SQLens: Continuous Code-to-SQL Visibility in the Wild

Xiao Yang
Alibaba Cloud
Hangzhou, China
ruanbu.yx@alibaba-inc.com

Mo Sha
Alibaba Cloud
Singapore
shamo.sm@alibaba-inc.com

Yiran Li
Alibaba Cloud
Beijing, China
yiranli.lyr@alibaba-inc.com

Suyang Zhong*
National University of Singapore
Singapore
suyang@u.nus.edu

Sheng Wang
Alibaba Cloud
Singapore
sh.wang@alibaba-inc.com

Fangyuan Zhou
Alibaba Cloud
Hangzhou, China
fory@alibaba-inc.com

Feifei Li
Alibaba Cloud
Hangzhou, China
lifeifei@alibaba-inc.com

Abstract

Modern Internet-scale services evolve, yet database interactions are increasingly obscured by languages, frameworks, and abstraction layers. This loss of visibility weakens the link between code changes and SQL behavior, leading to performance regressions, security risks, and costly diagnosis. **SQLens** is a practical system that restores auditable, end-to-end visibility between application code and executed SQL in large, heterogeneous codebases. It combines static program analysis with LLM-guided reasoning in a closed-loop workflow: starting from database emission sites, cooperative agents traverse control and data flows to reconstruct SQL construction and parameter binding. The system refines these mappings using SQL logs collected over time and stores them in a versioned knowledge layer supporting code-to-SQL lookup and SQL-to-code attribution. Deployed at Alibaba Group and evaluated on diverse open-source services, **SQLens** achieves high precision and recall in code-to-SQL reconstruction, reduces SQL-incident resolution time, and, when integrated with continuous integration pipelines and runtime observability, supports detection and mitigation of SQL issues, enabling continuous SQL governance at industrial scale.

CCS Concepts

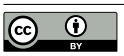
• **Information systems** → **Query languages**; **Database administration**; *Enterprise information systems.*

Keywords

Code-to-SQL visibility, SQL provenance, LLM

ACM Reference Format:
Xiao Yang, Mo Sha, Yiran Li, Suyang Zhong, Sheng Wang, Fangyuan Zhou, and Feifei Li. 2026. **SQLens**: Continuous Code-to-SQL Visibility in the Wild. In *Companion of the International Conference on Management of Data (SIGMOD Companion '26), May 31-June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3788853.3803091

*Suyang contributed to this work while interning at Alibaba Cloud.



SIGMOD Companion '26, Bengaluru, India

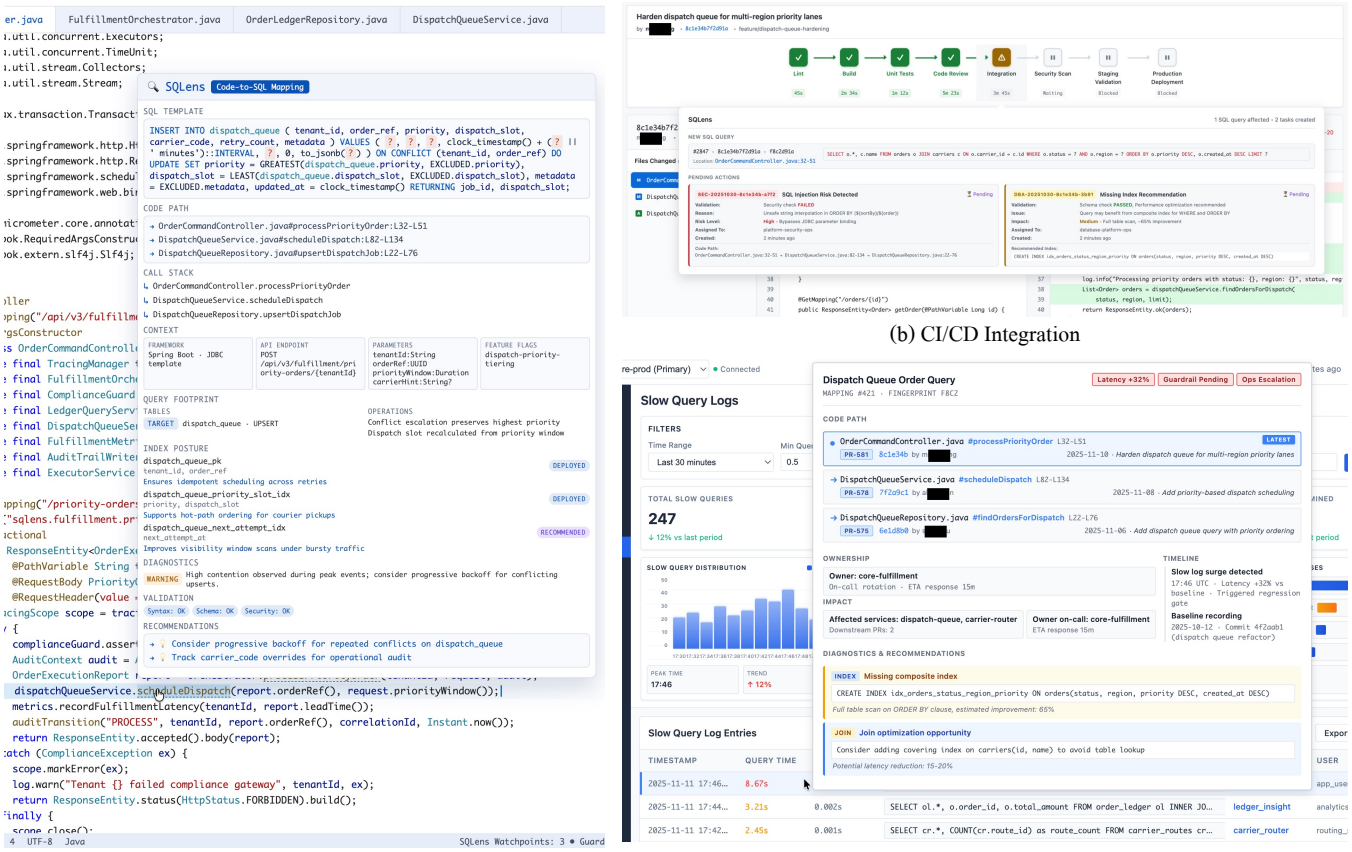
ACM ISBN 979-8-4007-2450-3/2026/05
https://doi.org/10.1145/3788853.3803091

1 Introduction

Large-scale Internet companies such as Alibaba Group operate complex software ecosystems that underpin e-commerce, payments, logistics, recommendation, and cloud services [10, 43]. These ecosystems are built on diverse languages and frameworks under microservice and multi-tenant architectures, where business logic spans multiple layers (APIs, service modules, data mappers, and shared libraries) and forms dense interdependencies across services [25, 42, 63]. Database interactions remain the foundation of system reliability and performance [36]. Yet visibility into how application code interacts with databases is increasingly difficult to sustain—a loss that directly affects operational safety and business continuity [21, 29]. In practice, even a single inefficient query or missing index can cascade across components, causing performance regressions and user-visible degradation [12, 26, 52, 78].

This need for reliability is increasingly undermined by the loss of transparency between source code and executed SQL [14, 46, 56, 64]. Modern large-scale systems depend on heterogeneous language stacks such as Java with Spring or MyBatis, Python with Django ORM, Node.js with TypeORM, and Go with lightweight SQL drivers. Each framework defines distinct conventions for query construction and parameter binding, while layered abstractions such as query builders, data mappers, configuration templates, and feature toggles further distance developers from the SQL behavior observed at runtime [3, 13, 37, 72]. Although these abstractions enhance modularity and productivity, they obscure the provenance of query execution. Consequently, engineers and reliability teams struggle to trace which code paths generate specific queries or how configuration and code changes alter data-access patterns. This erosion of linkage between code and SQL disrupts the feedback loop between code evolution and database performance, impeding large-scale preventive governance [73].

The visibility gap affects the entire software lifecycle. During code review, developers cannot tell whether a change introduces new queries or modifies access paths, letting inefficient or unsafe SQL reach production. Because test environments seldom match real data distributions, regressions often appear only after deployment [60]. In operation, slow queries and anomalies delay diagnosis as teams manually link logs, traces, and code [47]. Over time, refactoring and staff turnover erode knowledge of SQL behavior, raising the cost of each issue. Without a durable mapping between code and emitted SQL, organizations face recurring regressions, escalating maintenance costs, and persistent governance blind spots.



(a) IDE Code-to-SQL Visibility

(b) CI/CD Integration

(c) Slow Query Diagnostics

Figure 1: SQLens provides code-to-SQL visibility across the software lifecycle.

Closing this gap is beyond the reach of deterministic and static methods alone: production SQL is shaped by dynamic control flow, configuration-driven templates, and runtime composition—scenarios that even sophisticated static analysis struggles with and that go beyond cross-language coverage. This motivates semantics-aware, agent-based reasoning in addition to rule-based analysis.

Existing approaches such as manual review, static analysis, dynamic monitoring, provenance, and LLM-based reasoning each illuminate part of the problem but, in isolation, fail to provide a persistent, auditable code-to-SQL visibility layer that evolves with the codebase. The challenge is to establish a reliable bridge between application code and its resulting SQL behavior. In large, heterogeneous codebases, database access logic is rarely explicit and emerges from the interplay of these factors; addressing it requires discovering, inferring, and maintaining the mapping at scale without intrusive instrumentation or annotation, while remaining explainable and versioned. The difficulty lies not only in constructing these mappings once but in sustaining them under perpetual change, a defining property of large-scale development.

