AutoSQL: Extracting SQL Templates from Imperative ORM Code in Large-Scale Repositories

Junsong Pu
Sun Yat-sen University
Zhuhai, China
pujs@mail2.sysu.edu.cn

Yichen Li*
Chinese University of Hong Kong
Hong Kong, Hong Kong
ycli21@cse.cuhk.edu.hk

Zhuangbin Chen*
Sun Yat-sen University
Zhuhai, China
chenzhb36@mail.sysu.edu.cn

Zhihan Jiang
Chinese University of Hong Kong
Hong Kong, Hong Kong
zhjiang22@cse.cuhk.edu.hk

Zibin Zheng
Sun Yat-sen University
Zhuhai, China
zhzibin@mail.sysu.edu.cn

Abstract

Suboptimal SQL queries can significantly degrade the performance of cloud systems, motivating the extraction and auditing of SQL statements before deployment. However, Go ORM frameworks construct SQL imperatively through scattered method-call sequences, making it difficult to statically recover the resulting SQL templates. We present AutoSQL, a system that reconstructs SQL templates from Go ORM code. AutoSQL constructs a Code Index, a directed graph that captures structural dependencies between functions, types, and global variables as navigable edges. It then traces upstream call chains from ORM invocation sites to identify database-interacting functions as entry points. For each entry point, an LLM agent traverses the Code Index to collect code slices that influence SQL generation, switching to pattern-based search when the graph cannot resolve a retrieval goal. We call this strategy *Hybrid Context Retrieval*. Once sufficient context is collected, the agent synthesizes SQL templates. Evaluation on a benchmark of 579 test-covered entry points and 1,186 runtime-traced SQL statements from five large-scale Go repositories shows that AutoSQL achieves 75.13% to 79.76% recall, exceeding the static reachability baseline by 15.86% to 20.49% and outperforming existing methods by 4.57% to 49.45%.

CCS Concepts

• **Software and its engineering → Software maintenance tools**; **Software testing and debugging**.

Keywords

SQL Auditing, Object-Relational Mapping, Code Agents, Code Index

ACM Reference Format:
Junsong Pu, Yichen Li, Zhuangbin Chen, Zhihan Jiang, and Zibin Zheng. 2026. AutoSQL: Extracting SQL Templates from Imperative ORM Code in Large-Scale Repositories. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837420

*Corresponding authors.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837420

1 Introduction

Modern cloud services rely on database systems to provide reliable persistent storage. SQL queries that scan large tables without proper indexing, or that retrieve excessive data through missing filters, execute slowly and occupy database connections for extended periods. When such queries are triggered frequently, connection pools become exhausted and response latency increases for all services sharing the same database [6, 17]. To prevent inefficient queries from reaching production, developers perform SQL auditing, which reviews the SQL statements an application may execute and checks for performance risks before deployment.

In traditional database programming, developers write SQL as string literals in the source code, such as `db.Query("DELETE FROM users WHERE tenant_id = ?", id)`, making every query directly visible for review. However, modern applications increasingly adopt Object-Relational Mapping (ORM) frameworks, which generate SQL from method calls rather than requiring developers to write it directly. ORM frameworks are typically used in two distinct styles. In the *declarative* style, the mapping between code and database schema is fixed through static metadata (e.g., Java annotations like `@Table(name="users")`), and tools can extract SQL by parsing these declarations. In the *imperative* style, SQL is constructed dynamically. Each method call appends a component to a stateful builder object, and the final SQL emerges from the accumulated state at execution time. For example, a call sequence like `sess.Where("tenant_id=?", id).Delete(&User{})` constructs a `DELETE` statement through successive state mutations rather than a visible SQL string.

The Java ORM ecosystem predominantly adopts the declarative style, while the Go ORM ecosystem predominantly adopts the imperative style. The declarative style poses little challenge for SQL extraction, as tools can parse static metadata directly. The imperative style, however, creates a fundamental obstacle for SQL auditing, because the SQL statements an application will execute do not exist as inspectable artifacts in the source code. A single SQL statement may depend on code entities scattered across multiple files and modules, including a `TableName()` method that resolves the target table through runtime configuration, helper functions that progressively append `WHERE` conditions to a session object, and lifecycle

hooks that the framework invokes implicitly to inject additional column assignments. Reconstructing the complete SQL requires tracing all these dependencies and reasoning about how the ORM assembles them at runtime. Our task is to automatically extract such SQL templates from Go source code so that developers can audit them before deployment.

Existing static analysis tools for database code are designed for declarative ORM frameworks and cannot handle stateful, imperative query construction [25, 26]. Symbolic execution can theoretically explore all execution paths but suffers from path explosion in large codebases [4]. Recent LLM-based code agents offer an alternative through semantic reasoning, but existing code audit agents target security vulnerability detection [18, 32]. Their retrieval strategies are insufficient for the deep cross-boundary context collection that SQL reconstruction demands.

This paper presents AutoSQL, an LLM-based agent that reconstructs SQL templates from Go ORM code. The core challenge is context retrieval, as the agent must locate and collect the scattered code entities that contribute to each SQL statement across the repository. We first construct a Code Index, a directed graph that captures structural dependencies between code entities as navigable edges, and identify entry points for analysis by tracing upstream call chains from ORM invocation sites. For each entry point, the agent collects context through *Hybrid Context Retrieval*, which combines two complementary retrieval modes. The agent traverses the Code Index to efficiently trace the functions, type definitions, and implicit interface relationships that contribute to each SQL statement across files. However, some dependencies require finer-grained reasoning about how values propagate through dynamic language features, which symbol-level structural edges do not capture. For these cases, the agent switches to on-demand pattern-based search, synthesizing targeted regular expressions from its current retrieval goal to locate the relevant code directly in source files. Once sufficient context is collected, the agent synthesizes SQL templates by reasoning about ORM framework behaviors and enumerating control-flow paths that lead to different SQL variants.

We evaluate AutoSQL on five large-scale open-source Go repositories ranging from 38K to 845K lines of code, covering two ORM frameworks and three database dialects. We construct a benchmark of 579 test-covered entry points and 1,186 runtime-traced SQL statements, of which 40.73% rely on dynamic features that prevent pure static analysis. AutoSQL achieves 75.13% to 79.76% recall, surpassing the DBridge-style [25] static reachability baseline by 15.86% to 20.49% and outperforming existing code audit agents by 4.57% to 49.45%. Ablation results confirm that structural navigation via the Code Index improves recall by 0.92% to 22.07% over pattern-based search alone.

This work makes the following major contributions:

- We propose AutoSQL, an LLM-based agent that extracts SQL templates from imperative Go ORM code through Hybrid Context Retrieval, combining structural graph navigation with on-demand pattern-based search for implicit dependencies.
- We construct a benchmark dataset from five Go repositories comprising 579 test-covered entry points and 1,186 runtime-traced SQL statements, of which 40.73% rely on dynamic features beyond the reach of practical deterministic methods.
- Evaluation shows that AutoSQL achieves 75.13% ~ 79.76% recall on the runtime-traced benchmark, surpassing the DBridge-style static reachability baseline by 15.86% ~ 20.49% and existing code audit agents by 4.57% ~ 49.45%. Ablation results confirm that both structural graph navigation and pattern-based search contribute to the improvement.

2 Background and Motivation

2.1 Object-Relational Mapping Mechanics

Object-Relational Mapping (ORM) frameworks abstract database operations into application-level method calls, allowing developers to manipulate data without writing SQL directly [10, 27]. The framework translates these method calls into SQL at runtime. How this translation is specified determines whether the resulting SQL can be statically extracted from source code. ORM frameworks adopt one of two styles: declarative or imperative.

In the *declarative* style, the mapping between application code and database schema is specified through static metadata fixed at definition time. The Java ORM ecosystem implements this approach through annotations: `@Table(name="users")` binds a class to the `users` table, and `@Column(name="user_name")` binds a field to a column. Because annotation parameters must be compile-time constants, the schema mapping is statically determined, and tools can extract SQL structures by parsing these declarations without analyzing control flow.

