DBCooker: A Database Kernel-Specialized Coding Agent Harness

Wei Zhou
Shanghai Jiao Tong University
weizhoudb@sjtu.edu.cn

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

Qikang He
Shanghai Jiao Tong University
hqk0205@sjtu.edu.cn

Guoliang Li
Tsinghua University
liguoliang@tsinghua.edu.cn

Bingsheng He
National University of Singapore
dcsheb@nus.edu.sg

Chao Peng
Independent Researcher
chao.peng@acm.org

Yun Lin
Shanghai Jiao Tong University
lin_yun@sjtu.edu.cn

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

ABSTRACT

Database kernels continuously incorporate new built-in functions to support diverse applications. Synthesizing these native functions is highly complex, as it requires identifying multiple internal units, placing them in the correct source files, and reusing specific internal references. Although recent advancement in LLM-based coding agent frameworks (harnesses) exhibit superior capability, they are typically general-purpose and struggle with the massive and highly coupled database kernel repositories. To address this, we demonstrate ***DBCooker***, a database kernel–specialized coding agent harness. ***DBCooker*** offers three core functionalities: (1) *Kernel Dependency Inspector*, which enables users to inspect internal units and cross-file references via hybrid declaration collection, distinctive unit identification, and cross-unit reference analysis; (2) *Interactive Synthesis Planner*, which constructs high-level implementation plans through progressive code synthesis and three-stage code validation, allowing developers to verify and refine plans before execution; and (3) *Progressive Debugging Workspace*, which integrates with database-specific pipelines to provide efficient feedback and adaptive repair, powered by techniques such as *operation-as-tool abstraction* and *memory-augmented orchestration*. Conference attendees will experience how ***DBCooker*** seamlessly transforms high-level SQL function requests into validated, kernel-ready code for mainstream databases (e.g., SQLite, PostgreSQL, DuckDB).

PVLDB Reference Format:
Wei Zhou, Xuanhe Zhou, Qikang He, Guoliang Li, Bingsheng He, Chao Peng, Yun Lin, and Fan Wu. ***DBCooker***: A Database Kernel-Specialized Coding Agent Harness. PVLDB, 19(12): 4842 - 4845, 2026.
doi:10.14778/3827998.3828136

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://code4db.github.io/hi-opencook.

1 INTRODUCTION

Database-native function synthesis is both common and critical in daily database repository maintenance, serving as a fundamental mechanism for extending kernel capabilities and supporting evolving data processing needs. Modern databases continuously extend their kernel with such functions (e.g., string and numeric manipulation, JSON and XML analytics, and advanced aggregation [3, 5]). For example, PostgreSQL ships with hundreds of built-in functions and operators, enabling diverse operations from date extraction and text formatting to complex aggregates and JSON processing. These functions are compiled into the database kernel and remain transparent to query optimization and execution, enabling optimizations such as constant folding and vectorized execution. As a result, database-native function synthesis is highly relevant to the database community, especially for extending DBMS functionality and accelerating kernel development [9, 10, 12].

Table 1: Comparison of *DBCooker* and Other Frameworks.

Framework	Database Codebase Characterization	Function Synthesis Operation	Synthesis Workflow Orchestration
Codex [2]	×	Generic Coding Tools	LLM-driven General Coding
Claude Code [1]	×	Generic Coding Tools	LLM-driven General Coding
OpenClaw [4]	×	×	LLM-driven General Assistant
DBCooker	Declaration and Unit Analysis	Database-Specific Function Operations	Trajectory-Augmented Database Function Synthesis

Traditional development of database-native functions is a manual, repository-coupled process. Developers inspect documentation and system catalogs, identify the appropriate files and registration locations, and write and integrate implementation code into the kernel. With the rise of LLM-based coding agent harnesses, synthesis workflows have shifted from prompt-based code generation to agent-driven pipelines, where LLMs repeatedly search repositories, read context, and iteratively edit code. However, as summarized in Table 1, existing approaches still lack key capabilities necessary for efficient and correct database native function synthesis.