We present **SQLens**, a closed-loop, self-correction system that employs LLMs to restore code-to-SQL visibility in large and continuously evolving codebases. At a high level, **SQLens** is grounded in a fundamental intuition: solving a problem is often far more difficult than verifying a solution. While LLMs are capable of reasoning about code semantics and processing large numbers of tasks, their single-pass outputs are frequently approximate or unreliable. To address this, **SQLens** designs a workflow in which the solving phase produces preliminary results at scale, and the comparatively cheap and reliable verification phase refines and distills these results. More concretely, **SQLens** scans repositories to identify SQL emission sites such as ORM executors and driver invocation points. From each emission site, cooperative agents traverse control and data flows in both directions to collect the contextual information that shapes query construction. Guided by a ReAct-style reasoning loop, an LLM infers SQL templates together with their parameter bindings. A backward validation is then performed on SQL logs collected over time, validating whether each emitted SQL can be traced to its source code, and feeding confirmed results back into the versioned knowledge layer. This versioned knowledge layer, in turn, increases the accuracy and fidelity of code-to-SQL reconstruction.

The outcome is a persistent, queryable code-to-SQL index that preserves bidirectional visibility between application code and emitted SQL. It supports forward lookup from code to expected SQL and reverse lookup from observed SQL to likely source locations. Engineering teams rely on this layer in continuous integration for pull-request pre-review, in production for incident diagnosis and root-cause analysis, and in long-term governance to track evolving access patterns. Figure 1 shows **SQLens** integrated at three touchpoints—within the IDE, in CI/CD pull-request views, and on DBA slow-log consoles—exposing bidirectional code–SQL context across

development and operations. By unifying LLM-guided code-to-SQL reconstruction with SQL-to-code refinement, **SQLens** provides an explainable, versioned mapping that evolves with the codebase and complements tuning and automated physical design.

Deployed at Alibaba Group, **SQLens** integrates with CI pipelines and runtime visibility systems to provide pre-merge governance and post-deployment diagnostics. We evaluate **SQLens** on ten open-source, database-backed applications across Java, Ruby on Rails, and Python/Django. Our study examines how accurately **SQLens** reconstructs runtime SQL, how well it localizes queries to their originating code, and how it scales to industrial-scale codebases. Across a diverse Java benchmark suite, **SQLens** identifies most dynamically executed queries and, in several applications, almost always ranks the true origin within a small set of candidates. On ORM-heavy Ruby and Python applications, **SQLens** remains effective despite extensive framework indirection and, compared to specialized analyzers such as DBridge and SLocator, offers competitive SQL identification and localization while completing analysis.

The contributions of this paper are summarized as follows:

- We formulate the code-to-SQL visibility problem and derive design principles for continuous, explainable, and auditable visibility in polyglot services.
- We present **SQLens**, an agent-based, verification-informed analysis that builds a versioned, bidirectional code-to-SQL index with forward and reverse lookup APIs. Its workflow fuses static program facts with inference and refinement to keep the index accurate as code evolves.
- We assess **SQLens** on diverse, database-backed applications, demonstrating robust SQL identification and localization across heterogeneous stacks and realistic workloads.

2 Motivation and Problem Statement

Modern Internet-scale services evolve rapidly under agile development and automated deployment [23, 59]; in heterogeneous codebases, deep abstractions and diverse data-access frameworks obscure database interactions critical to correctness, tail latency, and cost. We describe the resulting code–SQL visibility gap and distill the requirements and scope that guide **SQLens**.

2.1 Problem Statement

At Alibaba's scale, application development and data management span thousands of repositories and microservices maintained by distributed teams. These systems change daily under continuous deployment, where even small updates to ORM logic or configuration can propagate into large volumes of production queries. Diversity in language stacks improves development speed but also fragments database-access semantics across layers.

Figure 2 illustrates the disconnect between the Code World and the SQL World. Even with mature CI/CD and monitoring, reviewers cannot inspect SQL effects in pull requests, staging rarely matches production data distributions, and runtime signals often lack code provenance [5, 19, 24, 58]. Together, these factors create a visibility gap: the loss of end-to-end insight into how code changes shape SQL behavior, making the production SQL footprint difficult to explain or reproduce. The impact is immediate: reviewers lack reliable evidence of SQL impact, and post-deployment dashboards surface

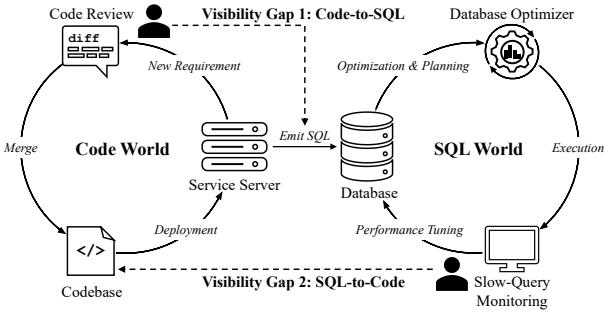


Figure 2: The visibility gap between code and SQL.

issues without code attribution, forcing manual correlation across signals. The gap breaks the feedback loop between code changes and database performance and keeps teams reactive. The objective of **SQLens** is to restore and maintain a persistent, auditable code–SQL mapping throughout the software lifecycle.

2.2 Limitations of Existing Practices

Over time, organizations have assembled a range of practices to reason about database access in complex systems. These practices include manual review, static analysis, dynamic monitoring, and model-based reasoning. Each method captures a partial view of data-access behavior and operates under different trade-offs. However, none maintains consistent visibility across the full lifecycle.

Manual review remains a basic safeguard. Senior engineers inspect code changes with the help of style conventions, checklists, and domain knowledge, and try to anticipate performance or correctness risks before a change is merged [11, 27, 65]. This approach can be effective for high-impact patches, but it does not keep up with the volume and speed of modern development. Review quality varies across teams, and reviewers rarely see the workload or data distributions that shape how queries behave in production.

Static analysis and linting frameworks [20, 21, 70] automate pattern detection and rule checking in source code [22, 67] and are widely integrated into CI pipelines [53]. Yet they rarely reconstruct a durable mapping from code to emitted SQL. They can flag hard-coded queries and unsafe operations, but struggle with scenario classes that challenge even advanced static analysis: dynamic control flow and dispatch (callbacks, reflection, multi-branch logic), configuration-driven templates (XML/YAML fragments and feature-flagged variants), and runtime query composition (string assembly and query-builder/ORM DSL patterns). Even state-of-the-art, language-specific tools remain tied to a single language and framework and degrade when these patterns combine [45, 46]. These limitations align with known SQL injection risks [34, 35]. Maintaining accurate rules across languages and ORMs is expensive, and static reasoning cannot capture runtime variation driven by parameters, feature flags, or deployment-specific configuration.

Dynamic monitoring tools such as distributed tracing and application performance monitors [31, 40, 48] observe SQL executions under live workloads and provide accurate performance signals, but they are runtime-only and rarely attribute queries to generating code paths; incident response therefore still requires manual correlation across traces, logs, and repositories. Database provenance

and lineage systems [17, 33, 55] explain result derivations but not the code locations that constructed the SQL.

Recent work applies large language models to code reasoning and SQL inference [32, 61]. LLMs capture richer semantics and transfer across languages, but their outputs are probabilistic and approximate; without supervision, reference data, and version control, they are difficult to audit and cannot serve as the sole mechanism for governing production database access.

Viewed together, these practices share the same limit: manual review does not scale, static and dynamic tools degrade under changing abstractions, and model-based reasoning reduces determinism and auditability. None provides an end-to-end, explainable, and auditable link between evolving code and the SQL it produces.

2.3 Design Goals and Principles

SQLens is guided by functional and non-functional requirements distilled from Alibaba Group's production experience for industrial-scale code-to-SQL visibility. Functionally, **SQLens** must support bidirectional mapping between source code and executed SQL to enable forward impact analysis and provenance-aware reverse lookup. Given a SQL instance from slow logs or traces, it should identify the originating code region together with CI metadata (introducing commit and pull request, author, timestamp, and service/module). **SQLens** should integrate with CI and pull-request workflows to detect SQL-impacting changes before deployment and support incremental updates by recomputing only affected mappings. Because inferences can be uncertain, the system needs feedback that assigns confidence and refines mappings to a reliability standard suitable for human inspection. Non-functionally, **SQLens** must be low-intrusion, requiring no business-logic changes and adding negligible runtime overhead. Outputs must be explainable and traceable, with clear provenance per mapping. Artifacts must be versioned and auditable to enable reproduction and rollback. The system must scale to heterogeneous codebases with millions of lines of code and thousands of daily commits.

These requirements reflect production settings where safety, auditability, and maintainability matter as much as detection accuracy. They call for a design that fits existing workflows with minimal disruption and motivate the architecture of **SQLens**.

2.4 Scope and Assumptions

SQLens targets SQL behavior that is statically traceable and semantically inferable, covering most operational risks in large-scale OLTP while remaining feasible under continuous development. The current implementation supports mainstream stacks in Java, Python, Node.js, and Go and covers SQL emitted through ORMs, data mappers, configuration-driven templates, and dynamic string assembly. Highly dynamic SQL arising from runtime code generation, implicit propagation across chained microservices, or closed-source dependencies is excluded from automated inference and handled via supervised review. In summary, **SQLens** restores auditable code–SQL visibility in complex, heterogeneous environments where existing approaches are partial and degrade under continuous evolution, providing a scalable mechanism for reconstruction and maintenance across the software lifecycle.