In the *imperative* style, SQL is constructed dynamically through sequences of method calls at runtime. Each call appends a component, such as a table reference, a filter condition, or a column assignment, to a stateful builder object. The final SQL emerges from the accumulated state when the query is executed. Frameworks such as XORM [43] and Beego-ORM [5] in the Go ecosystem adopt this style. Figure 1 illustrates an example: when the `BatchDelete` function executes, the ORM produces the SQL `UPDATE users_default SET is_deleted=?, deleted_at=? WHERE tenant_id=? AND owner_id=?`. We trace how each component of this SQL is assembled from code scattered across the repository.

2.1.1 Runtime Table Name Resolution. The table name `users_default` originates from the `TableName` method at label ②. Rather than returning a fixed string, this method contains conditional logic: it checks the global configuration map `config.TablePrefix` and returns a prefixed or default name accordingly. The configuration map itself is initialized in a separate module at label ⑦, based on environment variables. Extracting the table name therefore requires tracing from the ORM invocation to the `TableName` method definition, and then to the global variable initialization across module boundaries. We refer to this pattern, where schema information is resolved through method dispatch and runtime state, as *runtime schema resolution*.

2.1.2 Stateful Builder Pattern. The `WHERE` clause is assembled by the helper function `applyDeleteConditions`, which appends filter conditions through calls to `sess.And` at labels ⑤ and ⑥. The session object maintains internal state that accumulates these conditions as it passes through the call chain. When `Update` is finally called at label ①, the ORM reads this accumulated state to generate the complete `WHERE` clause. Extracting these conditions requires following

Figure 1: An Example of SQL Generation in XORM

the call chain from `BatchDelete` into the helper function and tracking how the session state is progressively mutated. This *builder pattern* constructs queries through stateful accumulation across multiple function calls rather than through static SQL strings.

2.1.3 Lifecycle Hooks. The `SET` clause contains two columns: `is_deleted`, explicitly assigned at label ③, and `deleted_at`, which appears nowhere in the main function. The second column is injected by the `BeforeUpdate` hook at label ④. The ORM framework discovers this hook through Go's structural typing [16]: the `User` struct implicitly satisfies the `BeforeUpdateProcessor` interface by defining a method with the matching signature, without any explicit `implements` declaration. The framework detects this relationship at runtime via reflection and invokes the hook automatically before executing the update.

This mechanism, known as a *lifecycle hook*, poses a particular challenge for SQL reconstruction. Because the framework discovers and invokes the hook via reflection, no explicit call edge from the ORM engine to `BeforeUpdate` appears in the program's call graph. An analysis that follows the call graph will not reach the hook's body, leaving the SQL columns it contributes unrecoverable unless the implicit invocation is explicitly modeled.

2.2 Motivation

2.2.1 Problem Statement. The three mechanisms above illustrate that a single SQL statement in imperative ORM code is an emergent product of code slices distributed across multiple files and modules. These slices contribute to the final SQL through diverse dependency types, i.e., method calls, global variable reads, and implicit interface implementations, and their roles only become apparent when reasoning about how the ORM assembles them at runtime. The resulting SQL does not exist as an inspectable artifact in the source code, preventing developers from auditing queries before deployment. Our task is to automatically reconstruct SQL templates from such code: given a Go repository that uses an ORM framework, extract the SQL statements the application may execute.

2.2.2 Challenges of Static Approaches. Existing static analysis tools for database code are designed for declarative ORM frameworks. Tools such as DBridge [25] and SLocator [26] extract SQL by parsing metadata annotations (*e.g.*, `@Entity`, `@NamedQuery`) to build static models of query logic. As DBridge itself notes, these techniques cannot handle stateful, imperative query construction where the SQL structure is an emergent property of builder state accumulation rather than a static definition.

Existing Go-native tools serve different purposes. Vulnerability scanners such as `govulncheck` [36] detect known CVEs in dependencies, and SQL linters such as `sqlcheck` [3] check SQL strings for anti-patterns but require the SQL to already exist as input. Neither attempts to reconstruct SQL from ORM code. More general program analysis techniques such as symbolic execution could theoretically enumerate all SQL variants by exploring execution paths, but suffer from path explosion in real-world codebases [4].

2.2.3 LLM Code Agents as an Alternative. Large Language Model (LLM) code agents offer a different approach: reasoning about code semantics rather than matching pre-defined syntactic patterns. This distinction is critical for imperative ORM code, where the same database operation can be expressed through diverse code structures. Consider the `BeforeUpdate` hook in Figure 1, which sets the timestamp at label ④. The same soft-delete logic could equally be implemented by manipulating low-level clause builders, using variable assignments, or concatenating raw SQL strings. A rule-based analyzer would require explicit patterns for each variant, whereas an LLM can infer that these syntactically different forms produce equivalent effects and synthesize the corresponding SQL. Recent work has demonstrated LLM agents' effectiveness in repository-level code understanding tasks including code auditing [18, 32], automated program repair [41, 44], type inference [33], and root cause analysis [23].

However, existing code audit agents are primarily designed for security vulnerability detection rather than semantic reconstruction of database operations. Extracting SQL templates from Go ORM code presents unique challenges: tracking the state accumulation

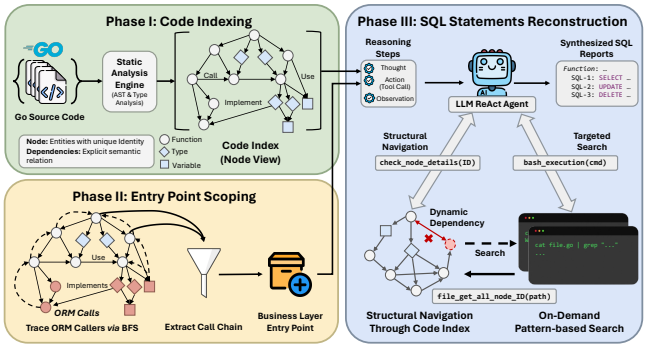

Figure 2: Overview of AutoSQL Methodology

of builder objects across multiple function calls, understanding implicit hook mechanisms such as `BeforeUpdate` that transform the semantics of an operation, and reconstructing complete SQL statements rather than merely detecting vulnerability patterns. This demands deep cross-boundary context retrieval that current agent architectures do not adequately support.

3 Methodology

3.1 Overview

As shown in Figure 2, AutoSQL reconstructs SQL templates from Go ORM code through three phases. In Phase I, we construct a Code Index, a directed graph that captures structural dependencies between code entities as a navigable backbone for the repository. In Phase II, we identify all direct ORM invocation sites and trace their upstream call chains within the Code Index, narrowing the analysis scope to functions that interact with the database. In Phase III, an LLM Agent collects code context relevant to each entry point through *Hybrid Context Retrieval*, which combines structural navigation over the Code Index with on-demand pattern-based search for dependencies beyond graph coverage, and then synthesizes SQL templates from the collected context. The Code Index and Hybrid Context Retrieval are framework-agnostic; ORM-specific knowledge (terminal APIs, struct tag semantics, hook conventions) is encapsulated in the system prompt and can be adapted to new imperative ORM frameworks independently.

3.2 Phase I: Code Indexing

We define the Code Index as a directed graph $G = (N, E)$ for navigation, where nodes are code entities and edges are dependency relations. We call these relations structural dependencies because they are explicit in program syntax and connect symbols through declaration and reference.

3.2.1 Indexing Strategy. The Code Index is designed to slice the entire repository at the symbol level and capture the structural dependencies between these symbols. While SQL reconstruction inherently requires reasoning about data flow, performing full-program data-flow analysis is practically infeasible for large-scale repositories. Conversely, limiting the scope to intra-procedural statement-level analysis offers little value, as dynamic SQL construction typically spans multiple functions and files.

Table 1: Node Categories in the Code Index

Node Type	Description
Function	All function and method definitions in the repository, including both standalone functions and methods attached to types.
Type	Struct and interface definitions. For structs, the node records field types and nested substruct definitions. For interfaces, it stores method signatures.
Variable	Global variable declarations along with their type information.

Table 2: Edge Categories in the Code Index

Edge Type	Description
Call	Direct invocation relationships between functions, including both regular function calls and method calls.
Use	References from a function to other symbols, such as reading or writing a global variable, creating a local variable of a certain type, or assigning a function to a variable.
Implement	A type implicitly implements an interface by providing matching methods, without explicit declaration.
Inherit	One type directly contains another inside its definition, gaining access to the embedded type's fields and methods.

