such as *"unknown jsonb value type"* detected during the validation of *"SELECT json_pretty()"*, and ultimately producing the ground-truth implementation.

Second, progressive code validation is crucial for reliable synthesis, and removing it leads to significant performance decline. For complex databases such as PostgreSQL, removing the three-stage validation causes a significant accuracy drop, with Acc_{EXE} falling from 78.62% to 6.9%. This drop reflects PostgreSQL's complexity (rather than ***DBCooker*** being specifically tuned for PostgreSQL), with more cross-file dependencies and more multi-branch execution paths than SQLite, where three-stage validation is vital for resolving diverse internal dependencies and verifying complex execution logic. Without feedback from the three-stage validation, ***DBCooker*** must rely on the LLM's pretrained knowledge (increasing the risk of producing incorrect or hallucinated macros) or coarse shell commands (ineffective for identifying deeper semantic failures). For instance, when synthesizing *date_trunc()*, (1) syntax and compliance-level validations identify the misuse of the *TIMESTAMP_GET_FIELD()* macro and missing *pg_proc* registration entries, while (2) semantic-level validation detects logic flaws in handling negative intervals. These feedbacks enable the LLM to iteratively refine its initial code implementations.

Third, components such as the pseudo-based plan generation and progressive validation work together to handle complex logic, and using them together improves accuracy. For example, when planning is disabled, PostgreSQL's Acc_{RES} increases slightly (5.52% → 7.59%), but this does not imply that planning is harmful. Instead, it reflects that coding plans and validations are interdependent for complex systems such as PostgreSQL. The coding plan is designed to be comprehensive, listing potential code elements (e.g., logic branches) to ensure LLM does not miss essential logic. For

Table 4. Extending SQLite with New Native Functions Generated by ***DBCooker*** (See code in our [artifact]).

Name	Category	Source	Declaration	Qwen Code	Claude Code	TRAE	*DBCooker*
covar_pop	Aggregate Function	PostgreSQL	WAGGREGATE(covar_pop, 2,0,0, covarPopStep, covarPopFinalize, covarPopFinalize, covarPopInverse, 0)	×	✓	✓	✓
bool_and	Aggregate Function	PostgreSQL	WAGGREGATE(bool_and, 1, 0, 0, boolAndStep, boolAndFinalize, boolAndFinalize, boolAndInverse, SQLITE_FUNC_ANYORDER)	×	✓	✓	✓
bool_or	Aggregate Function	PostgreSQL	WAGGREGATE(bool_or, 1, 0, 0, boolOrStep, boolOrFinalize, boolOrValue, boolOrInverse, SQLITE_FUNC_ANYORDER)	✓	×	✓	✓
century	Date Function	DuckDB	FUNCTION(century, 1, 0, 0, centuryFunc)	×	×	×	✓
monthname	Date Function	DuckDB	FUNCTION(monthname, 1, 0, 0, monthnameFunc)	×	×	×	✓
yearweek	Date Function	DuckDB	PURE_DATE(yearweek, 1, 0, 0, yearweekFunc)	×	✓	✓	✓
last_day	Date Function	DuckDB	FUNCTION(last_day, 1, 0, 0, last_dayFunc)	✓	×	✓	✓
lcm	Numeric Function	DuckDB	FUNCTION(lcm, 2, 0, 0, lcmFunc)	✓	✓	×	✓
even	Numeric Function	DuckDB	FUNCTION(even, 1, 0, 0, evenFunc)	×	×	×	✓
gamma	Numeric Function	DuckDB	FUNCTION(gamma, 1, 0, 0, gammaFunc)	×	×	×	✓
lgamma	Numeric Function	DuckDB	FUNCTION(lgamma, 1, 0, 0, lgammaFunc)	×	×	×	✓
nextafter	Numeric Function	DuckDB	FUNCTION(nextafter, 2, 0, 0, nextafterFunc)	×	×	×	✓
left	String Function	PostgreSQL	FUNCTION(left, 2, 0, 0, leftFunc)	×	✓	✓	✓
regexp_split_to_array	String Function	PostgreSQL	FUNCTION(regexp_split_to_array, 2, 0, 0, regexpSplitToArrayFunc)	×	×	×	✓
repeat	String Function	PostgreSQL	FUNCTION(repeat, 2, 0, 0, repeatFunc)	✓	✓	×	✓
to_hex	String Function	PostgreSQL	FUNCTION(to_hex, 1, 0, 0, toHexFunc)	×	×	✓	✓
translate	String Function	PostgreSQL	FUNCTION(translate, 3, 0, 0, translateFunc)	×	×	×	✓