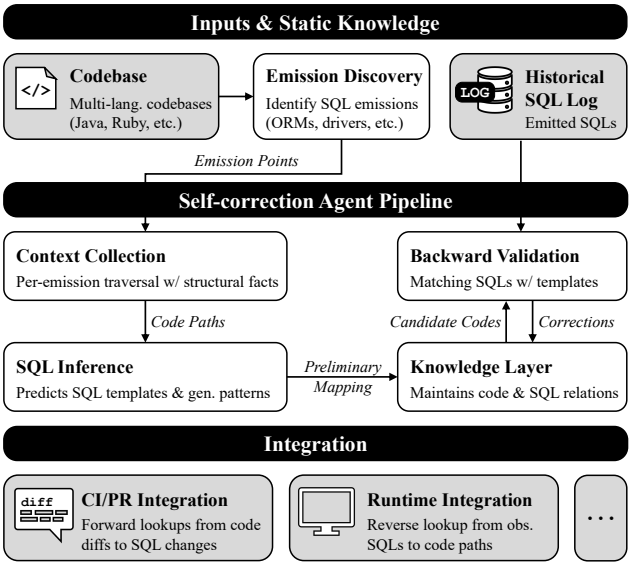


Figure 3: Overview of SQLens.

3 SQLens Architecture

SQLens is a closed-loop, self-correction system that maintains code-to-SQL visibility across heterogeneous and continuously evolving codebases. It combines the generalization ability of large language models with the structural precision of static program analysis. At the core of the design is a ReAct-style agent pipeline that starts from SQL emission sites in the application, traverses the surrounding code paths to recover how queries are constructed, and uses a feedback path from observed SQL back to code to verify and refine the inferred mappings as the codebase evolves. The outputs of this pipeline are maintained in a versioned code-to-SQL index. Each entry links code paths with normalized SQL behavior, contextual information, and validation results, and carries version metadata derived from the underlying repositories.

3.1 Architecture Overview

To realize this design, **SQLens** organizes its components into three layers (Figure 3): a static input layer, an agent pipeline, and a persistent knowledge-and-integration layer. The top layer ingests multi-language repositories, configuration files, and ORM metadata. A scanning agent identifies SQL emission sites (e.g., ORM executors, query builders, and driver interfaces) by prompting an LLM to invoke search tools such as `grep` and interpret matches, yielding broad coverage while avoiding omissions from single-pass, long-context reasoning. In parallel, a static analysis component extracts call graphs, symbol tables, and structures and normalizes them into a unified fact schema for downstream agents.

SQL emission sites form the analytical entry points for the agent pipeline. For each site, a controller schedules ReAct-style agents that traverse caller and callee paths, collect parameter and control-flow context, and infer SQL templates and bindings. Agents share intermediate state through a lightweight memory layer and consult static facts for constraint checking. This traversal produces preliminary code-to-SQL mappings, which are later validated and refined using observed SQL from execution logs.

```java
// src/main/java/com/acme/user/UserController.java:L23-36
@RestController
class UserController {
  @Autowired UserService svc;

  @GetMapping("/api/products")
  Page<UserDTO> list(
      @RequestParam Optional<Long> categoryId,
      @RequestParam Optional<String> sortBy,
      @RequestParam Optional<String> order,
      @RequestParam int page, int size) {
    return svc.listProducts(categoryId, sortBy, order, page, size);
  }
}
```

```java
// src/main/java/com/acme/user/UserService.java:L40-62
@Service
class UserService {
  @Autowired FeatureFlags flags;
  @Autowired UserMapper mapper;

  Page<UserDTO> listProducts(Optional<Long> categoryId,
        Optional<String> sortBy, Optional<String> order,
        int page, int size) {
    boolean includeCategory = flags.isEnabled("include_category");
    int offset = (page - 1) * size;
    String sortCol = sortBy.orElse("created_at");
    String sortDir = order.orElse("DESC");

    return mapper.selectProducts(
        categoryId.orElse(null),
        sortCol, sortDir, includeCategory, offset, size);
  }
}
```

```java
// src/main/java/com/acme/user/UserMapper.java:L10-16
@Mapper
interface UserMapper {
  List<Product> selectProducts(Long categoryId,
      String sortBy, String order,
      boolean includeCategory, int offset, int size);
}
```

```xml
<!-- src/main/resources/mappers/ProductMapper.xml:L120-150 -->
<select id="selectProducts">
  SELECT p.id, p.name, p.price, p.created_at
  <if test="includeCategory">
    , c.name AS category_name
    FROM products p LEFT JOIN categories c ON c.id = p.category_id
  </if>
  <if test="!includeCategory">
    FROM products p
  </if>
  <if test="categoryId != null">WHERE p.category_id = #{categoryId}</if>
  ORDER BY ${sortBy} ${order}
  LIMIT #{size} OFFSET #{offset}
</select>
```

Figure 4: Running example: controller, service, mapper, and dynamic SQL template for the product-listing API.

The bottom layer stores these mappings in a versioned knowledge base, recording the code path, normalized SQL, context, validation results, and version metadata. It supports bidirectional lookup from code to expected SQL and from observed SQL to candidate code regions, and serves as the anchor point for the analysis pipeline and the integrations discussed below.

3.2 Running Example

We illustrate the operation of **SQLens** using a production-style Java stack that implements a paginated product-listing API. Requests follow a standard three-layer architecture: a Spring MVC controller exposes the HTTP endpoint, a service layer applies feature flags and composes query parameters, and a MyBatis mapper delegates to an XML template that builds dynamic SQL. Mapper interfaces annotated with `@Mapper` are compiled into framework proxies whose invocations eventually reach `PreparedStatement.executeQuery`. During repository scanning, **SQLens** uses the LLM to recognize this framework convention and to treat `@Mapper` methods as SQL *emission sites*, that is, entry points that anchor subsequent path expansion and reasoning. The scanning agent derives simple regex filters to locate likely emission sites, validates them against nearby code context, and refines them iteratively, instead of relying on a single long-context pass over the entire codebase.

Figure 4 shows the controller, service, mapper interface, and the corresponding MyBatis XML template. From the identified emission site, **SQLens** applies the traversal and inference stages described in Section 3.3 to expand the reachable call chain toward the controller and reconstruct the contextual elements that influence SQL generation. These include a feature flag `include_category` that guards a conditional `LEFT JOIN` on `categories`, an optional `categoryId` predicate, and pagination propagated from `page`/`size` to `OFFSET`/`LIMIT`. In many operational settings, full access to source files and configuration is not available. The backward verification stage compensates for these gaps by linking collected SQL logs to their originating code and extracting essential details, which are stored in a project-level knowledge base (`project_knowledge`) for downstream analysis. This three-layer structure is representative of enterprise systems in which query behavior arises from the combined effect of framework templates, runtime configuration, and user input rather than any single code component.

The `project_knowledge` and the `mapping_tuple` produced during analysis are shown in Figure 5. The `project_knowledge` summarizes historical SQL patterns observed through backward verification. Guided by this information, the `mapping_tuple` records the reconstructed SQL template, the end-to-end code path, dynamic fragments, validation results, and system recommendations. Because `${sortBy}` and `${order}` bypass JDBC parameter binding, crafted inputs can alter ordering semantics or reference unintended identifiers, a pattern common in template-driven ORMs. For instance, if `sortBy` is set to `"created_at; DROP users; --"`, the concatenated template introduces a statement terminator and comment, enabling injection in environments that permit multi-statements. **SQLens** marks these fragments as dynamic, fails the `security` validator, and queues the case for supervised review. In contrast, the `syntax` and `schema` validators succeed, showing that the template has no SQL syntax errors and that the referenced tables and columns exist. Based on the inferred template, **SQLens** expects at least two supporting indexes for filtering and ordering paths, and the resulting mapping can be queried in both directions by the integrations described later in this section.

3.3 Analysis Pipeline

Building on this architecture, **SQLens** detects SQL emission sites, traces their code paths, infers SQL behavior, and adds validated mappings to the knowledge layer, forming a feedback loop that stays aligned with the evolving codebase.

Emission Discovery. The workflow begins by scanning code repositories with an LLM-based detector that identifies SQL emission points such as ORM executors, driver calls, or raw query constructors. In **SQLens**, this detector first learns from project-specific

```yaml
project_knowledge:
  - "Prioritize explicit column list over wildcard (*) in SELECT statement."

mapping_tuple:
  code_path:
    - src/main/java/com/acme/user/UserController.java#list:L29-36
    - src/main/java/com/acme/user/UserService.java#listProducts:L46-66
    - src/main/java/com/acme/user/UserMapper.java#selectProducts:L10-20
    - src/main/resources/mappers/ProductMapper.xml#selectProducts:L120-150

  sql:
    template: |
      SELECT p.id, p.name, p.price, p.created_at
      FROM products p
      /* [opt] WHERE p.category_id = ? */
      ORDER BY ${sortBy} ${order}
      LIMIT ? OFFSET ?
    bindings:
      - categoryId:Long (optional)
      - size:Int
      - offset:Int
    dynamic_segments:
      - ORDER BY ${sortBy} ${order}
    pattern:
      - table: products
      - target: id, name, price, created_at
      - order_by: true
      - group_by: null

  context:
    framework: Spring+MyBatis
    api: GET /api/products
    feature_flags:
      - include_category (true, UserService.java:L49)
    params:
      - categoryId? (UserController.java:L29 -> ProductMapper.xml:L124)
      - sortBy?[created_at] (UserController.java:L30 -> ProductMapper.xml:L125)
      - order?[DESC] (UserController.java:L31 -> ProductMapper.xml:L125)
      - size (UserController.java:L32 -> UserMapper.xml:L221)
      - offset (UserService.java:L56 -> UserMapper.xml:L221)
    branches:
      - includeCategory -> LEFT JOIN categories (ProductMapper.xml:L121-122)

  validation:
    syntax: OK
    schema: OK
    security:
      status: FAIL
      reason: "unsafe ${sortBy}/${order} interpolation in ORDER BY"
    review:
      status: pending
      assigned_to: security-ops-team

  metadata:
    version: pr581@41c29f7
    timestamp: 2025-10-30T18:43:12Z
    recommendations:
      - add index on products(category_id)
      - add index on products(created_at)
```

Figure 5: Running example: persisted code-to-SQL mapping tuple with validation results and the associated project-level knowledge synthesized during backward validation.

configuration and code samples, then derives grep-style matching rules that can be executed by standard tools. It applies and refines these rules iteratively, inspecting matches and reviewing related files, including configuration and build scripts, to validate candidates and filter false positives. As illustrated by the `@Mapper` pattern in Figure 4, this setup lets the detector capture framework conventions without per-framework rules. Each detected site then becomes the entry point for traversal.