Therefore, rather than building complex statement-level dependence graphs, we keep the graph lightweight to serve purely as a navigation backbone and delegate the actual value-flow reasoning to the LLM. In practice, the scattered slices needed for SQL reconstruction, such as struct definitions and helper functions, are highly reachable via structural relations. By storing the complete source code at each symbol node, we provide sufficient context for the LLM to deduce intra-procedural data flow directly from the retrieved code slices. This design choice is supported by prior findings [22], which demonstrate that LLMs reason more effectively over raw source code than over parsed intermediate representations.

3.2.2 Graph Structure. The graph contains three categories of nodes and four categories of edges, summarized in Tables 1 and 2. Each node stores its source code and is uniquely indexed by a *Global Identity* tuple (module, package, and symbol name) for precise retrieval. The node's metadata includes its file location, function signature, and field types. Crucially, every symbol reference within the metadata is resolved to a Global Identity to act as a direct link. For example, the metadata of `BatchDelete` (Figure 1) links `UserRepository` and `DeleteOptions` via their Global Identities, making these dependencies immediately reachable for the LLM Agent. With the Code Index constructed, the next step is to identify which functions warrant analysis.

3.3 Phase II: Entry Point Scoping

Directly analyzing every function within a repository-scale codebase is computationally wasteful, as the vast majority of functions do not interact with the database. To focus the Agent's reasoning

power on database interaction logic, we employ a scoping strategy that identifies all direct ORM invocation sites and traces their upstream call chains within the Code Index.

3.3.1 Identification of ORM Invocation Sites. The process begins by identifying functions that directly invoke ORM framework methods that execute SQL statements against the database, such as `Insert`, `Delete`, `Update`, `Get`, and `Find` on database session objects. Specifically, we rely on a predefined list of these terminal method signatures tailored to the supported ORM frameworks. These terminal methods trigger actual database operations. Functions containing these terminal method calls serve as the initial set of candidates for entry point analysis.

3.3.2 Upstream Call Chain Tracing. However, analyzing these invocation sites in isolation may sometimes be insufficient. In well-structured Go applications, variables that influence query conditions are frequently constructed by upstream callers and passed down as parameters. For example, in Figure 1, the terminal `Update` method is invoked directly within `BatchDelete` (Label ①). While the local control flow captures the core database operation, the specific query constraints can be heavily influenced by the `opts` parameter. To better understand how these fields might be populated in practical scenarios, it is beneficial to trace the call chain upstream to identify the callers of `BatchDelete`. Relying solely on the execution site without its upstream context might yield overly generic or incomplete SQL templates.

To resolve this, we trace the inverse call edges from ORM invocation sites up to a bounded depth D. Formally, let $\mathcal{S}_{orm} \subset N$ be the set of functions that directly invoke ORM APIs, and define the entry point set as $\mathcal{E} = \bigcup_{k=0}^{D} C_k$, where $C_k = \{u \in N \mid \exists v \in C_{k-1}, (u, v) \in E_{call}\}$ and $C_0 = \mathcal{S}_{orm}$. Here E_{call} denotes the set of *Call* edges in the Code Index. For each entry point, Phase II records the call chain to the ORM invocation site for subsequent analysis.

3.4 Phase III: SQL Statement Reconstruction

For each entry point identified in Phase II, the Agent reconstructs SQL statements through iterative context collection followed by single-turn SQL synthesis. Guided by predefined ORM framework knowledge, the Agent operates under the ReAct (Reasoning and Acting) paradigm [45], issuing tool invocations as *Actions*, receiving code slices and dependency information as *Observations*, and reasoning about data flow in *Thought* steps. This iterative process continues until the Agent accumulates sufficient context to synthesize SQL templates. We describe the retrieval toolset, the context collection strategy, and the SQL synthesis procedure below.

3.4.1 Retrieval Toolset. The Agent accesses the Code Index and source files through three tools, summarized in Table 3. These tools enable both structural navigation through the Code Index and pattern-based search through source files, with mechanisms to bridge between the two retrieval modes.

3.4.2 Context Collection. Given an entry point from Phase II, the Agent receives the call chain to the ORM invocation site along with the source code of functions in that chain. The Agent reasons about data flow in *Thought* steps, understanding how variables are transformed and how query conditions accumulate across function calls.

Table 3: Retrieval Tools for Code Navigation

Tool	Description
`check_node_details`	Accepts a Global Identity to query the Code Index, returning the node's source code, its metadata, and its forward and backward structural edges to other entities in the graph. It prompts if the node was already queried in the same session.
`file_get_all_node_identity`	Accepts a file path and extracts Global Identities of all entities defined within it.
`bash`	Executes shell commands such as `grep` to read source files, perform pattern-based search, and access configuration files in the repository.

The Agent then collects additional code context beyond the call chain, such as type definitions, lifecycle hooks, and configuration values that influence the generated SQL.

To collect these dependencies, the Agent uses a strategy called *Hybrid Context Retrieval*, which combines structural navigation through the Code Index with on-demand pattern-based search. The Agent primarily navigates the Code Index, which records structural relationships between functions, types, and variables as graph edges. This allows the Agent to trace dependencies across files by following edges. For example, when following the call to `TableName` in Figure 1, the Code Index directs the Agent to the `User` struct's `TableName` method at Label ②, rather than all methods named `TableName` across the repository.

However, structural navigation alone cannot satisfy all retrieval goals. The system prompt instructs the Agent to switch to pattern-based search when the Code Index cannot resolve the current retrieval goal to a specific code location. This occurs under two conditions. First, the Agent switches when no outgoing edge from the current node leads toward the retrieval goal, indicating that the target lies outside the graph's scope, such as configuration files or dynamically constructed calls. Second, the Agent switches when the retrieval goal requires information that graph edges do not encode. The symbol-level graph records which functions access a given symbol, but does not record details such as whether each access is a read or write, which arguments are passed, or which map key is involved. When the retrieval goal requires filtering based on such details, the Agent synthesizes a search pattern that encodes the precise syntactic signature of the target, then executes pattern-based search to locate it directly.

Pattern-based search runs on demand by turning the retrieval goal into a concrete search pattern that matches the target code form. Consider locating where `TablePrefix` is initialized in Figure 1: the `TableName` method at label ② reads `TablePrefix["user"]`, and the Code Index shows all functions that access this variable. The Agent understands the semantic goal is to find where the map key `"user"` is assigned a value, not just any access to `TablePrefix`. It constructs the pattern `TablePrefix[\"user\"]\s*=` that specifically matches assignment to this key, locating the initialization at label ⑦.

This design avoids a common tradeoff in existing approaches. Fixed pattern matching needs a predefined library of patterns for

many code structures, which does not transfer well across projects with different coding styles. Whole program data flow can track how values propagate, but it needs pointer analysis to model aliasing and does not scale to large codebases. The Agent instead builds the search pattern from the current semantic goal and avoids both a fixed rule set and global analysis.

Once the Agent discovers relevant code through pattern-based search, it uses the `file_get_all_node_identity` tool to map the discovery back to the Code Index, enabling further structural navigation from that point.This bidirectional switching provides two advantages. First, it allows the Agent to maintain high precision for dependencies with clear structural paths. Second, it effectively handles edge cases where the Code Index lacks the necessary semantic information.

The Agent stops context collection when neither structural traversal nor pattern-based search yields new SQL-relevant code slices along the current call chain. The context window limit of 90% of its maximum serves as a hard cutoff to prevent context overflow.

3.4.3 SQL Template Synthesis. Once the Agent accumulates sufficient context, it synthesizes SQL templates through a single turn Chain of Thought inference [40]. The system prompt guides the Agent to perform the following reasoning steps.

ORM Implicit Behavior Analysis. The Agent identifies implicit ORM semantics that alter the generated SQL without explicit invocation in the application code. Guided by framework-specific knowledge in the system prompt, the Agent analyzes data models for two primary implicit behaviors: struct tags and lifecycle hooks. For tag semantics, an annotation like `xorm:"deleted"` dictates that the framework internally transforms `DELETE` operations into `UPDATE` statements. For lifecycle hooks, if a struct implements a specific interface method like `BeforeUpdate`, the Agent recognizes that the framework automatically triggers this hook to inject or modify column values (e.g., updating a timestamp) right before query execution, ensuring the final SQL template captures these hidden state mutations.