(Limitation 1) Lack of Database Repository Characterization. Implementing native functions is substantially more complex than writing SQL-level UDFs due to the large and deeply structured database codebases. Each native function often spans multiple internal implementation units, for instance, overloads for different argument types or helper routines for date and JSON processing, and requires precise declarations, correct registration entries, and integration with internal macros and APIs. However, existing general coding agents [1, 2] lack explicit database codebase characterization, forcing them to spend considerable effort on inefficient repository traversal and file searches to identify the small subset of relevant references and structural patterns for correct synthesis [7].

(Limitation 2) Insufficient Database-Specific Operations and Workflow Orchestration. Even after the relevant dependencies

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828136

are identified, correctly synthesizing native functions requires database-aware operations to ensure both implementation compliance and semantic correctness. It includes generating precise declarations, linking to internal APIs, and coordinating multiple implementation units within a structured synthesis workflow. However, existing harnesses provide only generic coding tools [2] and general-purpose agent workflows [4, 11], without explicit mechanisms to enforce database-specific semantics or orchestrate complex dependencies. Consequently, these methods might miss essential declarations and produce unresolved references, especially for complex functions such as advanced JSON operators, where multiple internal components must be coordinated correctly [6, 8].

Our Methodology. To achieve automated and accurate kernel-level database development, we propose ***DBCooker***, an LLM-based system that extends our research work [12] while overcoming limitations of generic coding agents. *(1) The Kernel Dependency Inspector* implements robust function characterization to expose the internal dependency graphs and pruned reference sets, collecting function declarations, identifying relevant internal units, and analyzing cross-unit references to produce characterization insights. *(2) The Interactive Synthesis Planner* integrates database-specific operations to construct high-level implementation plans, incrementally generate kernel code, and validate both syntax and semantics. *(3) The Progressive Debugging Workspace* orchestrates LLMs and tool calls through operation-as-tool abstraction and memory-augmented orchestration to dynamically choose similar and suitable synthesis workflows for functions of varying complexity, and provides interactive repair feedback. *During the system use, users can submit requests for the target database, inspect the characterized dependencies, and experience the interactive synthesis of new native functions.*

2 *DBCOOKER* OVERVIEW

Unlike general coding agents that expose intermediate steps and lack database-specific knowledge, ***DBCooker*** integrates function characterization, interactive synthesis, and adaptive workflow orchestration. It provides users with explicit insights into function declarations, internal units, and cross-unit dependencies, supporting the generation of kernel-ready code. As illustrated in Figure 1, it coordinates modules to implement three core functionalities.

(1) Kernel Dependency Inspector. The workflow begins when a user submits a high-level function synthesis request (e.g., *covar_pop* for population calculation in SQLite). Instead of immediately generating code, ***DBCooker*** activates its *Function Characterization* module. This module scans the target database repository and instantly presents the user with a curated dashboard of relevant textual declarations, system catalog entries, and essential cross-unit references. Users can visually inspect this context to ensure the implementation environment has been correctly characterized.

(2) Interactive Synthesis Planner. With the dependencies grounded, ***DBCooker*** invokes *Model Service Layer* to generate a structured pseudo-code plan via *the Pseudo-based Plan Generation*. This plan acts as an interactive skeleton, specifying target files and block-level intents. Users can review and edit this plan in the interface. Once approved, *the Progressive Code Synthesis* incrementally fills in the code details, focusing only on the distinctive logic, while reusing standard database boilerplate, and *the Three-Stage Code Validation* ensures syntax and semantic correctness.