Path Traversal and Context Collection. For every emission site, the system spawns a traversal agent that explores the code paths leading to that site by using ReAct-style agents guided by hints from the static analysis tool. The agents walk both forward and backward along call graphs to gather context such as parameter flows, conditional logic, and configuration dependencies. Static analysis is used only to extract structural facts, including call graphs and symbol references, which are standard outputs of common analysis frameworks. The detailed, language-specific control and data-flow reasoning is handled by the LLM agents. They coordinate through shared memory and stop once the evidence satisfies the sufficiency criteria for SQL inference.

SQL Inference via ReAct Agents. Once sufficient context has been collected, the agent transitions from exploration to inference. Given the accumulated context and emission metadata, the LLM predicts an SQL template with corresponding parameter bindings that represent the likely query issued at runtime. The ReAct loop then iteratively reasons about missing evidence, consults static hints, and refines this prediction until the inferred SQL stabilizes or meets the confidence thresholds. After all emission sites are processed, the system outputs an initial code-to-SQL mapping together with pattern annotations that support SQL-to-code filtering.

Backward Validation and Knowledge Integration. The workflow next performs backward validation, moving from SQL to code using database logs from the service under analysis. Because each logged query can be tied to a specific code version via commit history, verification is often more tractable than forward inference. Given a SQL statement, the system filters candidates and identifies plausible emission sites, then applies validators for syntax, schema consistency, and engine compatibility; ambiguous cases are routed to lightweight manual review. This stage self-corrects by confirming and refining preliminary mappings and supports diagnosis by attributing online issues to responsible code. Confirmed mappings are stored in the persistent code-to-SQL index with provenance, confidence, and version metadata, enabling bidirectional lookup and incremental updates as the codebase evolves.

Taken together, these stages embed LLM reasoning in a controlled feedback loop, turning imperfect single-pass inference into verifiable, continuously maintained mappings. The system reconstructs and validates code–SQL relationships over time, leveraging the observation that verifying a candidate mapping is often easier and more reliable than generating one from scratch.

3.4 Data Model and Knowledge Layer

At the heart of **SQLens** is a versioned knowledge layer that captures and maintains the discovered relations between source code and SQL behavior. It acts both as a persistent metadata store for incremental updates and as a queryable foundation for analysis, compliance, and operational feedback.

Each entry in the knowledge layer is represented as a nested record with top-level fields `code_path`, `sql`, `context`, `validation`, and `metadata`. `code_path` encodes the call chain leading to a specific SQL emission point. `sql` stores the normalized SQL `template`, parameter `bindings`, and `dynamic_segments`. `context` aggregates language and framework information, API endpoints, schema references, propagated parameters, feature flags, and branch conditions. `validation` captures syntax, schema, and security outcomes together with review status. `metadata` includes the `version`, `timestamp`, and `recommendations` for performance or security (for example, index suggestions). Optional confidence scores can also be recorded under `metadata` when available.

Versioning and incremental recomputation keep this index synchronized with code evolution while avoiding full reprocessing. Only mappings affected by a change are updated, and prior versions are retained for auditability and time-travel analysis; implementation details are provided in Section 4.1.

The knowledge layer exposes bidirectional query interfaces. A forward lookup traces from code elements to the SQL statements they are expected to emit, enabling PR-time detection of SQL-affecting changes. A reverse lookup starts from an observed SQL statement, such as one extracted from slow-query logs, and retrieves its generating code context, commit history, and responsible developer. It supports both exact fingerprint matching and semantic similarity search over normalized SQL embeddings. The versioned index also enables lightweight analytics and continuous update tracking, such as observing how query patterns change across releases or how often specific validators fail.

3.5 Interfaces and Integrations

The knowledge layer of **SQLens** integrates with existing development, visibility, and governance workflows in large-scale production environments. Through lightweight interfaces and incremental synchronization, **SQLens** serves as a shared visibility service for developers, operators, and database administrators.

CI/PR Integration During continuous integration, **SQLens** performs forward lookups from modified code regions to identify potential SQL-affecting changes. Pull requests that introduce new or altered mappings are automatically annotated with warnings or links to the relevant entries in the code-to-SQL index. This feedback allows reviewers to assess correctness, performance risk, and security compliance of database interactions before merging. Each CI/PR check runs incrementally and in parallel, scoped to changed symbols and their adjacent emission sites, so that pipeline latency and overhead remain low.

Runtime Integration At runtime, **SQLens** supports reverse lookup from observed SQL statements to their corresponding code paths. The system integrates with APM traces, slow-query logs, and database performance dashboards to link operational anomalies back to their generating code. When a problematic query is detected in production, **SQLens** retrieves the associated mapping, commit history, and developer attribution, which accelerates root-cause analysis and post-mortem diagnosis and turns SQL-level monitoring into actionable, code-aware insights.

Dashboards and Analytics For operational governance, **SQLens** offers dashboards that summarize hot queries, mapping histories, and access-pattern trends across services. DBAs and performance engineers can trace how SQL behavior changes over time, analyze workload composition by framework or schema, and relate query shifts to release events. Aggregated statistics and confidence scores provide an organization-level view of database-access health and support capacity planning and policy enforcement.

APIs and Command-Line Tools The knowledge layer exposes RESTful and command-line interfaces for integration with external systems. These APIs support programmatic queries, batch exports, and periodic synchronization with internal data catalogs or compliance platforms. Developers can also invoke local lookups to trace SQL provenance directly from their IDE or testing environment, giving a consistent view of code-to-SQL behavior in both offline development and production contexts.

4 Implementation Details

SQLens is realized as a scalable production service that operationalizes the conceptual pipeline (Section 3). It is deployed as a distributed microservice system designed for resilience and low operational overhead, and it maintains continuous, versioned code-to-SQL mappings across hundreds of actively developed repositories with minimal human effort.

4.1 Engineering Architecture and Stack

SQLens is implemented as a distributed microservice system composed of five stateless modules: *scanner*, *constructor*, *validator*, *knowledge layer*, and *API gateway*. Each module is deployed as a containerized service managed by an internal job scheduler and communicates through message queues and gRPC endpoints. This modular layout supports independent scaling, isolates faults, and integrates with the corporate CI/CD pipeline. Figure 6 illustrates the system architecture and control flow.

Technology Stack and Deployment. The system combines static program analysis, LLM reasoning, and distributed orchestration. The scanner and validator interface with text matching utilities (for example `grep`) and static analysis tools (for example `pylsp`) to extract program facts and interact with an internal LLM service for semantic emission-point detection and SQL inference. The *constructor* runs an agent engine on a distributed task runtime with shared-memory coordination and checkpointing so that agents can reuse intermediate state when traversing large codebases. All modules run as stateless services connected through a distributed job queue and metadata bus. Each analysis task executes in an isolated agent instance that operates on repository snapshots pulled from internal Git mirrors. Redis holds transient agent state and context caches, while PostgreSQL stores confirmed mappings durably. Incremental analysis is initiated by Git hooks and CI events so that maintenance cost scales with the rate of code changes rather than the size of the repository.

Module Composition. These modules implement the functions of the conceptual pipeline described in Section 3 and their responsibilities are summarized below.

(1) Scanner. The scanner crawls registered repositories, parses configuration files, and invokes static analysis tools to extract program facts, including call graphs, function signatures, and variable bindings. Before active search, the LLM-based emission-point detector is guided to build an internal view of the project through an iterative verification loop. This phase relies on the LLM's language-agnostic reasoning and its trained knowledge of common ORM conventions and SQL generation patterns, so only limited project-specific hints are needed in uncommon cases. Repeated direct LLM search can produce large contexts and inconsistent judgments across files. To avoid this effect, the LLM uses pattern matching tools to surface candidate locations, which are then validated and refined. Each confirmed emission point becomes the starting point for downstream reasoning.

(2) Constructor. For each emission point, the engine spawns ReAct agents that traverse call paths to recover the contexts reaching that point. Agents use static analysis relations as priors and coordinate through shared memory to avoid redundant exploration. Traversal stops once the recovered context is sufficient to infer the emitted SQL. In addition to producing a preliminary code-to-SQL mapping, the constructor extracts SQL pattern annotations such as referenced tables, predicate targets, and other query attributes, which enable efficient SQL-to-code lookup in later analysis.

(3) Validator. Given collected SQL queries, the validator first performs SQL-to-code resolution by searching the preliminary mapping using SQL patterns and matching emitted queries to confirm their originating code segments. After a template is confirmed, a rule-based validator applies checks for syntactic correctness, schema conformance, and security policy compliance. Invalid or ambiguous cases are queued for human review through a web interface. All outcomes are versioned and stored for traceability. Detailed validation and supervision logic is described in Section 4.4.

(4) Knowledge Layer. The knowledge layer maintains a versioned store of validated mappings as nested records with top-level fields `code_path`, `sql`, `context`, `validation`, and `metadata`. Each record carries provenance hashes, feature-flag annotations, validator outcomes, and recommendations, and it supports both forward (code-to-SQL) and reverse (SQL-to-code) lookup. Versioning metadata links each record to its associated commit or pull request so the system can answer time-travel queries over SQL evolution.

(5) API Gateway. The gateway exposes REST and gRPC endpoints for integration with CI and pull request pipelines, visibility dashboards, and compliance systems. External clients use these endpoints to query mappings, trigger incremental scans, or retrieve SQL affecting diffs for automated review.