Control-Flow Enumeration. This is a prompt-level design: we instruct the Agent to enumerate possible control-flow paths leading to distinct SQL variants before synthesizing each template, allowing the model to prune infeasible paths through semantic reasoning. We instruct the Agent to identify all SQL generation points, which are locations where ORM APIs are invoked to execute SQL statements, within the collected code and enumerate control-flow paths that lead to different SQL variants. Conditional branches that affect table names, column selections, or `WHERE` conditions result in separate SQL templates. Table names are resolved from `TableName()` methods or derived from struct names. The placeholder `<*>` is used only when an attribute cannot be inferred from the collected code snippets. Finally, the Agent synthesizes SQL statements according to the user-specified database dialect.

3.4.4 Example Walkthrough. To illustrate the complete reconstruction process, we use the example from Figure 1, assuming an upstream tracing depth of $D = 0$ in Phase II.

Context Collection. Starting from `BatchDelete`, the Agent traces forward to identify the SQL execution point at `sess.Update` (Label ①), and the explicitly set soft-delete flag (Label ③). Tracing backward, it identifies the `BeforeUpdate` hook via inspecting its type metadata (Label ④), and follows the call graph to `applyDeleteConditions` (Labels ⑤, ⑥) to gather `WHERE` conditions. To resolve the target table, it visits `TableName` (Label ②), where it encounters the global `TablePrefix` (Label ⑧). Requiring dataflow context beyond the graph, the Agent switches to pattern-based search to locate the variable's initialization in `Init` (Label ⑦).

SQL Synthesis. From the collected context, the Agent deduces that the explicitly set `IsDeleted` flag (Label ③) triggers the `BeforeUpdate` hook (Label ④) to populate the `DeletedAt` timestamp. Applying ORM implicit behavior analysis, it maps these fields to the `is_deleted` and `deleted_at` columns. The Agent also integrates the `tenant_id` and `owner_id` conditions gathered from `applyDeleteConditions` (Labels ⑤, ⑥). To simplify the demonstration, we assume the logic at both labels is triggered. Finally, by enumerating control-flow paths in `TableName` (Label ②), the Agent generates three complete SQL templates: `UPDATE {prod, test}_users SET is_deleted=?, deleted_at=? WHERE tenant_id=? AND owner_id=?` and `UPDATE users_ default SET is_deleted=?, deleted_at=? WHERE tenant_id=? AND owner_id=?`.

4 Study Design

To evaluate the effectiveness and efficiency of AutoSQL for SQL auditing in large-scale Go applications, we conducted a comprehensive empirical study to answer the following research questions:

- **RQ1 (Effectiveness):** How does AutoSQL compare to existing approaches in recall and accuracy?
- **RQ2 (Efficiency):** How does AutoSQL compare to existing approaches in token and time cost?
- **RQ3 (Ablation Study):** How does AutoSQL's Hybrid Context Retrieval compare against alternative retrieval strategies?
- **RQ4 (Parameter Sensitivity):** How does the upstream tracing depth D affect recall and cost?

4.1 Subject Systems

We selected five large-scale, open-source Go repositories to serve as our experimental dataset. To ensure experimental representativeness, our selection prioritizes projects characterized by widespread industrial adoption, substantial codebase scale, and extensive reliance on ORM frameworks involving complex dynamic SQL construction. All methods in our evaluation share the same Code Index built by Abcoder [9] as a one-time offline preprocessing step. Table 4 summarizes the statistics of the subject systems, including the indexing time for each repository.

4.2 Dataset Construction

We constructed the evaluation dataset through two stages: *runtime collection* and *test case sampling*.

Runtime Collection. First, to capture realistic data, we instrumented the source code of each subject system by modifying the underlying ORM configurations. We injected logging hooks to intercept database operations, recording both the generated SQL statements and their triggering call stacks during the execution of standard integration and regression test suites.

Test Case Sampling. To establish the ground truth, we employed random sampling to select a subset of test cases from these test suites. Three domain experts then analyzed the corresponding

Table 4: Statistics of Subject Systems. LoC (Lines of Code) excludes comments and blank lines. Index Time is the wall-clock time for building the Code Index.

Repository	Description	Stars	Go Files	LoC	ORM	Dialect	SQLs	Test Cases	Index (s)
Answer [2]	Q&A platform for teams	15.3k	434	55,961	XORM	SQLite	145	70	32
Gitea [13]	Self-hosted Git service	52.9k	2,789	332,970	XORM	SQLite	454	272	172
Go-Admin [14]	Admin panel framework	8.9k	291	37,939	Custom	MySQL	85	46	14
Grafana [15]	Analytics & monitoring platform	71.5k	5,085	844,968	XORM	MySQL	219	95	1,221
Harbor [20]	Cloud-native registry	27.3k	1,562	161,870	Beego	PostgreSQL	283	96	173
Total			10,161	1,433,708			1,186	579	1,612

SQL statements and execution paths captured from the sampled tests. Ultimately, we selected 579 integration test cases from these repositories, collecting and labeling 1,186 SQL statements. To ensure a fair comparison against the collected ground truth, we restrict all auditing methods to analyze only the entry points whose call chains intersect with the ground truth call stacks.

Evaluation Scope. Our ground truth is derived from runtime SQL logs captured during official integration and regression test execution. Recall is measured only against these runtime-traced SQL statements, and all methods are compared on the same set of call-stack-intersecting entry points. Entry points not exercised by any test lack independent ground truth and are therefore excluded from quantitative evaluation, as manual labeling is subjective and does not scale, while writing code to trigger uncovered paths amounts to authoring new integration tests rather than an independent evaluation method. The benchmark thus evaluates the task of recovering SQL templates for test-covered database-interacting entry points. AutoSQL's pipeline is applicable to all candidate entry points identified in Phase II (2,332 in total across the five repositories), but quantified recall and error analysis are scoped to this runtime-traced benchmark.

4.3 Comparative Methods

To answer **RQ1** and **RQ2**, we evaluate AutoSQL against two representative agent-based code auditing frameworks and a *Static Reachability* baseline derived from the capability boundaries of DBridge-style [25] deterministic approaches.

- **AutoSQL:** Our proposed approach using the Hybrid Context Retrieval strategy. We set the upstream tracing depth $D = 4$ for entry point scoping.
- **Static Reachability:** This baseline represents the practical capability boundary of DBridge-style [25] deterministic approaches on our dataset. During the ground truth labeling process, domain experts manually inspected each SQL construction path and marked it as *statically reachable* if the code contains none of the following three dynamic features: stateful builder patterns, reflection-based hook invocation, or runtime schema resolution. The recall of this baseline reflects the proportion of SQL statements that can be resolved through such deterministic approaches.
- **iCodeReviewer:** A general-purpose code audit agent that uses an expert system to synthesize prompts based on predefined rules [32]. We configure the rules for SQL reconstruction as follows: given a target function containing ORM execution APIs, the system extracts its direct callers, referenced types, and global variables to construct a single prompt. The agent then performs one-shot LLM inference to generate the audit result without iterative refinement.
- **RepoAudit:** A general-purpose code audit agent that navigates the codebase solely through a structural index without shell commands [18]. Since the original RepoAudit only explores the codebase along call relationships, we re-implemented its index with type reference edges to support SQL reconstruction. It employs a ReAct-style reasoning loop with a persistent memory mechanism, iteratively collecting function dependencies across multiple turns before synthesizing the final audit output.