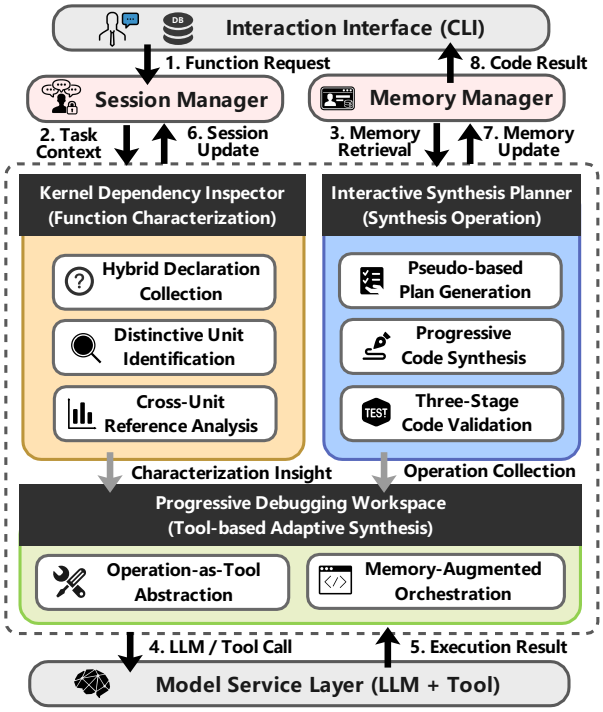


Figure 1: System Overview of *DBCooker*.

(3) Progressive Debugging Workspace. The synthesized code is automatically pushed to *the Progressive Debugging Workspace*, which orchestrates execution using *Operation-as-Tool Abstraction* and *Memory-Augmented Orchestration*, coordinated by *the Session Manager* and *Memory Manager*. It performs syntax checks, database-compliance verifications, and semantic SQL tests. If an error occurs, the system translates the compiler feedback into an actionable repair trajectory, allowing the user to guide the code refinement.

3 METHODOLOGY

To implement the three core functionalities above, ***DBCooker*** realizes a workflow that first systematically extracts and organizes function dependencies, then generates and incrementally synthesizes implementation plans, and finally iteratively refines code through adaptive, memory-guided orchestration.

► **Robust Function Characterization.** To address the problem of overwhelming repository complexity and sparse relevant references, ***DBCooker*** constructs a focused representation of function units and their dependencies to behave as a kernel dependency inspector. *(1) The Hybrid Declaration Collection* combines a document-based function collection, which parses official documentation to extract textual semantics, usage examples, and function categories, with catalog-based declaration extraction, which retrieves authoritative signatures, argument types, and registration information from system catalogs or kernel registration files; *(2) The Distinctive Unit Identification* applies graph-based unit extraction to construct inter-unit dependency graphs and pairwise unit Pruning to remove redundant units shared across functions, retaining only distinctive units that require new synthesis; *(3) The Cross-Unit Reference Analysis* identifies and prunes cross-unit references by type, preserving only the most relevant context for the synthesis model.

Figure 2: End-to-End Workflow of *DBCooker* (from the initial user CLI function request to the final web synthesis report).

► **Progressive Synthesis Operation.** To generate correct multi-unit code while preserving database-specific semantics and focusing on distinctive logic, ***DBCooker*** proposes coordinated synthesis operations that serve as an interactive synthesis planner, incorporating database development conventions. *(1) The Pseudo-based Plan Generation* generates candidate implementation skeletons, specifying target files, code blocks, and required reference units, and applies deterministic filtering to select the most faithful and minimal plans. *(2) The Progressive Code Synthesis* incrementally fills in function logic using templates derived from related functions, while adapting the synthesis scope if template-guided generation fails. *(3) The Three-Stage Code Validation* verifies correctness at syntax, compliance, and semantic levels using SQL test cases derived from official test suites and pseudo-plans.

► **Adaptive Repair Orchestration.** To accommodate the synthesis of functions with varying complexity and to handle synthesis failures and complex dependencies across multiple units, ***DBCooker*** adaptively orchestrates different operations and iteratively refines generated code within a unified workflow, offering a progressive debugging workspace. *(1) Operation-as-Tool Abstraction* encapsulates each synthesis operation, such as plan generation, as a callable operation with defined inputs, outputs, and execution semantics. *(2) Memory-Augmented Orchestration* maintains a structured memory that records executed operations, the associated function units, resolved references, and encountered errors. Memory is indexed by function signatures and reference identifiers, enabling the orchestration engine to retrieve relevant past attempts when synthesizing similar or dependent units.