Versioning and Incremental Updates. The knowledge layer evolves alongside the codebase. When a pull request or commit alters relevant files, **SQLens** performs incremental recomputation over the affected regions instead of reprocessing the full repository. Updated mappings are appended as new versions and older ones are retained for audit and lineage analysis. This versioning model preserves historical visibility into how SQL changes over time and supports reproducible evaluation across revisions. Incremental recomputation is driven by version control diffs that intersect static analysis based dependency slices for the affected emission sites. The process is idempotent and failed attempts are retried when validator checks report transient errors.

Repository Integration. SQLens integrates with internal Git and code review platforms. Each repository provides a configuration file that declares its ORM frameworks, build paths, and schema references. During scanning, **SQLens** resolves inter module dependencies and performs cross repository traversal when call chains extend across service boundaries. Analysis results and warnings are surfaced as inline annotations that link each SQL related change to its inferred query context and validation outcome.

4.2 Development Infrastructure Integration

SQLens integrates with Alibaba's development, deployment, and monitoring ecosystem to provide continuous visibility into SQL behavior without changing established workflows. It connects to

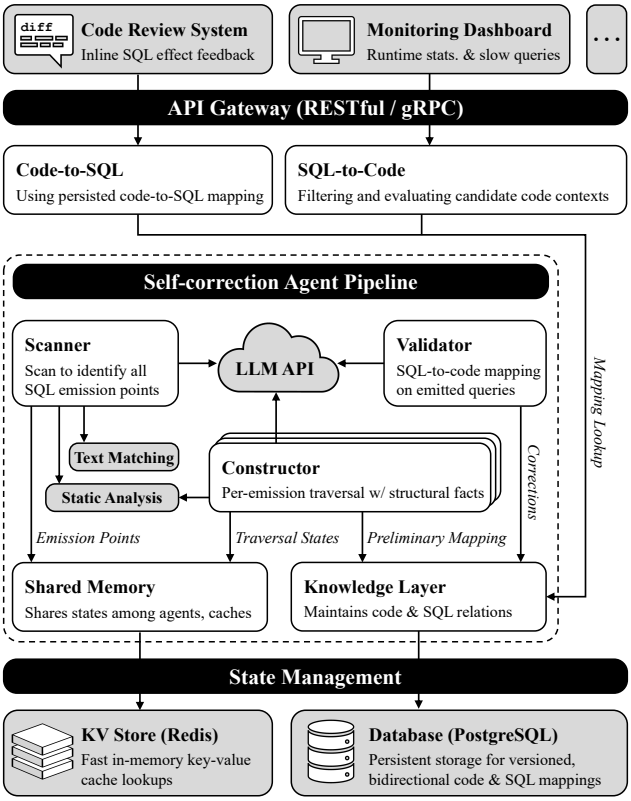


Figure 6: Implementation architecture of SQLens.

the company's internal Git based version control and code review platform through pre commit and pre merge hooks. When a pull request is submitted, a hook inspects modified files and triggers the **SQLens** API to run an incremental scan over the affected regions. Using cached structural facts and historical mappings, the system analyzes only the changed emission points and their dependent call paths. The results appear directly in the pull request interface and show new or modified SQL templates, validation outcomes, and possible performance or security issues. This early feedback helps developers and reviewers understand SQL related changes before merging and keeps database behavior aligned with code evolution.

To remain aligned with evolving database environments, **SQLens** connects to the enterprise schema registry, configuration services, and internal data catalogs. Each inferred query is checked against the current schema to confirm table and column validity, and verified mappings are exported to the catalog for lineage analysis and compliance review. All components run in sandboxed containers that operate on parsed abstract syntax trees rather than live data, which isolates processing from business sensitive content. LLM interactions pass through internal API gateways with rate control and sandboxed serialization to avoid exposure of proprietary code. Repository access is governed by short lived, least privilege credentials issued by the corporate CI environment.

4.3 Agent Orchestration

Each analysis job in **SQLens** is executed by a pool of lightweight agents coordinated by Alibaba's internal distributed task engine,

which functions similarly to Ray style task frameworks deployed on Kubernetes. Every emission site identified during scanning is submitted as an independent analysis task and the scheduler dynamically assigns each task to a worker pod. This distributed execution model provides wide parallelism and preserves responsive incremental updates as repositories evolve.

Agents follow the ReAct execution paradigm described in Section 3.3 and combine iterative reasoning with actionable traversal across the codebase. Each agent runs inside a worker container and maintains partial progress in a shared memory layer backed by Redis, which provides a key value store for inter agent coordination. This shared memory stores contextual artifacts such as partially explored call graphs, intermediate bindings, and inferred code fragments. When multiple agents explore overlapping regions of the same repository, they reuse cached contexts rather than repeating computation. This reuse limits redundant traversal and keeps system behavior stable under frequent updates. For recovery and audit, agents periodically flush context checkpoints to Redis or LMDB so later runs can resume from preserved states rather than restart.

To maintain reliability under large workloads, the task scheduler monitors agent health and applies timeouts and automatic retries for stalled or failed jobs. Each task carries execution limits based on codebase complexity. When a timeout occurs, the agent checkpoints its current reasoning state and is rescheduled without affecting other tasks. Agents are isolated through process level sandboxes to prevent cascading failures across the worker pool. At runtime, the shared memory structure supports efficient reuse of intermediate artifacts. Deduplication of call-graph fragments, adaptive batching of emission sites, and memoization of validated contexts reduce recomputation and analysis overhead, so incremental updates touch only modified regions instead of reprocessing entire repositories.

Overall, the agent orchestration layer provides scalable, fault tolerant, and state aware execution for the full analysis pipeline. It turns full repository reasoning into a continuous, parallel workflow that adapts to rapid development, version evolution, and intensive pull request activity in large industrial environments.

4.4 Validation and Supervision Workflow

The validation and supervision subsystem of **SQLens** ensures that each inferred mapping is technically sound and operationally reliable. It combines multi-stage automatic checks with an interactive review loop so that the system maintains high precision while improving inference stability through accumulated feedback.

SQLens employs two complementary forms of validation. The first is the LLM-based SQL-to-code consistency verification introduced earlier. The second is the automatic validation module detailed here. This module applies three sequential tiers of checks *syntax validation*, *schema validation*, and *security validation*. Syntax validation uses SQL parsers to confirm that generated statements match the grammar of the target database dialect. Schema validation cross references inferred tables, columns, and joins with the enterprise schema registry to detect mismatches or outdated references. Security validation applies heuristics and rule-based detectors to identify unsafe constructs such as string interpolation, wildcard selections, or non-parameterized conditions. Each mapping is annotated with structured validator results and a composite confidence score, which determines its priority in later supervision.

Mappings that fall below a confidence threshold or produce ambiguous validator signals enter the supervision workflow. The supervision layer provides a web interface where reviewers examine the reconstructed SQL template, code provenance, and validator outputs in a single view. Each record moves through a state machine with states *pending*, *reviewed*, *confirmed*, and *merged* so that the system preserves accountability and reproducibility. Reviewers may approve mappings, request correction, or attach reasoning notes, which are stored alongside version metadata in the knowledge layer. Confirmed mappings are reintegrated into the index and become available to downstream CI or runtime consumers.

Over time, reviewer feedback is distilled into project-level signals that refine prompting strategies and validation heuristics. This adaptation mechanism improves inference quality without requiring model retraining or external data exposure.

5 Evaluation

We evaluate **SQLens** along four questions that reflect its core capabilities: (i) how accurately it identifies SQL, (ii) how precisely it localizes SQL to code, (iii) how much backward verification improves mapping quality, and (iv) how well the full workflow performs and scales. These dimensions match the stages in which **SQLens** is used in practice, from recognizing SQL in diverse code paths, to attributing it to source locations, to strengthening mappings with runtime evidence, and finally to sustaining analysis throughput on large codebases. Although **SQLens** is deployed in Alibaba's production environment, all quantitative results in this section are drawn from open-source applications to ensure transparency and reproducibility. We use the same workflow and configuration in both industrial and open-source settings so that results remain comparable. While internal code and data cannot be disclosed, we describe the observed benefits of the production deployment qualitatively, including its effect on CI/CD integration and runtime diagnosis.

5.1 Experimental Setup

5.1.1 Subject Applications. To reflect the range of database-backed services seen in production, we evaluate **SQLens** on ten open-source applications across Java, Ruby, and Python. All rely on relational databases through established ORM frameworks and are actively maintained. The applications cover administrative dashboards, blogs and forums, bookmarking services, content management systems, and e-commerce platforms, providing a broad and diverse basis for evaluation. Their domains, implementation languages, and popularity indicators are summarized in Table 1.

The Java group consists of 8 applications drawn from the benchmark suite used in the DBridge study [45]. They rely on common stacks such as Spring JDBC, JPA, and Hibernate, and their codebases reach up to 90 k lines with non-trivial schemas. Their test suites are modest, exercising about 39 distinct SQL statements on average, which we account for when interpreting coverage.

To broaden the evaluation beyond Java, we include one Ruby on Rails and one Python/Django application. Redmine represents a mature Rails application with an active test suite and well-documented setup. Saleor represents a widely deployed Django-based e-commerce service with substantial database activity and strong test coverage. These two applications complement the Java group by exercising ORM-heavy frameworks with different query construction patterns.

Table 1: Subject Applications Used in the Evaluation

Application	Domain	Language	GitHub Stars
eladmin	Admin Dashboard	Java	21.8 k
mblog	Blog	Java	0.6 k
SpringBlog	Blog	Java	1.6 k
JavaQuarkBBS	Forum	Java	0.9 k
favorites-web	Bookmarking	Java	4.9 k
OnlineMall	E-commerce	Java	0.4 k
wallride	CMS	Java	0.1 k
bbs-pro	Forum	Java	–
Redmine	Issue Tracker	Ruby	5.8 k
Saleor	E-commerce	Python	22.2 k

5.1.2 Data Preparation. Ground-truth SQL statements and their originating code locations are obtained by running each application's test suite with SQL logging enabled. For the Java applications, we reuse the JDBC-level logs available in the DBridge benchmark. For the Ruby and Python applications, we check out fixed commits, execute their official test suites, and instrument their database access layers to record all executed SQL along with call stacks. All collected SQL is normalized through case folding, canonical whitespace, and parameter abstraction to produce a stable set of unique queries. This follows established practice [45, 46] and allows direct comparison between reconstructed and observed SQL.