To answer **RQ3**, we compare AutoSQL against three baselines that represent the mainstream retrieval and control paradigms employed by general-purpose coding agents. A recent empirical observation supports this comparison strategy: mini-SWE-agent [28], a minimal (~100-line) bash-only ReAct scaffold, achieves 76.8% on SWE-bench Verified [35] with Claude 4.5 Opus, only 2.4 percentage points below the top-performing Live-SWE-agent [42] (79.2%) with the same model. SWE-bench Verified is a human-curated subset of 500 software engineering tasks in which annotators confirmed that each problem description is unambiguous, the test patch is correct, and the task is solvable from the available context; in other words, each instance constitutes a well-defined task. That a minimal scaffold achieves comparable performance to the top agent on these well-defined tasks suggests that scaffold complexity alone may matter less than the choice of retrieval paradigm when the underlying LLM is strong. Motivated by this hypothesis, we identify three widely-used paradigms in the literature: (a) shell/search-based ReAct, as exemplified by SWE-Agent [44]; (b) embedding-based RAG retrieval [39]; and (c) hierarchical multi-agent decomposition [21]. Proprietary coding agents (e.g., Codex [30]) are not used as baselines because their internal retrieval, prompting, and control policies are not transparent or independently configurable, precluding controlled comparison. Moreover, much of their engineering targets open-ended tasks where requirements are underspecified and iterative clarification and planning are central. As the mini-SWE-agent observation above suggests, such additional scaffold complexity may not be the dominant performance factor for well-defined tasks. AutoSQL's task boundary is well-defined: given a database-interacting entry point, recover its SQL templates. Comparing retrieval paradigms under the same LLM therefore isolates the relevant variables more rigorously. Our three baselines directly

instantiate these paradigms, collectively covering the core retrieval and control mechanisms of general-purpose coding agents.

- **Bash-Only Agent:** A standard ReAct agent equipped solely with shell tools. This baseline represents the raw capability of an LLM to navigate the codebase using unstructured pattern-based search without any structural guidance. It is architecturally equivalent to mini-SWE-agent [28], making it a competitive, non-trivial baseline.
- **Vector-RAG Agent:** An agent equipped with a vector search tool in addition to standard shell utilities. The codebase is segmented into 30-line chunks and indexed using the `doubao-embedding-large_text-250515` model. The agent invokes this tool with keywords to retrieve relevant code snippets based on vector similarity.
- **Multi-Agent System:** A hierarchical framework where a *Manager* agent decomposes the auditing task and delegates specific sub-tasks, such as function analysis and symbol lookup, to subordinate *Worker* agents. All agents in this system are equipped with shell tools to navigate the codebase.

To answer **RQ4**, we evaluate AutoSQL with different upstream tracing depth values $D \in \{0, 2, 4\}$ to analyze the impact of this parameter on recall and token consumption.

4.4 Experimental Configuration

Models. For **RQ1** ~ **RQ3**, we evaluate all methods using two models at a temperature of 0.1: kimi-k2-0905 [29], representing leading open-weights models for code intelligence, and Claude 4.5 Sonnet [1], representing popular proprietary models; **RQ4** is evaluated exclusively on kimi-k2-0905.

Metrics. We employ four key metrics for evaluation. All metrics involving correctness judgments use LLM-assisted evaluation followed by domain expert review to ensure accuracy.

- **Recall:** The ratio of ground truth SQL statements with at least one functionally equivalent prediction. To be considered a valid match, the predicted SQL must originate from an entry point whose call chain intersects the ground truth's runtime stack, and it must be *functionally equivalent* (i.e., guaranteeing identical row selections or state mutations under any database state).
- **Estimated Precision (P):** Since the runtime-traced ground truth does not cover all SQL templates the code may produce, a predicted SQL absent from the logs is not necessarily a false positive. We estimate precision through sampling: for each method/repository pair, we sample 20 unmatched predictions (1,200 samples in total) and label each as *false positive* (strictly infeasible given the code) or *possible positive* (supported by the code but absent from runtime logs). The estimated precision is $P = (predicted - unmatched \times sampled\_fpr)/predicted$, where *sampled_fpr* is the sampled false-positive rate.
- **Token Cost:** This tracks the average number of input and output tokens consumed per entry point analyzed. It serves as a direct proxy for economic cost and inference latency. Average token usage is weighted by the number of test cases across all repositories.
- **Error Categories:** For each ground truth SQL that has no matching prediction, we classify the failure mode into three categories: (1) *Wildcard Missing*, where the predicted SQL uses a wildcard for an attribute whose value is fixed by code and not determined by runtime inputs; (2) *Condition Missing*, where conditions like `WHERE` clauses or `JOIN` conditions are incomplete or inconsistent; and (3) *Not Predicted*, where the method produces no SQL that matches that ground truth statement.

Execution Environment. We conducted all auditing experiments on a Linux server running Debian 10. The system operates within a KVM virtualized environment configured with 32 virtual CPUs and 64GB of RAM. We accessed the kimi-k2-0905 model via an internally deployed service and the Claude 4.5 Sonnet model via the official Anthropic API.

5 Experiment Results

5.1 RQ1: Effectiveness

Table 5 presents the recall, prediction precision, and processing time (excluding code index construction) comparison between AutoSQL and existing SQL auditing approaches across five subject systems. Since Claude 4.5 Sonnet is accessed through an external API with variable network latency and rate limiting, we only report processing time for kimi-k2, which is deployed on our internal infrastructure with stable and reproducible execution conditions.

Dynamic features limit static reachability at 59.27% recall. The remaining 40.73% of statements rely on reflection-based hooks, runtime schema resolution, and stateful builders, demonstrating that effective auditing must actively resolve dynamic runtime behaviors rather than relying solely on syntactic extraction.

AutoSQL addresses baseline failure modes through hybrid retrieval and ORM-aware synthesis. Unlike iCodeReviewer, which suffers from insufficient single-round context collection that causes dominant *Condition Missing* errors, and RepoAudit, whose rigid graph traversal misses non-structural dependencies and leads to *Not Predicted* errors, AutoSQL achieves 75.13% ~ 79.76% recall. It mitigates these issues via Hybrid Context Retrieval, combining the Code Index with *ad-hoc* pattern-based search to escape structural dead ends. Additionally, its ORM Implicit Behavior Analysis identifies framework-specific semantics from struct tags and lifecycle hooks, resolving dynamic columns and operations to further reduce *Wildcard* errors. Consequently, AutoSQL achieves an estimated precision of 96.41%/93.09% (kimi/Claude). Compared with RepoAudit (97.76%/88.40%), AutoSQL achieves substantially higher recall with only a modest precision difference, indicating that broader control-flow enumeration introduces few additional false positives. iCodeReviewer's lower precision (38.18%/64.85%) reflects its tendency to produce incomplete templates due to insufficient context.

5.2 RQ2: Efficiency

Table 5 and Table 6 present the effectiveness and efficiency comparison between AutoSQL and existing SQL auditing approaches.

AutoSQL consumes 94.0K to 136.4K input tokens per entry point and takes 196.4 minutes in total, surpassing both baselines in resource usage. However, this overhead is justified by its higher recall. The token cost is not due to inefficient retrieval but stems from the complexity of SQL reconstruction, which requires assembling fragmented logic from across the codebase. For instance, its 188× to 227× higher token usage compared to iCodeReviewer directly yields a recall gain of up to 45.45%, demonstrating that a minimal context

Table 5: RQ1: Effectiveness Analysis. Recall (%) and Estimated Precision (P, %). Processing time (T, in minutes, excluding code index construction) is reported for kimi-k2 only. iCR = iCodeReviewer, RA = RepoAudit, AS = AutoSQL

Repo	GT	Static Reachability	kimi-k2-0905						Claude 4.5 Sonnet						Time (min)		
			iCR		RA		AS		iCR		RA		AS		iCR	RA	AS
			R	P	R	P	R	P	R	P	R	P	R	P			
Answer	145	64.14	33.79	28.91	42.76	**97.37**	**73.10**	88.02	54.48	55.86	75.86	83.82	**86.21**	**90.17**	1.8	15.0	29.7
Gitea	454	55.07	41.19	37.46	71.59	**100.00**	**83.48**	96.89	46.04	66.56	83.48	**90.13**	**87.00**	89.99	6.6	54.5	81.9
Go-Admin	85	20.00	24.71	56.52	35.29	**90.07**	**61.18**	85.53	31.76	60.00	56.47	88.71	**61.18**	**91.19**	0.7	15.2	24.0
Grafana	219	78.54	19.63	32.95	68.95	**100.00**	**80.37**	**100.00**	31.05	56.87	78.54	93.13	**81.74**	**100.00**	3.1	20.7	33.3
Harbor	283	60.42	18.37	50.86	38.16	92.22	**62.90**	**100.00**	34.28	77.24	56.18	80.20	**68.90**	**96.81**	2.3	23.7	27.5
Avg / Total	1186	59.27	29.68	38.18	57.00	**97.76**	**75.13**	96.41	40.47	64.85	73.19	88.40	**79.76**	**93.09**	14.6	129.1	196.4

Table 6: RQ2: Efficiency Analysis. Error distribution (W=Wildcard, C=Condition Missing, NP=Not Predicted), token consumption (In=Input, Out=Output, in K tokens), and average tool-call count per entry point (CC). Sta. = Static Reachability.