example, the plan for *date_trunc()* includes logic for handling *tsrange* inputs (e.g., extracting range bounds), which are invalid for standard *timestamp* inputs. Without validation to remove these errors, LLM would try to apply this range logic to standard inputs, causing type errors that the validation would catch. When the plan is disabled, LLM avoids these errors via its pre-trained knowledge but still misses logic branches (e.g., for *interval* types). The absolute difference is small (2.07%) compared to ***DBCooker***'s improvement (37.50%), showing that turning off these components fails to handle complex systems.

7.5 New Native Function Synthesis

To evaluate ***DBCooker*** on new function synthesis, we extend SQLite's capabilities by introducing new functions from other databases. Specifically, we collect function declarations (e.g., name, category, and SQL examples from PostgreSQL and DuckDB) to simulate new functionality requests in SQLite. The function codes are not used as references, since native functions differ greatly across databases (see Section 4), making direct code transfer impractical. Table 4 lists the newly synthesized SQLite functions, where ***DBCooker*** presents superior performance for three reasons.

*First, **DBCooker** accurately implements all required function units for a new function and declares it correctly.* It realizes the core processing logic and enables SQL interface by declaring the function. For example, it implements three required function units for the SQLite *covar_pop()* aggregate function, including *covarPopStep* to update sums and counts, *covarPopFinalize* to compute the final covariance, and *covarPopInverse* to perform inverse calculations. The function is then declared with *WAGGREGATE(covar_pop, ..., covarPopStep, covarPopFinalize, ..., covarPopInverse, 0)* in *src/func.c*, making it callable via *SELECT covar_pop()*. In contrast, Qwen Code declares but fails to implement *covarPopInverse*, introducing compliance errors *"undeclared 'xInverse' in definition of macro 'WAGGREGATE' "*.

*Second, **DBCooker** effectively leverages reference function units within the database repository to support accurate function synthesis.* It correctly incorporates both standard and functional reference units in each synthesized function. For example, in the *bool_or* function, ***DBCooker*** uses *sqlite3_aggregate_context* for context management and *sqlite3_value_type(argv[0]) != SQLITE_NULL* for input validation within *boolOrStep*. In contrast, Claude Code introduces a redefinition error for 'struct SumCtx' during synthesis. Furthermore, ***DBCooker*** successfully implements the *regexp_split_to_array* function by utilizing reference units such as *sqlite3_value_text* and *sqlite3_str_new*,

whereas the other three methods encounter undefined references to 'sqlite3re_compile' and 'sqlite3re_match' in the external *sqlite3re.c*, resulting in compliance errors.

Third, ***DBCooker*** *adaptively manages diverse functions and progressively refines code to achieve the intended functionality.* It determines appropriate declarations and generates function codes for diverse functions through adaptive tool orchestration as shown in Table 4. For instance, it differentiates aggregate functions from other scalar functions using distinct macros (e.g., *WAGGREGATE* vs. *FUNCTION*) and assigns specific function names and arguments according to the category (e.g., *centuryFunc* with a single argument for the date function *century()*). In contrast, other methods rely on a uniform synthesis process, exploring the database repository from scratch, which reduces effectiveness. For example, Claude Code fails to generate the *century* function, and TRAE fails to generate *translate* due to exhaustive and irrelevant repository searches.