5.1.3 Implementation and Environment. **SQLens** is implemented mainly in Python, which hosts the analysis and orchestration logic. A small amount of middleware in other languages connects the system to enterprise development, deployment, and monitoring environments. For all experiments, we run **SQLens** as a containerized service on Alibaba Cloud using ECS instances with 16 vCPUs and 64 GB of RAM, and we enable parallel execution across cores unless noted otherwise. **SQLens** interacts with the Qwen3-Max model through Alibaba's Bailian platform API, using temperature 0 and per-prompt response caching to reduce run-to-run variance.

5.2 SQL Identification Effectiveness

We first evaluate how accurately **SQLens** identifies SQL statements embedded in application code. This capability is fundamental: if the system fails to recover a large fraction of runtime queries, downstream mapping, localization, and governance cannot be complete. For each application, Table 2 reports the total number of unique dynamically executed SQL statements (#Dyn SQL) and three metrics: Ident. (the number of distinct SQL strings that a tool infers statically after normalization), Match. (how many of those inferred strings match a normalized dynamic query), and Cov. (coverage, defined as Match./#Dyn SQL). Coverage serves as a recall-oriented indicator, while the ratio Match./Ident. shows how many inferred candidates correspond to queries that are actually executed. On the Java benchmarks, we compare against DBridge [45], a state-of-the-art pointer-analysis–based static analyzer for Java-to-database value flows; for the Ruby and Python applications, no comparable static baseline exists to the best of our knowledge.

Across the Java applications, **SQLens** attains coverage that is broadly comparable to DBridge while requiring substantially less framework-specific engineering. On several systems with conventional ORM stacks, **SQLens** and DBridge achieve similar coverage,

Table 2: SQL Identification Results

Application	# Dyn SQL	SQLens			DBridge		
		Ident.	Match.	Cov.	Ident.	Match.	Cov.
eladmin	50	111	30	60%	83	37	74%
mblog	36	131	31	86%	60	26	72%
SpringBlog	20	40	16	80%	22	19	95%
JavaQuarkBBS	34	86	26	76%	42	22	65%
favorites-web	29	103	25	86%	50	22	76%
OnlineMall	25	56	23	92%	31	25	100%
wallride	51	226	20	39%	83	21	41%
bbs-pro	68	152	59	87%	402	55	81%
Redmine	904	2888	546	60%	–	–	–
Saleor	481	1955	138	29%	–	–	–

and in some cases **SQLens** recovers more dynamic queries by exploring a wider set of code paths and configuration branches (for example, on `JavaQuarkBBS` and `favorites-web`). In a few applications with heavily customized templates and schema-specific conventions, DBridge retains an advantage, reflecting its finely tuned value-flow rules within the Java/JDBC ecosystem. Overall, the per-application trends suggest a complementary picture: DBridge remains a strong baseline within its native domain, while **SQLens** closes much of the gap through a model-driven pipeline that generalizes across Java frameworks, without per-project human efforts.

Beyond Java, **SQLens** applies the same emission-site–anchored workflow to Ruby on Rails and Python/Django stacks. On the mature Rails application `Redmine`, it identifies a broad set of static SQL candidates and covers most runtime queries despite deep ORM layers and a large test suite. On the Django-based `Saleor`, the measured coverage indicates that many meta-programmed and dynamically composed queries are not fully exercised by the official tests. This evaluation style—deriving ground truth from test-suite execution to obtain real call stacks and emitted SQL—is widely used in academic studies, but it differs from the richer and more varied execution paths present in industrial production systems. Even so, these applications fall outside the reach of Java-focused analyzers such as DBridge, yet **SQLens** still reconstructs a substantial portion of their dynamic SQL, showing that a single model-driven design can offer consistent visibility across heterogeneous language stacks with only minimal adaptation.

To understand the remaining gaps, we manually inspect representative identification failures. For example, in a typical Java ORM case the ground-truth dynamic query uses a wildcard projection (`SELECT r.*`) while **SQLens** infers an equivalent template that explicitly enumerates all projected columns over the same join and predicate structure. ORM frameworks (for example, Spring Data JPA and Hibernate integrations) often choose between wildcard and explicit projections based on entity-loading semantics, projection hints, or mapper metadata. Absent a strict constraint, our LLM-guided inference tends to expand wildcards into column lists using the available schema information. From the database perspective, the two queries are semantically equivalent; however, our evaluation adopts a conservative syntactic matching rule that treats them as different and counts this case as a missed match.

This analysis indicates that part of the apparent coverage gap in Table 2 arises from strict normalization and matching criteria rather

than from an inability to reconstruct the underlying SQL intent. Relaxing projection-form equivalence or enriching the matcher with ORM-specific heuristics would turn many such cases into successful matches, increasing measured coverage without changing the inference pipeline itself. At the same time, the current conservative policy makes the reported coverage a lower bound on **SQLens**'s true reconstruction ability, which is desirable for downstream governance. Taken together, the results show that **SQLens** already identifies a large and useful fraction of runtime SQL across diverse languages and frameworks, while leaving clear headroom for further improvements through more semantics-aware matching.

5.3 SQL-to-code Localization Accuracy

This section evaluates how effectively **SQLens** localizes runtime SQL statements to their originating source locations. We treat *emission sites* as application-level points that issue database queries (for example, lines invoking ORM APIs) and, using the ground-truth mapping described above, measure Top@K localization accuracy per application. A query is counted as correctly localized at Top@K if its true emission site appears within the top-K ranked candidates in Table 3; Top@1 corresponds to direct pinpointing, while Top@3 and Top@5 measure how often the correct site appears in a short candidate list. To our knowledge, there is no cross-language baseline that targets SQL-to-code localization at this granularity, so the analysis focuses on the absolute performance of **SQLens** and relates it qualitatively to static, rule-based analyzers.

On the Java benchmark suite, **SQLens** localizes most queries with strong Top@K accuracy. Top@1 usually falls between 60–80%, while Top@3 and Top@5 often approach full coverage, with some applications (for example, `favorites-web`) reaching 100% at Top@5. In practice, this means that most dynamically observed queries map either directly to their true call site or appear within a short, easy-to-inspect candidate list, which is adequate for slow-query diagnosis and regression triage in large codebases.

The Ruby on Rails and Django applications exhibit more challenging behavior due to their dynamic dispatch and meta-programming features. In `Redmine` and `Saleor`, Top@1 accuracy is lower than in Java but still provides useful guidance, and Top@5 brings a large fraction of queries within the top five candidates. Manual inspection indicates that many remaining cases involve framework-internal helpers or highly dynamic query paths that are also difficult for traditional static analyses without additional, framework-specific modeling. Overall, the results indicate that **SQLens** achieves near-complete localization with compact candidate sets on statically typed Java stacks and still narrows the search space substantially on ORM-heavy Ruby/Python projects, providing a language-agnostic SQL-to-code mapping layer that complements specialized value-flow analyzers as heterogeneous codebases evolve.

5.4 Effectiveness of Backward Validation

We assess how backward validation strengthens SQL inference and how far mappings learned from a small execution slice can generalize. This experiment reflects the incremental setting in which **SQLens** maintains a versioned knowledge layer that is gradually refined by new executions. We focus on two dynamic-language applications, `Redmine` (Rails) and `Saleor` (Django). For each, we partition the official tests into a 20% *train* slice and an 80% *test* slice, populate the knowledge layer using the train slice, freeze it, and then evaluate on the held-out tests without further logging. Coverage (Match./#SQL after normalization) is reported under a *zero-shot* baseline and a *few-shot* setting that reuses mappings learned from the 20% slice. As shown in Table 4, even a small training budget leads to consistent improvements. On `Redmine`, coverage increases from 53.65% to 60.40%, and on `Saleor` from 25.16% to 28.69%. The gains arise from mappings that generalize beyond the specific training queries and capture recurring ORM patterns, including scaffolded CRUD paths and framework-level callbacks that reappear across the codebase. `Saleor` shows a smaller increase because many of its meta-programmed and dynamically composed queries are exercised only lightly by the available tests, yet the learned mappings still recover additional unseen queries.

Together with the per-application results in Table 2, these findings show that modest execution data is sufficient to seed and incrementally enhance a reusable knowledge layer. In practice, **SQLens** does not run in a zero-shot regime: industrial deployments accumulate mappings over time, and backward validation offers a low-overhead mechanism to keep the knowledge layer current and improve inference quality as workloads evolve.

Table 3: Code-to-SQL Localization Results

Application	Emission Sites	Top@1	Top@3	Top@5
eladmin	35	67%	81%	87%
mblog	31	73%	88%	88%
SpringBlog	22	70%	85%	90%
JavaQuarkBBS	25	66%	87%	89%
favorites-web	23	88%	98%	100%
OnlineMall	23	83%	96%	96%
wallride	37	63%	77%	81%
bbs-pro	59	78%	83%	96%
Redmine	933	41%	53%	60%
Saleor	414	51%	58%	66%

Table 4: Coverage Improvement from Backward Validation

Application	#SQL	Zero-shot		Few-shot	
		Match.	Cov.	Match.	Cov.
Redmine	904	485	53.65%	546	60.40%
Saleor	481	121	25.16%	138	28.69%

5.5 Performance and Scalability of the Analysis

We evaluate whether **SQLens** is efficient and scalable enough for continuous integration and production workflows. The end-to-end pipeline has three stages aligned with Section 3.3: a *scanner* that parses projects and indexes call graphs and ORM metadata, a *constructor* that performs LLM-guided path traversal and SQL inference per emission site, and a *validator* that refines inferred queries using observed executions when available. Across all 10 applications, the scanner completes in under 10 minutes per project; the constructor dominates cost, requiring 2–5 minutes of time and about 0.1–0.9M tokens per emission site (approximately 0.38M on average); and the validator adds less than 1 minute and around 10K tokens per site. Because emission sites are processed in parallel, end-to-end wall-clock time remains within 5–20 minutes per project, grows

roughly linearly with code size and the number of emission sites, and is comparable on Java benchmarks to prior static tools such as DBridge, which reports around 3.5 minutes per project.