Repository	Sta.	iCodeReviewer					RepoAudit					AutoSQL				
	NP	W	C	NP	In/Out	CC	W	C	NP	In/Out	CC	W	C	NP	In/Out	CC
kimi-k2-0905																
Answer	52	5	86	5	0.5/0.1	0	21	35	27	22.5/0.6	5.11	7	26	6	75.6/1.1	8.29
Gitea	204	14	230	23	0.5/0.1	0	35	50	44	13.9/0.5	3.00	9	42	24	84.3/1.2	8.22
Go-Admin	68	18	28	18	0.6/0.1	0	37	10	8	58.4/1.0	7.17	12	16	5	264.1/2.0	19.13
Grafana	47	3	168	2	0.6/0.2	0	3	46	19	11.1/0.6	2.88	2	37	4	58.5/1.2	6.98
Harbor	112	4	199	28	0.5/0.1	0	6	127	41	24.3/0.7	4.95	8	85	12	88.2/1.3	9.03
Total / Avg.	483	44	711	76	0.5/0.1	0	102	268	139	19.7/0.6	3.89	38	206	51	94.0/1.2	9.02
Claude 4.5 Sonnet																
Answer	52	3	56	3	0.6/0.3	0	7	25	3	74.8/3.0	6.57	3	15	2	95.9/1.8	8.96
Gitea	204	15	195	16	0.6/0.2	0	9	41	25	79.2/2.3	5.49	10	29	20	143.4/2.0	10.71
Go-Admin	68	22	26	8	0.7/0.2	0	21	5	11	264.2/2.6	14.83	17	11	5	262.7/2.6	19.13
Grafana	47	2	135	4	0.7/0.3	0	3	40	4	43.9/1.8	4.18	2	35	3	78.0/1.9	7.67
Harbor	112	5	175	2	0.6/0.3	0	2	114	8	65.5/3.8	6.10	3	72	13	143.3/1.9	13.27
Total / Avg.	483	47	587	33	0.6/0.2	0	42	225	51	85.3/2.6	6.25	35	162	43	136.4/2.0	11.09

is insufficient. Compared to RepoAudit, the increased cost is attributable to our methodology, such as Hybrid Context Retrieval and implicit ORM analysis, which enable a more thorough control-flow enumeration. This comprehensive context is essential for reasoning about control flow and cannot be significantly compressed without sacrificing accuracy. In terms of agent iterations, AutoSQL averages 9.02/11.09 tool calls per entry point (kimi/Claude), compared to 3.89/6.25 for RepoAudit; the increase reflects the additional structural traversal and pattern-based search needed to resolve cross-file dependencies.

5.3 RQ3: Ablation Study

To quantify the contribution of our Hybrid Context Retrieval design, we compare AutoSQL against three representative agent paradigms, each employing a distinct retrieval or control strategy. Table 7 presents the detailed ablation results.

Hybrid Context Retrieval outperforms all three mainstream agent paradigms. Structural navigation via the Code Index shows consistent advantages over alternative retrieval strategies. AutoSQL outperforms the Bash-Only agent by 0.92% to 22.07% across individual projects, confirming that purely pattern-based search may miss structural dependencies. Vector-RAG achieves similar recall to Bash-Only, indicating that embedding-based similarity struggles with structural code relations. Furthermore, AutoSQL yields the lowest *Condition Missing* errors (206/162 for kimi/Claude) because structural graph traversal ensures more comprehensive context collection. In contrast, the Multi-Agent approach suffers more *Not Predicted* and *Condition Missing* errors, especially on Claude, as task decomposition divides the necessary global reasoning context across isolated agents. Across all methods, estimated precision remains generally high across all methods (88.12% ~ 96.41% for kimi, 79.98% ~ 96.25% for Claude), indicating that false-positive rates are low regardless of retrieval paradigm. Methods with more thorough context retrieval tend to enumerate more control-flow paths, which can introduce a small number of additional false positives; however, this tradeoff is modest compared to the recall gains. Since these three baselines collectively cover the core retrieval mechanisms (shell-based ReAct, embedding-based RAG) and control paradigms (hierarchical decomposition) of general-purpose coding agents, AutoSQL's consistent advantage suggests that Hybrid Context Retrieval, which combines structural

Table 7: RQ3: Ablation Study. (a) Recall (%) and Estimated Precision (P, %). (b) Error distribution (W=Wildcard, C=Condition Missing, NP=Not Predicted) and token consumption. Bash = Bash-Only, Vec = Vector, MA = Multi-Agent, AS = AutoSQL.

(a) Recall & Estimated Precision

Repo	GT	kimi-k2-0905								Claude 4.5 Sonnet							
		Bash		Vec		MA		AS		Bash		Vec		MA		AS	
		R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P
Answer	145	66.21	93.58	69.66	87.53	71.03	**96.77**	**73.10**	88.02	64.14	71.71	75.86	**96.94**	68.97	93.77	**86.21**	90.17
Gitea	454	81.50	83.27	81.06	90.26	77.53	93.17	**83.48**	**96.89**	82.16	89.67	85.24	**96.63**	83.70	67.03	**87.00**	89.99
Go-Admin	85	54.12	92.03	55.29	78.97	55.29	**97.03**	**61.18**	85.53	50.59	**100.00**	52.94	79.22	56.47	91.79	**61.18**	91.19
Grafana	219	77.63	89.55	75.34	96.48	79.45	96.50	**80.37**	**100.00**	80.82	96.39	80.37	**100.00**	80.37	96.47	**81.74**	**100.00**
Harbor	283	60.78	96.97	56.89	97.04	60.42	**100.00**	**62.90**	**100.00**	67.14	96.94	65.72	94.09	54.77	**97.15**	**68.90**	96.81
Avg	1186	72.01	88.12	70.99	91.75	71.42	95.53	**75.13**	**96.41**	73.86	90.90	76.22	**96.25**	72.43	79.98	**79.76**	93.09

(b) Error Distribution & Token Consumption

Repository	Bash-Only					Vector					Multi-Agent					AutoSQL				
	W	C	NP	In/Out	CC	W	C	NP	In/Out	CC	W	C	NP	In/Out	CC	W	C	NP	In/Out	CC
kimi-k2-0905																				
Answer	9	29	11	46.9/1.1	8.41	5	31	8	62.6/1.0	8.64	3	34	5	54.1/2.7	18.83	7	26	6	75.6/1.1	8.29
Gitea	14	54	16	80.0/1.3	11.42	14	53	19	89.2/1.2	11.18	17	70	15	100.8/3.1	25.87	9	42	24	84.3/1.2	8.22
Go-Admin	19	14	6	138.8/1.6	17.43	18	7	12	138.6/1.4	16.00	19	8	11	108.6/3.3	28.70	12	16	5	264.1/2.0	19.13
Grafana	5	38	6	71.3/1.4	10.87	7	43	4	68.8/1.2	9.48	2	41	2	116.5/3.1	24.38	2	37	4	58.5/1.2	6.98
Harbor	3	90	18	74.7/1.3	10.21	3	100	19	78.7/1.1	9.85	2	105	5	102.8/3.7	26.68	8	85	12	88.2/1.3	9.03
Total / Avg.	50	225	57	78.3/1.3	11.24	47	234	62	84.8/1.2	10.76	43	258	38	98.7/3.2	25.13	38	206	51	94.0/1.2	9.02
Claude 4.5 Sonnet																				
Answer	1	40	11	51.9/2.9	7.33	0	21	14	86.2/1.7	9.81	4	24	17	126.9/3.4	18.56	3	15	2	95.9/1.8	8.96
Gitea	17	41	23	118.4/2.0	14.59	12	40	15	103.1/1.8	11.90	15	42	17	249.7/5.7	36.50	10	29	20	143.4/2.0	10.71
Go-Admin	28	9	5	153.3/2.7	17.04	18	6	16	170.5/2.1	18.85	24	9	4	207.9/5.0	30.17	17	11	5	262.7/2.6	19.13
Grafana	3	36	3	78.1/1.8	10.41	4	36	3	62.0/1.6	8.24	2	39	2	113.1/3.2	18.95	2	35	3	78.0/1.9	7.67
Harbor	3	71	19	133.5/2.0	13.24	3	75	19	115.2/1.7	10.82	2	80	46	169.8/3.5	19.05	3	72	13	143.3/1.9	13.27
Total / Avg.	52	197	61	109.0/2.2	13.00	37	178	67	101.7/1.7	11.42	47	194	86	195.9/4.6	28.05	35	162	43	136.4/2.0	11.09

graph navigation with on-demand pattern-based search, plays an important role in SQL reconstruction and is unlikely to be substituted by scaffold complexity alone.