4 DEMONSTRATION

We first demonstrate the end-to-end workflow of how ***DBCooker*** enables users to interactively perform function synthesis, inspection, and debugging for database kernel development via the CLI interface. Next, we demonstrate the quantitative evaluation of ***DBCooker*** with the general coding framework (Claude Code [1]).

4.1 End-to-End Experience

Figure 2 illustrates the end-to-end user experience of ***DBCooker*** through a concrete native function synthesis task for *covar_pop* absent in SQLite that computes population covariance. It demonstrates how a user starts with an interactive session, characterizes the target repository, performs adaptive synthesis, and inspects both the high-level summary report and the detailed code changes.

• **Step-1: Setup Interaction Interface.** Users first launch the interactive interface of ***DBCooker*** from the working directory (i.e., *.../code/sqlite*), as a terminal-style CLI with a session banner, command input area, live progress stream, and quick navigation hints. These elements together establish the entry point for the subsequent end-to-end synthesis workflow, in which they submit a request to synthesize the SQLite built-in aggregate *covar_pop*.

• **Step-2: Execute Characterization Command.** Once the session is ready, users interact with the CLI via three lightweight actions. *(1) Input Entry:* Users focus on the input line, which serves as the entry point for issuing requests and commands in the live session. *(2) Command Discovery:* The interface contains slash commands (displayed by */help*) that expose built-in commands with different functionalities and help users quickly select the intended operation. *(3) Function Characterization:* Users execute the command */characterization*, upon which ***DBCooker*** analyzes the target repository and returns a compact structural summary. In the *covar_pop* example, ***DBCooker*** scans the SQLite repository under the working directory and reports that it contains 10,065 functions with 9,435 unique names, spanning 2,208 text files and 553,700 lines. It further summarizes the file distribution (e.g., 343 *.c* files and 46 *.h* files) and highlights representative files, such as *src/btree.c*, giving users an immediate sense of the repository's scale and structure.

• **Step-3: Perform Adaptive Synthesis.** After characterization, users submit the synthesis request for the *covar_pop* function to ***DBCooker***, which performs adaptive synthesis by combining the

request with repository context and live tool feedback. This process can be categorized into three primary sub-steps.

(1) Planning: ***DBCooker*** automatically produces a structured plan that identifies *src/func.c* as the relevant implementation location and the built-in registration entry as the key validation point.

(2) Exploration: Instructed by this plan, ***DBCooker*** conducts targeted repository inspection by viewing relevant code regions and checking whether *covar_pop* and its registration entry *WAGGREGATE(covar_pop, ...)* are already present.

(3) Coding and Testing: When needed, ***DBCooker*** applies concrete edits to the target file and validates the result through subsequent tool feedback. This stage is interactive and adaptive, allowing users to optionally adjust each tool's usage. Rather than treating synthesis as a one-shot generation task, ***DBCooker*** alternates among planning, repository inspection, and coding-and-validation actions to incrementally converge to the final result.

• **Step-4: View Summary Report.** Upon completion, users can inspect the synthesis outcome in two sub-steps.

(1) CLI-to-Web Transition: The CLI prints a clickable link to the full summary report (i.e., *.../0000_trajectory.html*), enabling users to directly jump from the live terminal to the generated web report.

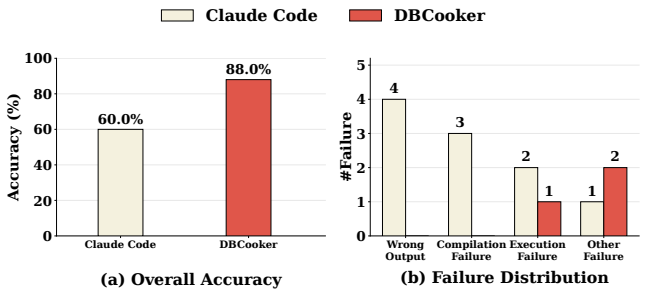
(2) Report Inspection: The web report consolidates the end-to-end execution into a single artifact, summarizing the final outcome, interaction statistics, tool invocations, and step-wise traces. In the *covar_pop* case, it shows that the run completed successfully after a sequence of planning, repository inspection, and edit-related actions, allowing users to understand how ***DBCooker*** progressively handled the synthesis task in *src/func.c*. The report thus serves not only as the final summary of a synthesis task but also as a structured web-based view that complements the live CLI interaction.