In Alibaba's deployment, these properties allow **SQLens** to run in CI/CD pipelines and periodic governance jobs. A full SQL-visibility pass over a mid-sized service fits within typical pipeline budgets, incremental runs further reduce steady-state overhead, and the resulting code-to-SQL mappings make it easier to trace slow queries and missing indexes back to responsible code paths, shortening SQL-related incident triage without additional instrumentation.

6 Related Work

Static Program Analysis. General-purpose static analysis frameworks such as Soot provide established methods for call-graph and data-flow analysis over Java bytecode and IR [66]. In industry, Semmle/QL introduces a rule-based querying model [67], Facebook's Continuous Reasoning brings formal methods into CI/CD with tools such as RacerD [8, 53], and Tricorder integrates analyzers into code review workflows [58]. These systems work well for static properties but generally do not recover dynamic SQL constructed through templates or string assembly, particularly in heterogeneous ORM environments. Our work complements these approaches by combining static program facts with multi-step ReAct-style planning and backward validation to produce verifiable and auditable code-to-SQL mappings that integrate with CI and review pipelines.

Code-to-SQL Mapping and Localization. Prior work uses static and hybrid analysis to recover SQL behavior from code for checking, optimization, and security [18, 21, 45, 46, 49–51, 70]. These techniques are code-aware but largely single-language, ORM-limited, and lack a versioned, bidirectional mapping for CI integration or runtime reverse lookup. In contrast, **SQLens** performs bottom-up reconstruction from emission sites and applies backward validation over observed SQL, yielding a versioned, auditable index that supports forward and reverse queries across heterogeneous stacks.

Static Analysis for Multilingual Codebases. Cross-language analyses construct unified views over multilingual systems via language-specific frontends, shared representations, and language-agnostic runtimes [4, 28, 38, 41, 44, 57]. In contrast to per-language analyzers, **SQLens** leverages LLM generalization to align heterogeneous code and ORM frameworks into a shared code-to-SQL reconstruction for polyglot services.

Runtime Visibility and Distributed Tracing. Distributed tracing provides end-to-end timing and causality but usually remains at service- or call-stack granularity. Systems such as Pivot Tracing [48], X-Trace [31], Hindsight [75], and Snicket [7] extend runtime visibility yet do not link traces to fine-grained SQL behavior. We instead align runtime SQL, including slow queries, with a versioned code-to-SQL index to enable reverse lookup from anomalies to concrete emission sites and parameter flows.

Data Provenance and Lineage. Data provenance explains how query results are derived, from semiring-based models [33] to systems such as Titian that record lineage in Spark [39]. These works ask "where results come from" rather than "how application code produces the SQL", while our code-to-SQL knowledge graph links queries to their emission sites and contexts. Buneman and Tan review deployment issues and open challenges [9].

SQL Injection Defenses and Templated SQL. SQL injection has been widely studied. AMNESIA combines static analysis with runtime monitoring for templated SQL [34], and surveys catalogue attack taxonomies and defenses [35]. Static analyses track untrusted inputs [71], while dynamic approaches infer intended query structure at runtime [6]. **SQLens** instead anchors on SQL emission sites to reconstruct dynamic templates and parameter flows across frameworks, unifying security checks with index and plan suggestions in a single auditable pipeline for both pre-PR and runtime use.

Workload-Driven Database Tuning. Workload-driven tuning automates DBMS configuration from runtime signals, as in OtterTune [1] and its cloud extension [74], in self-driving DBMS architectures [54], and in cloud databases [15, 16]. These systems tune the engine rather than trace how application code issues SQL or block regressions before merge. **SQLens** instead embeds governance in CI and links runtime SQL anomalies back to emission sites, while real-time SQL templating and sequence mining [68] is complementary and can supply template-level regression signals.

LLM-Assisted Code Reasoning and Text-to-SQL. Learning-based code models enable cross-language transfer [30, 69], and surveys summarize opportunities and challenges for Big Code [2], but their predictions lack auditability and reproducibility for governance. **SQLens** combines LLM generalization with static program facts, multi-agent ReAct planning, and automated or human validation to yield versioned, interpretable code-to-SQL mappings across frameworks. Text-to-SQL advances via prompt and workflow design [32, 61, 76], and LLMs are applied to tasks such as automated repair [62] and DB bug reproduction from community reports [77], underscoring their complement to static analyzers.

Prior work advances static analysis, tracing, provenance, security, and learned code understanding, but typically does not provide a maintainable, versioned, and auditable mapping from runtime SQL back to its generating code across polyglot stacks. **SQLens** fills this gap by coupling emission-site–anchored reconstruction with backward validation to build a bidirectional, versioned index that supports CI-time checks, runtime reverse lookup, and integration with tracing, provenance, and tuning systems.

7 Conclusion

SQLens provides an auditing-grade visibility layer that links application code to its generated SQL in polyglot, fast-evolving systems. Unlike rule-based static analyzers and runtime-only tracers, it uses a closed-loop, self-correcting workflow that combines LLM-guided ReAct-style reasoning with language-aware program facts to reconstruct normalized SQL templates and bindings at execution endpoints. **SQLens** maintains these mappings in a versioned, queryable index that supports bidirectional lookup between code and SQL across continuous integration and production workflows. Supervised validation and provenance tracking keep inferences auditable and reproducible, while a lightweight integration interface brings SQL-aware checks to pull requests and ties operational anomalies to responsible code owners. Deployed at Alibaba Group and evaluated on diverse open-source services, **SQLens** reduces diagnosis latency and improves the accuracy of code–query attribution with low workflow overhead, providing a practical basis for large-scale SQL governance and secure database operations.

References

[1] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 1009–1024.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4 (2018), 81:1–81:37.

[3] Maxime André, Marco Raglianti, Anthony Cleve, and Michele Lanza. 2025. Understanding Data Access in Microservices Applications Using Interactive Treemaps. In *33rd IEEE/ACM International Conference on Program Comprehension, ICPC@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025*. IEEE, 216–220.

[4] Steven Arzt, Tobias Kussmaul, and Eric Bodden. 2016. Towards cross-platform cross-language analysis with soot. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2016, Santa Barbara, CA, USA, June 14, 2016*. ACM, 1–6.

[5] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. USENIX Association, 307–320.

[6] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. 2007. CANDID: preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. ACM, 12–24.

[7] Jessica Berg, Fabian Ruffy, Khanh Nguyen, Nicholas Yang, Taegyun Kim, Anirudh Sivaraman, Ravi Netravali, and Srinivas Narayana. 2021. Snicket: Query-Driven Distributed Tracing. In *HotNets '21: The 20th ACM Workshop on Hot Topics in Networks, Virtual Event, United Kingdom, November 10-12, 2021*. ACM, 206–212.

[8] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: compositional static race detection. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 144:1–144:28.

[9] Peter Buneman and Wang-Chiew Tan. 2018. Data Provenance: What next? *SIGMOD Rec.* 47, 3 (2018), 5–16.

[10] Shaosheng Cao, Xinxing Yang, Cen Chen, Jun Zhou, Xiaolong Li, and Yuan Qi. 2019. TitAnt: Online Real-time Transaction Fraud Detection in Ant Financial. *Proc. VLDB Endow.* 12, 12 (2019), 2082–2093.

[11] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. 2024. Toward effective secure code reviews: an empirical study of security-related coding weaknesses. *Empir. Softw. Eng.* 29, 4 (2024), 88.

[12] Surajit Chaudhuri and Vivek R. Narasayya. 1998. AutoAdmin 'What-if' Index Analysis Utility. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*. ACM Press, 367–378.

[13] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. ACM, 1001–1012.

[14] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2016. Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks. *IEEE Trans. Software Eng.* 42, 12 (2016), 1148–1161.

[15] Zongzhi Chen, Mo Sha, Feifei Li, Sheng Wang, Baolin Huang, Guoqing Ma, Huaxiong Song, Ke Yu, Xizhe Zhang, and Yuan Wang. 2026. CloudJump III: Optimizing Cloud Databases for Tiered Storage. In *Companion of the 2026 International Conference on Management of Data, SIGMOD/PODS 2026, Bengaluru, India, May 31–June 5, 2026*. ACM.

[16] Zongzhi Chen, Xinjun Yang, Mo Sha, Feifei Li, Kang Wang, Zheyu Miao, Jie Xu, Jianfeng Wang, and Sheng Wang. 2025. CloudJump II: Optimizing Cloud Databases for Shared Storage. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*. ACM, 336–349.

[17] James Cheney, Laura Chiticariu, and Wang Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Found. Trends Databases* 1, 4 (2009), 379–474.

[18] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM, 3–14.

[19] Mike Chow, Yang Wang, William Wang, Ayichew Hailu, Rohan Bopardikar, Bin Zhang, Jialiang Qu, David Meisner, Santosh Sonawane, Yunqi Zhang, Rodrigo Paim, Mack Ward, Ivor Huang, Matt McNally, Daniel Hodges, Zoltan Farkas, Caner Gocmen, Elvis Huang, and Chunqiang Tang. 2024. ServiceLab: Preventing Tiny Performance Regressions at Hyperscale through Pre-Production Testing. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 545–562.