Token consumption reveals the efficiency trade-offs among different retrieval paradigms. AutoSQL consumes more input tokens than the Bash-Only and Vector-RAG baselines (averaging 94.0K/136.4K vs. 78.3K ~ 84.8K / 101.7K ~ 109.0K for kimi/Claude). This increase occurs because resolving cross-file dependencies, such as implicit hooks and type definitions, requires fetching additional code slices to mitigate *Condition Missing* errors. Conversely, Vector-RAG retrieves code based on semantic similarity, introducing structurally irrelevant chunks without improving recall. Furthermore, the Multi-Agent paradigm incurs the highest output token overhead (3.2K/4.6K vs. 1.2K/2.0K for AutoSQL) due to inter-agent coordination. Consequently, AutoSQL achieves higher recall by directing the token budget toward structural context rather than semantic similarity searches or inter-agent communication. This is also reflected in tool-call counts: Bash-Only averages 11.24/13.00 calls per entry point (kimi/Claude) due to exploratory search, while AutoSQL requires only 9.02/11.09 calls by leveraging Code Index navigation; the Multi-Agent paradigm incurs the most at 25.13/28.05, driven by inter-agent task delegation.

5.4 RQ4: Parameter Sensitivity

Table 8 analyzes the impact of the upstream tracing depth parameter D on recall and token consumption.

The optimal tracing depth depends on project-specific ORM encapsulation. Go-Admin requires $D = 2$ for meaningful recall (jumping from 1.18% to 62.35%), indicating heavily encapsulated ORM calls. Conversely, Grafana achieves strong recall at $D = 0$ (84.93%) and peaks at $D = 2$ (88.13%); expanding D further to 4 introduces noise, dropping recall to 80.37%. Gitea shows consistent recall improvements up to $D = 4$. Token consumption mirrors these structural differences. For projects with shorter ORM invocation chains (e.g., Grafana), token usage actually decreases from $D = 2$ to $D = 4$. This happens because providing additional upstream context upfront prevents the agent from executing token-expensive supplementary searches during Phase III. In contrast, projects with very deep call chains (e.g., Harbor and Go-Admin) exhibit continuous token increases without proportional recall gains. Thus, optimal depth selection balances uncovering encapsulated logic against the noise and cost of excessive upstream layers.

6 Threats to Validity

Internal Validity. Our ground truth derives from runtime logs during test execution, covering only SQL statements triggered by

Table 8: RQ4: Parameter Sensitivity Analysis. Impact of upstream tracing depth D on recall and token consumption using kimi-k2-0905.

Repository	GT	D=0		D=2		D=4		Token (In/Out, K)		
		Recall	Hits	Recall	Hits	Recall	Hits	D=0	D=2	D=4
Answer	145	65.52	95	63.45	92	73.10	106	31.7/0.7	75.5/1.1	75.6/1.1
Gitea	454	52.64	239	75.55	343	83.48	379	64.6/1.1	96.2/1.2	84.3/1.2
Go-Admin	85	1.18	1	62.35	53	61.18	52	122.4/1.2	251.4/1.9	264.1/2.0
Grafana	219	84.93	186	88.13	193	80.37	176	55.6/1.1	75.9/1.3	58.5/1.2
Harbor	283	57.60	163	61.84	175	62.90	178	40.4/0.8	84.3/1.2	88.2/1.3
Average	1186	57.67	684	72.18	856	75.13	891	52.7/1.0	99.7/1.3	94.0/1.2

tested paths. SQL statements in untested paths are absent from the ground truth, so a predicted SQL not in the logs may still be correct, preventing exact precision measurement. Our sampling-based estimation mitigates this but carries inherent sampling uncertainty. Three domain experts cross-validated the static reachability labels, though expert judgment may vary. Additionally, AutoSQL emits wildcards when dependencies cannot be resolved through pattern-based search or when values are determined by external sources at runtime, and may miss SQL variants when complex control-flow conditions prevent complete path enumeration.

External Validity. We evaluated our approach on five Go repositories (38K ~ 845K LoC) with two ORM frameworks (XORM and Beego), one custom SQL builder (Go-Admin), and three SQL dialects (SQLite, MySQL, PostgreSQL). Imperative ORM frameworks share a common high-level design such as method-chain builders, struct-tag column mappings, and lifecycle hooks, so AutoSQL's retrieval strategy transfers across frameworks. However, each ORM has its own terminal APIs and conventions; adapting to a new ORM requires updating the framework-specific knowledge in the system prompt. Our quantitative evaluation is scoped to test-covered entry points; results should not be generalized to untested entry points without further validation. As our selected open-source projects are well-known and likely present in LLM pre-training corpora, data contamination is a potential concern. However, our evaluation dataset is derived from instrumented test execution and expert annotation, which is unlikely to appear in any pre-training corpus. The models are thus evaluated on their ability to reconstruct these SQL statements from source code, not to recall memorized answers.

7 Related Work

In this section, we discuss related work in the following two areas and highlight the gaps that AutoSQL addresses.

7.1 SQL Synthesis from Code

Static analysis techniques have been explored to synthesize or locate SQL queries from application code. Early work [7, 8] applies program synthesis to convert imperative code into equivalent SQL queries. SLocator [26] combines static analysis with information retrieval to locate SQL generation sites in Java applications. DBridge [25] constructs Java-to-database value flows through pointer analysis. These approaches are designed for Java's declarative ORM frameworks and cannot handle the stateful builder patterns and reflection-based mechanisms in Go ORMs.

7.2 LLM-Based Code Analysis and Auditing

While LLMs have been applied to code analysis, recent evaluations [11, 12] reveal they still struggle with real-world complexity and obfuscation. To mitigate this, various approaches decompose tasks for verification [37, 38], simulate pseudo-code execution [19], or translate formal specifications [46]. For repository-level auditing, tools like iCodeReviewer [32] and RepoAudit [18] target vulnerability detection via single-pass or ReAct-style inference. Additionally, several methods integrate LLMs with static analysis: IRIS [24] infers taint specifications, ErrorPrism [34] tracks error propagation, and RepoGraph [31] leverages code graphs for structural navigation. However, these existing approaches primarily focus on security vulnerabilities or general code comprehension, lacking the tailored mechanisms required for the semantic reconstruction of dynamic database operations from imperative ORM code.

8 Conclusion

In this paper, we propose AutoSQL, an LLM-based agent for extracting SQL templates from imperative ORM code. Imperative ORMs obscure SQL behind dynamic method-call sequences, with a substantial portion of statements relying on runtime features, such as dynamic table resolution, stateful builders, and lifecycle hooks, that evade deterministic static analysis. Existing code audit agents fail to retrieve the non-structural context these patterns demand. To address these challenges, AutoSQL combines structural code navigation with on-demand, pattern-based search to resolve dynamic dependencies and reconstruct SQL templates that static methods cannot reach. Evaluation on a runtime-traced benchmark (with 579 test-covered entry points) from five large-scale repositories demonstrates that AutoSQL achieves 75.13%–79.76% recall, surpassing deterministic static analysis by up to 20.49 percentage points and existing code audit agents by 4.57–49.45 percentage points.