8 Related Work

General Code Generation. Recent advances in code generation are broadly categorized into three categories: (1) Prompt-based methods (e.g., Codex [20], GitHub Copilot [45]) treat code generation as conditional text generation from natural language or code prompts; (2) Agent-based systems (e.g., Claude Code [1], SWE-Dev [43], Qwen Code [15]) enhance LLMs with planning and tool usage for multi-step reasoning and code debugging; (3) Training-based models (e.g., Code Llama [29], StarCoder [21], WizardCoder [34]) focus on architectural improvements via pretraining on large code corpora or reinforcement learning from execution traces. Despite these advances, they are still limited in complex database-native function synthesis (see results in Section 7), which requires precise file searches, function-level references, and adaptive synthesis across varying input types and database constraints.

UDF Optimization. Existing work on User-Defined Function (UDF) optimization focuses on transforming high-level UDF logic into efficient internal representations. Froid [39] converts imperative UDFs into relational algebraic expressions, embedding them into SQL for cost-based optimization and parallel execution. Tuplex [41] accelerates and compiles Python UDFs into native code, with a dual-mode strategy that aggressively optimizes the common case and handles exceptions through a fallback path. Unlike ***DBCooker***, these methods focus on logic rewriting or runtime compilation rather than kernel-level code implementation.

Runtime Code Generation. Existing studies accelerate execution through temporary machine code. HyPer [36] compiles queries into LLVM-based machine code using a data-centric push-based model that retains data in CPU registers for optimized locality and branch prediction. Weld [37] employs a common intermediate representation for cross-library workflows, minimizes data movement, and generates efficient parallel code. These approaches differ from ***DBCooker***, focusing on temporary code for immediate execution rather than reusable, persistent native functions.

Database Migration. Recent research uses LLMs to bridge gaps across SQL dialects. CrackSQL [49, 50] automates dialect translation via a local-to-global strategy, iteratively refining translations by fixing local execution failures. PARROT [51] proposes a benchmark for cross-system SQL translation, specifically testing system-specific syntax. These works highlight the complexity and manual effort required to reimplement native functions across database systems.

9 Conclusion and Future Work

In this paper, we present ***DBCooker***, the first LLM-based system for automatic database native function code synthesis. It proposes a function characterization module to capture function declarations, distinctive units, and cross-unit references, with three database-aware synthesis operations (pseudo-based plan generation, progressive code synthesis, and three-stage code validation) and

adaptive tool orchestration for diverse function synthesis. Experiments on three databases show that ***DBCooker*** outperforms state-of-the-art methods and effectively supports new function synthesis.

As frontier models advance, ***DBCooker*** can continue to strengthen their capabilities for more effective function synthesis.

(1) Massive Codebase Fragmentation vs. Long-Context Reasoning: Database codebases are massive (e.g., over one million lines of code in PostgreSQL [14]) with scattered function units (e.g., separating declarations in *pg_proc.dat* from implementations in *src/func.c*), making it difficult to fully include in LLMs' context window. Even if future LLMs can process the entire codebase, blindly including everything incurs high inference costs, and effective synthesis still requires reliably identifying a small set of relevant, scattered units among a vast amount of irrelevant code. ***DBCooker***'s *Function Characterization* explicitly identifies these units, preventing LLMs from missing critical information that long-context reasoning over the entire codebase alone may overlook [33, 40].

(2) Deterministic Correctness Requirements vs. Probabilistic Generative Synthesis: Database systems are mission-critical, and their functions must satisfy strict, deterministic correctness guarantees. Even if future LLMs become more reliable, their outputs are still based on probabilistic generation and cannot inherently guarantee exact database correctness [19, 48]. ***DBCooker***'s *Function Code Synthesis Operations* remain essential by explicitly enforcing structural templates and correctness constraints via pseudo-plans and three-stage progressive validation. This external enforcement improves reliability by turning probabilistic generation into verifiable and correct implementations, instead of relying on LLMs alone to ensure strict database correctness.