[20] Johannes Dahse and Thorsten Holz. 2014. Static Detection of Second-Order Vulnerabilities in Web Applications. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. USENIX Association, 989–1003.

[21] Arjun Dasgupta, Vivek R. Narasayya, and Manoj Syamala. 2009. A Static Analysis Framework for Database Applications. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*. IEEE Computer Society, 1403–1414.

[22] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. 2007. Keynote Address: .QL for Source Code Analysis. In *Seventh IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2007), September 30 - October 1, 2007, Paris, France*. IEEE Computer Society, 3–16.

[23] Torgeir Dingsøyr, Nils Brede Moe, Tor Erlend Fægri, and Eva Amdahl Seim. 2018. Exploring software development at the very large-scale: a revelatory case study and research agenda for agile method adaptation. *Empir. Softw. Eng.* 23, 1 (2018), 490–520.

[24] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70.

[25] Henrietta Dombrovskaya and Richard Lee. 2014. Talking to the Database in a Semantically Rich Way. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*. OpenProceedings.org, 676–687.

[26] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proc. VLDB Endow.* 2, 1 (2009), 1246–1257.

[27] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David A. Wagner. 2013. An Empirical Study on the Effectiveness of Security Code Review. In *Engineering Secure Software and Systems - 5th International Symposium, ESSoS 2013, Paris, France, February 27 - March 1, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7781)*. Springer, 197–212.

[28] Sedick David Baker Effendi, Xavier Pinho, Andrei Michael Dreyer, and Fabian Yamaguchi. 2025. Scalable Language Agnostic Taint Tracking using Explicit Data Dependencies. In *Proceedings of the 14th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2025, Seoul, Republic of Korea, 16 June 2025*. ACM, 36–42.

[29] K. Venkatesh Emani, Tejas Deshpande, Karthik Ramachandra, and S. Sudarshan. 2017. DBridge: Translating Imperative Code to SQL. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 1663–1666.

[30] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*. Association for Computational Linguistics, 1536–1547.

[31] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-Trace: A Pervasive Network Tracing Framework. In *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings*. USENIX.

[32] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proc. VLDB Endow.* 17, 5 (2024), 1132–1145.

[33] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*. ACM, 31–40.

[34] William G. J. Halfond and Alessandro Orso. 2005. AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*. ACM, 174–183.

[35] William G. J. Halfond, Jeremy Viegas, and Alessandro Orso. 2006. In *2006 IEEE International Symposium on Secure Software Engineering, ISSSE 2006, Arlington, VA, USA, March 16 -17, 2006*.

[36] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. ACM, 981–992.

[37] Gansen Hu, Zhaoguo Wang, Chuzhe Tang, Jiahuan Shen, Zhiyuan Dong, Sheng Yao, and Haibo Chen. 2024. WeBridge: Synthesizing Stored Procedures for Large-Scale Real-World Web Applications. *Proc. ACM Manag. Data* 2, 1 (2024), 64:1–64:29.

[38] Mingzhe Hu, Qi Zhao, Yu Zhang, and Yan Xiong. 2023. Cross-Language Call Graph Construction Supporting Different Host Languages. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023*. IEEE, 155–166.

[39] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd D. Millstein, and Tyson Condie. 2015. Titian: Data Provenance Support in Spark. *Proc. VLDB Endow.* 9, 3 (2015), 216–227.

[40] Jonathan Kaldor, Jonathan Mace, Michal Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An

End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 34–50.

[41] Jacob Kreindl, Daniele Bonetta, and Hanspeter Mössenböck. 2019. Towards efficient, multi-language dynamic taint analysis. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2019, Athens, Greece, October 21-22, 2019*. ACM, 85–94.

[42] Rodrigo N. Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. 2021. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. *Proc. VLDB Endow.* 14, 13 (2021), 3348–3361.

[43] Feifei Li. 2019. Cloud native database systems at Alibaba: Opportunities and Challenges. *Proc. VLDB Endow.* 12, 12 (2019), 2263–2272.

[44] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. 2022. PolyCruise: A Cross-Language Dynamic Information Flow Analysis. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. USENIX Association, 2513–2530.

[45] 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.

[46] 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.

[47] Minghua Ma, Zheng Yin, Shenglin Zhang, Sheng Wang, Christopher Zheng, Xinhao Jiang, Hanwen Hu, Cheng Luo, Yilin Li, Nengjun Qiu, Feifei Li, Changcheng Chen, and Dan Pei. 2020. Diagnosing Root Causes of Intermittent Slow Queries in Large-Scale Cloud Databases. *Proc. VLDB Endow.* 13, 8 (2020), 1176–1189.

[48] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot tracing: dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. ACM, 378–393.

[49] Dimitris Mitropoulos and Diomidis Spinellis. 2009. SDriver: Location-specific signatures prevent SQL injection attacks. *Comput. Secur.* 28, 3-4 (2009), 121–129.

[50] Csaba Nagy and Anthony Cleve. 2017. A Static Code Smell Detector for SQL Queries Embedded in Java Code. In *17th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2017, Shanghai, China, September 17-18, 2017*. IEEE Computer Society, 147–152.

[51] Csaba Nagy, Loup Meurice, and Anthony Cleve. 2015. Where was this SQL query executed? a static concept location approach. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*. IEEE Computer Society, 580–584.

[52] Thanh H. D. Nguyen, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2014. An industrial case study of automatically identifying performance regression-causes. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. ACM, 232–241.

[53] Peter W. O'Hearn. 2018. Continuous Reasoning: Scaling the impact of formal methods. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. ACM, 13–25.

[54] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org.

[55] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-grained Lineage at Interactive Speed. *Proc. VLDB Endow.* 11, 6 (2018), 719–732.

[56] Karthik Ramachandra, Ravindra Guravannavar, and S. Sudarshan. 2012. Program analysis and transformation for holistic optimization of database applications. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14, 2012*. ACM, 39–44.

[57] Tobias Roth, Julius Näumann, Dominik Helm, Sven Keidel, and Mira Mezini. 2024. AXA: Cross-Language Analysis through Integration of Single-Language Analyses. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*. ACM, 1195–1205.

[58] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma S"oderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, 598–608.

[59] Mojtaba Shahin, Mansooreh Zahedi, Muhammad Ali Babar, and Liming Zhu. 2019. An empirical study of architecting for continuous delivery and deployment. *Empir. Softw. Eng.* 24, 3 (2019), 1061–1108.

[60] Shudi Shao, Zhengyi Qiu, Xiao Yu, Wei Yang, Guoliang Jin, Tao Xie, and Xintao Wu. 2020. Database-Access Performance Antipatterns in Database-Backed Web Applications. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 58–69.

[61] Liang Shi, Zhengju Tang, Nan Zhang, Xiaotong Zhang, and Zhi Yang. 2025. A Survey on Employing Large Language Models for Text-to-SQL Tasks. *ACM Comput. Surv.* 58, 2, Article 54 (Sept. 2025), 37 pages.

[62] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. In *IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2023, Melbourne, Australia, May 16, 2023*. IEEE, 23–30.

[63] Thodoris Sotiropoulos, Stefanos Chaliasos, Vaggelis Atlidakis, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Data-Oriented Differential Testing of Object-Relational Mapping Systems. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1535–1547.

[64] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2022. Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 4–18.

[65] Patanamon Thongtanunam, Shane McIntosh, Ahmed E. Hassan, and Hajimu Iida. 2015. Investigating Code Review Practices in Defective Files: An Empirical Study of the Qt System. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*. IEEE Computer Society, 168–179.

[66] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*. IBM, 13.

[67] Mathieu Verbaere, Elnar Hajiyev, and Oege de Moor. 2007. Improve software quality with SemmleCode: an eclipse plugin for semantic code search. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. ACM, 880–881.

[68] Jiaqi Wang, Tianyi Li, Anni Wang, Xiaoze Liu, Lu Chen, Jie Chen, Jianye Liu, Junyang Wu, Feifei Li, and Yunjun Gao. 2023. Real-time Workload Pattern Analysis for Large-scale Cloud Databases. *Proc. VLDB Endow.* 16, 12 (2023), 3689–3701.

[69] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Association for Computational Linguistics, 8696–8708.

[70] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar T. Devanbu. 2007. Static checking of dynamically generated queries in database applications. *ACM Trans. Softw. Eng. Methodol.* 16, 4 (2007), 14.

[71] Yichen Xie and Alex Aiken. 2006. In *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*. USENIX Association.

[72] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*. ACM, 1299–1308.

[73] Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. 2018. How *not* to structure your database-backed web applications: a study of performance bugs in the wild. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 800–810.

[74] Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J. Gordon. 2018. A Demonstration of the OtterTune Automatic Database Management System Tuning Service. *Proc. VLDB Endow.* 11, 12 (2018), 1910–1913.

[75] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. 2023. The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*. USENIX Association, 321–339.

[76] Xinyi Zhang, Tiantian Chen, Zhentao Han, Zhaoyan Hong, Wei Lu, Sheng Wang, Mo Sha, Anni Wang, Shuang Liu, Yakun Zhang, Feifei Li, and Xiaoyong Du. 2026. Why Database Manuals Are Not Enough: Efficient and Reliable Configuration Tuning for DBMSs via Code-Driven LLM Agents. *Proc. VLDB Endow.* 19, 6 (2026), 1358–1371.

[77] Suyang Zhong, Mo Sha, Sheng Wang, Fangyuan Zhou, Feifei Li, and Kian-Lee Tan. 2026. DBugScribe: Automatic Database Bug Reproduction from Community Reports. In *Companion of the 2026 International Conference on Management of Data, SIGMOD/PODS 2026, Bengaluru, India, May 31–June 5, 2026*. ACM.

[78] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. Morgan Kaufmann, 1087–1097.