Data Availability

All artifacts and data in this paper are publicly available on Zenodo at https://doi.org/10.5281/zenodo.20967213.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62402536).

References

[1] Anthropic. 2025. Claude 4.5 Sonnet System Card. https://www.anthropic.com. Accessed: 2026-01-15.

[2] Apache Software Foundation. 2022. Apache Answer: A Q&A platform software for teams at any scales. https://github.com/apache/answer

[3] Joy Arulraj. 2024. sqlcheck: Automatically identify anti-patterns in SQL queries. https://github.com/jarulraj/sqlcheck. Accessed: 2025-11-24.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (May 2018), 39 pages. doi:10.1145/3182657

[5] Beego. 2025. ORM | Beego. https://beegodoc.com/en-US/developing/orm/. Accessed: 2025-11-24.

[6] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 1001–1012. doi:10.1145/2568225.2568259

[7] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2012. Inferring SQL Queries Using Program Synthesis. arXiv:1208.2013 [cs.PL] https://arxiv.org/abs/1208.2013

[8] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 3–14. doi:10.1145/2491956.2462180

[9] CloudWeGo. 2025. ABCoder: An AI-oriented code-processing framework for enhancing coding context for large language models. https://github.com/cloudwego/abcoder

[10] Derek Colley, Clare Stanier, and Md Asaduzzaman. 2018. The Impact of Object-Relational Mapping Frameworks on Relational Query Performance. In *2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE)* (Southend, Essex, UK). IEEE, Piscataway, NJ, USA, 47–52. doi:10.1109/iCCECOME.2018.8659222

[11] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2025. Vulnerability Detection with Code Language Models: How Far Are We?. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, Piscataway, NJ, USA, 1729–1741. doi:10.1109/ICSE55347.2025.00038

[12] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. 2024. Large language models for code analysis: do LLMs really do their job?. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) *(SEC '24)*. USENIX Association, USA, Article 47, 18 pages. https://dl.acm.org/doi/10.5555/3698900.3698947

[13] Gitea. 2016. Gitea: Git with a cup of tea, a painless self-hosted all-in-one software development service. https://github.com/go-gitea/gitea

[14] GoAdminGroup. 2018. GoAdmin: A Golang framework helps gopher to build a data visualization and admin panel in ten minutes. https://github.com/GoAdminGroup/go-admin

[15] Grafana Labs. 2013. Grafana: The open and composable observability and data visualization platform. https://github.com/grafana/grafana

[16] Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. 2020. Featherweight go. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 149 (Nov. 2020), 29 pages. doi:10.1145/3428217

[17] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/2670979.2670986

[18] Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. 2025. RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025 (Proceedings of Machine Learning Research)*, Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (Eds.). PMLR / OpenReview.net. https://proceedings.mlr.press/v267/guo25n.html

[19] Yu Hao, Weiteng Chen, Ziqiao Zhou, and Weidong Cui. 2023. E&V: Prompting Large Language Models to Perform Static Analysis by Pseudo-code Execution and Verification. arXiv:2312.08477 [cs.SE] https://arxiv.org/abs/2312.08477

[20] Harbor. 2016. Harbor: An open source trusted cloud native registry project that stores, signs, and scans content. https://github.com/goharbor/harbor

[21] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, et al. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=VtmBAGCN7o

[22] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 111 (April 2024), 26 pages. doi:10.1145/3649828

[23] Yichen Li, Yulun Wu, Jinyang Liu, Zhihan Jiang, Zhuangbin Chen, Guangba Yu, and Michael R. Lyu. 2025. COCA: Generative Root Cause Analysis for Distributed Systems with Code Knowledge. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, Piscataway, NJ, USA, 1346–1358. doi:10.1109/ICSE55347.2025.00234

[24] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. In *The Thirteenth International Conference on Learning Representations*. International Conference on Learning Representations, ICLR, Singapore, 24 pages. https://openreview.net/forum?id=9LdJDU7E91

[25] Yufei Liang, Teng Zhang, Ganlin Li, Tian Tan, Chang Xu, Chun Cao, Xiaoxing Ma, and Yue Li. 2025. Pointer Analysis for Database-Backed Applications. *Proc. ACM Program. Lang.* 9, PLDI, Article 204 (June 2025), 25 pages. doi:10.1145/3729307

[26] Wei Liu and Tse-Hsun Chen. 2023. SLocator: Localizing the Origin of SQL Queries in Database-Backed Web Applications. *IEEE Transactions on Software Engineering* 49, 6 (2023), 3376–3390. doi:10.1109/TSE.2023.3253700

[27] Martin Lorenz, Günter Hesse, and Jan-Peer Rudolph. 2016. Object-relational Mapping Revised - A Guideline Review and Consolidation. In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - ICSOFT-EA* (Lisbon, Portugal). INSTICC, SciTePress, Setubal, Portugal, 157–168. doi:10.5220/0005974201570168

[28] mini-swe-agent. 2025. mini-swe-agent. https://mini-swe-agent.com

[29] Moonshot AI. 2025. Kimi k2: Large Language Model Technical Report. https://platform.moonshot.cn. Model version 0905. Accessed: 2026-01-15.

[30] OpenAI. 2025. Introducing Codex. https://openai.com/index/introducing-codex/ https://openai.com/index/introducing-codex/.

[31] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2025. RepoGraph: Enhancing AI Software Engineering with Repository-Level Code Graph. In *13th International Conference on Learning Representations, ICLR 2025*. International Conference on Learning Representations, ICLR, Singapore, 30361–30384.

[32] Yun Peng, Kisub Kim, Linghan Meng, and Kui Liu. 2025. iCodeReviewer: Improving Secure Code Review with Mixture of Prompts. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Seoul, Korea, Republic of). IEEE Press, 3204–3215. doi:10.1109/ASE63991.2025.00264

[33] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. 2023. Generative Type Inference for Python. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Echternach, Luxembourg) *(ASE '23)*. IEEE Press, Piscataway, NJ, USA, 988–999. doi:10.1109/ASE56229.2023.00031

[34] Junsong Pu, Yichen Li, Zhuangbin Chen, Jinyang Liu, Zhihan Jiang, Jianjun Chen, Rui Shi, Zibin Zheng, and Tieying Zhang. 2025. ErrorPrism: Reconstructing Error Propagation Paths in Cloud Service Systems. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 3534–3545. doi:10.1109/ASE63991.2025.00292

[35] SWE-bench. 2026. SWE-bench. https://www.swebench.com/ Accessed: 2026-06-28.

[36] The Go Security Team. 2022. Go vulnerability management. https://go.dev/blog/vuln. Accessed: 2025-11-24.

[37] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. 2024. LLMDFA: analyzing dataflow in code with large language models. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 4181, 30 pages.

[38] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, and Xiangyu Zhang. 2024. Sanitizing Large Language Models in Bug Detection with Data-Flow. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 3790–3805. doi:10.18653/v1/2024.findings-emnlp.217

[39] Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025. CodeRAG-Bench: Can Retrieval Augment Code Generation?. In *Findings of the Association for Computational Linguistics: NAACL 2025*. Association for Computational Linguistics, Albuquerque, New Mexico, 3199–3214. doi:10.18653/v1/2025.findings-naacl.176

[40] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837. doi:10.52202/068431-1800

[41] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE037 (June 2025), 24 pages. doi:10.1145/3715754

[42] Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. 2025. Live-SWE-agent: Can Software Engineering Agents Self-Evolve on the Fly?

arXiv:2511.13646 [cs.SE] https://arxiv.org/abs/2511.13646

[43] XORM. 2025. XORM - A simple and powerful ORM for Go. https://xorm.io/. Accessed: 2025-11-24.

[44] 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. *Advances in Neural Information Processing Systems* 37 (2024), 125 pages. doi:10.52202/079017-1601

[45] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*. International Conference on Learning Representations, ICLR, Kigali, Rwanda, 33 pages. https://openreview.net/forum?id=WE_vluYUL-X

[46] Mingwei Zheng, Danning Xie, Qingkai Shi, Chengpeng Wang, and Xiangyu Zhang. 2025. Validating Network Protocol Parsers with Traceable RFC Document Interpretation. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA078 (June 2025), 23 pages. doi:10.1145/3728955

Received 2026-03-26; accepted 2026-06-18