• **Step-5: Check Code Changes.** Finally, users review concrete code modifications in the code-change view, which presents patch-level differences in a *git diff*-style format. It allows them to directly examine the relevant changes in files such as *src/func.c*.

4.2 Real-World Evaluation

To further assess the effectiveness of ***DBCooker***, we evaluate its performance on real-world new-function synthesis and compare it against a state-of-the-art coding agent (i.e., Claude Code [1]) with the same LLM backbone (i.e., Qwen3-Coder-Plus). Specifically, we synthesize 25 real-world SQLite functions from PostgreSQL and DuckDB, covering a broad category of diverse functions.

Figure 3 reports the overall synthesis accuracy, defined as whether generated functions compile and pass the expected test cases, together with the distribution of failure types. ***DBCooker*** achieves 88.0% accuracy, substantially outperforming Claude Code at 60.0%. This improvement mainly stems from repository-grounded synthesis and verification. For example, in the *covar_pop* case, ***DBCooker*** first inspects the repository to identify existing or related implementations, locates the relevant code in *src/func.c*, and verifies SQLite-specific implementation patterns, including the aggregate state *CovarPopCtx*, step/inverse/final routines, and the corresponding *WAGGREGATE(covar_pop, ...)* registration. By following these engine-specific conventions before editing, ***DBCooker*** reuses SQLite's native aggregate-function design and avoids duplicate definitions, missing registrations, or inconsistent implementations. In contrast, Claude Code often fails structurally or semantically.



(a) Overall Accuracy **(b) Failure Distribution**

Figure 3: Performance Comparison of Real-World Synthesis.

For *bool_or*, it redundantly defines *SumCtx* despite an existing definition, causing a compilation error. For *nextafter*, it returns *1.0* instead of the immediately adjacent floating-point value from *1.0* toward *2.0*, indicating incorrect functional behavior. These cases are consistent with the failure distribution in Figure 3: Claude Code mainly suffers from wrong outputs and compilation errors, whereas ***DBCooker*** exhibits neither, demonstrating stronger robustness for repository-level function synthesis.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Claude Code. *(Anthropic).* https://www.claude.com/product/claude-code
[2] Codex. *(OpenAI).* https://openai.com/codex
[3] DuckDB. *(extension).* https://duckdb.org/docs/stable/extensions/overview
[4] OpenClaw. *(GitHub).* https://github.com/openclaw/openclaw
[5] PostgreSQL. *(Functions and Operators).* https://www.postgresql.org/docs/current/functions.html
[6] Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, and Hongwan Gao et.al. 2025. NL2Repo-Bench: Towards Long-Horizon Repository Generation Evaluation of Coding Agents. *CoRR* abs/2512.12730 (2025).
[7] 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.
[8] Minh Le-Anh, Huyen Nguyen, Khanh An Tran, Nam Le Hai, Linh Ngo Van, Nghi D. Q. Bui, and Bach Le. 2026. Do Not Treat Code as Natural Language: Implications for Repository-Level Code Generation and Beyond. *CoRR* abs/2602.11671 (2026).
[9] 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.
[10] 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.
[11] Wei Zhou, Xuanhe Zhou, Shaokun Han, Hongming Xu, Guoliang Li, Zhiyu Li, Feiyu Xiong, and Fan Wu. 2026. Are We Ready For An Agent-Native Memory System? *arXiv preprint* (2026). https://arxiv.org/abs/2606.24775
[12] Wei Zhou, Xuanhe Zhou, Qikang He, Guoliang Li, Bingsheng He, Quanqing Xu, and Fan Wu. 2026. Automating Database-Native Function Code Synthesis with LLMs. *Proc. ACM Manag. Data* 4, 3 (2026), 141:1–141:26.