(3) Dynamic Codebase Adaptation vs. Static Training Bias: Database functions evolve across versions (e.g., changes in function signatures, required macros, or system catalog definitions) [8]. Even with broader and newer training data, future LLMs inevitably learn a mixture of database versions and deprecated conventions, which might conflict with the exact requirements of a target database version [30, 32]. ***DBCooker*** addresses this by dynamically retrieving version-specific database implementation routines and enforcing them via *Adaptive Tool Orchestration*, allowing direct adaptation to database version changes without model retraining.

Acknowledgments

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

References

[1] Claude Code. *(Anthropic)*. https://www.claude.com/product/claude-code
[2] Claude Opus 4.1. *(Anthropic)*. https://www.anthropic.com/news/claude-opus-4-1
[3] Claude Sonnet 4.5. *(Anthropic)*. https://www.anthropic.com/claude/sonnet
[4] DuckDB. *(Dependency Management in DuckDB Extensions)*. https://duckdb.org/2024/03/22/dependency-management
[5] DuckDB. *(extension)*. https://duckdb.org/docs/stable/extensions/overview
[6] DuckDB. *(function)*. https://duckdb.org/docs/stable/sql/functions/overview
[7] DuckDB. *(Repository)*. https://github.com/duckdb/duckdb
[8] DuckDB. *(Versioning of Extensions)*. https://duckdb.org/docs/stable/extensions/versioning_of_extensions
[9] GPT-5. *(OpenAI)*. https://platform.openai.com/docs/models/gpt-5
[10] Oracle to PostgreSQL Migration. *(Cost)*. https://www.datapatroltech.com/blog/oracle-postgresql-migration-cost-savings
[11] Oracle to PostgreSQL Migration Challenge. *(EnterpriseDB)*. https://www.enterprisedb.com/oracle-postgres-migration-challenges-legacy-database
[12] Oracle to PostgreSQL Migration Challenge. *(Estuary)*. https://estuary.dev/blog/oracle-to-postgresql/
[13] PostgreSQL. *(extension)*. https://www.postgresql.org/docs/current/xfunc-c.html
[14] PostgreSQL. *(function)*. https://www.postgresql.org/docs/current/functions-comparison.html
[15] Qwen Code. *(Qwen)*. https://qwenlm.github.io/qwen-code-docs/
[16] SQLite. *(function)*. https://sqlite.org/lang_corefunc.html
[17] Understand. *(SciTools)*. https://scitools.com/
[18] Maryam Abbasi, Marco V Bernardo, Paulo Váz, José Silva, and Pedro Martins. 2024. Adaptive and scalable database management with machine learning integration: A PostgreSQL case study. *Information* 15, 9 (2024), 574.
[19] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code Generation with Generated Tests. In *ICLR*. OpenReview.net.
[20] Mark Chen, Jerry Tworek, Heewoo Jun, and et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021).
[21] Jean-Baptiste Döderlein, Nguessan Hermann Kouadio, Mathieu Acher, Djamel Eddine Khelladi, and Benoît Combemale. 2025. Piloting Copilot, Codex, and StarCoder2: Hot temperature, cold prompts, or black magic? *J. Syst. Softw.* 230 (2025), 112562.
[22] Korry Douglas and Susan Douglas. 2003. *PostgreSQL: a comprehensive guide to building, programming, and administering PostgresSQL databases*. SAMS publishing.
[23] Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, Yun Lin, Yingfei Xiong, Chao Peng, and Xia Liu. 2025. Trae Agent: An LLM-based Agent for Software Engineering with Test-time Scaling. *CoRR* abs/2507.23370 (2025).
[24] Haralampos Gavriilidis, Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2023. In-situ cross-database query processing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2794–2807.
[25] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *ICLR*. OpenReview.net.
[26] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).
[27] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. *CoRR* abs/2406.00515 (2024).
[28] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *ICLR*. OpenReview.net.
[29] Roman Kochnev, Arash Torabi Goodarzi, Zofia Antonina Bentyn, Dmitry Ignatov, and Radu Timofte. 2025. Optuna vs Code Llama: Are LLMs a New Paradigm for Hyperparameter Tuning? *CoRR* abs/2504.06006 (2025).
[30] Sachit Kuhar, Wasi Uddin Ahmad, Zijian Wang, Nihal Jain, Haifeng Qian, Baishakhi Ray, Murali Krishna Ramanathan, Xiaofei Ma, and Anoop Deoras. 2025. LibEvolutionEval: A Benchmark and Study for Version-Specific Code Generation. In *NAACL (Long Papers)*. Association for Computational Linguistics, 6826–6840.
[31] Yujia Li, David H. Choi, Junyoung Chung, and et al. 2022. Competition-Level Code Generation with AlphaCode. *CoRR* abs/2203.07814 (2022).
[32] Linxi Liang, Jing Gong, Mingwei Liu, Chong Wang, Guangsheng Ou, Yanlin Wang, Xin Peng, and Zibin Zheng. 2025. RustEvo^2: An Evolving Benchmark for API Evolution in LLM-based Rust Code Generation. *CoRR* abs/2503.16922 (2025).
[33] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Trans. Assoc. Comput. Linguistics* 12 (2024), 157–173.

[34] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. In *ICLR*. OpenReview.net.
[35] Antonios Makris, Konstantinos Tserpes, Giannis Spiliopoulos, and Dimosthenis Anagnostopoulos. 2019. Performance Evaluation of MongoDB and PostgreSQL for Spatio-temporal Data.. In *EDBT/ICDT Workshops*.
[36] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.
[37] Shoumik Palkar, James Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, and Matei Zaharia. 2017. A Common Runtime for High Performance Data Analysis. In *CIDR*. www.cidrdb.org.
[38] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.
[39] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *Proc. VLDB Endow.* 11, 4 (2017), 432–444.
[40] Stefano Rando, Luca Romani, Alessio Sampieri, Yuta Kyuragi, Luca Franco, Fabio Galasso, Tatsunori Hashimoto, and John Yang. 2025. LongCodeBench: Evaluating Coding LLMs at 1M Context Windows. *CoRR* abs/2505.07897 (2025).
[41] Leonhard F. Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *SIGMOD Conference*. ACM, 1718–1731.
[42] Qwen Team. 2025. Qwen3 Technical Report. *CoRR* abs/2505.09388 (2025).
[43] Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. 2025. SWE-Dev: Building Software Engineering Agents with Training and Inference Scaling. In *ACL (Findings)*. Association for Computational Linguistics, 3742–3761.
[44] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024. OpenDevin: An Open Platform for AI Software Developers as Generalist Agents. *CoRR* abs/2407.16741 (2024).
[45] Michel Wermelinger. 2023. Using GitHub Copilot to Solve Simple Programming Problems. In *SIGCSE (1)*. ACM, 172–178.
[46] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *NeurIPS*.
[47] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR*. OpenReview.net.
[48] Weiqiu You, Anton Xue, Shreya Havaldar, Delip Rao, Helen Jin, Chris Callison-Burch, and Eric Wong. 2025. Probabilistic Soundness Guarantees in LLM Reasoning Chains. *CoRR* abs/2507.12948 (2025).
[49] 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 (2025), 141:1–141:26.
[50] Wei Zhou, Yuyang Gao, Xuanhe Zhou, and Guoliang Li. 2025. CrackSQL: A Hybrid SQL Dialect Translation System Powered by Large Language Models. *arXiv Preprint* (2025). https://arxiv.org/abs/2504.00882
[51] 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*. https://huggingface.co/papers/2509.23338
[52] 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.
[53] 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.
[54] 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* (2026). https://arxiv.org/abs/2601.17058
[55] 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 × DATA. *arXiv preprint arXiv* (2025). https://arxiv.org/abs/2505.18458
[56] 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.

Received October 2025; revised January 2026; accepted February 2026