International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(22s) 2026 pp. 1290-1298
https://doi.org/10.70917/ijcisim-2026-5520
Article

AI-Enabled SQL Review in the Software Development Life Cycle: A Framework for Shift-Left Database Quality Assurance

Maneesh Singh

Hexaware Technologies, USA
ORCID: 0009-0003-5956-8880

Abstract: SQL defects such as injection vulnerabilities, anti-patterns such as missing index definitions, and destructive database schema migrations remain one of the most common and expensive defect categories. Despite the increasing use of continuous integration tooling and DevOps-style governance, such defects are commonly detected late in the delivery cycle, at the staging or production phase, at which stage remediation costs are an order of magnitude higher than when they were introduced. We propose and evaluate a 5-gate, AI-assisted SQL review framework to be applied across the entire SDLC, from developer IDEs to pull requests and CI/CD pipelines, and to staging and production systems. Each of the five gates of the framework is augmented by LLM-based components to enable contextualized query evaluation, SQL execution plan interpretation, and migration risk classification. Alongside the technical architecture is a layer of governance components that include human override, data privacy and rule ownership. Benefits from pilot sites include a 60–70% reduction in SQL incidents in production over a six-month period, a 70% reduction in the mean time to detection (MTTD) of SQL incidents, and the near-elimination of SQL injection findings at the PR merge gate. The framework shows how left-shifting SQL quality assurance, or converting incident-based reactive quality assurance into proactive automated review, reduces delivery risk, engineering rework, and cloud compute spending while preserving developer autonomy through simple rule governance.
Keywords: SQL quality assurance; shift-left testing; large language models; static analysis; DevSecOps; CI/CD pipeline; database governance; software development life cycle

1. Introduction

Enterprise applications are data-centric, often relying on relational databases as the source of truth for financial transactions, customer data, and business states. The most important artifacts of a software delivery pipeline are the database queries and the schema themselves. SQL statements with bugs (e.g. security vulnerabilities, performance anti-patterns, and unsafe migrations) have a large blast radius, they are difficult to roll back, and they have a high mean time to recovery (MTTR) [1]. SQL artifacts historically have not benefited from as much automated quality checking as application code, and this asymmetry is reflected in a disproportionate number of production incident reports from SQL statements.

When applied to DevOps practice, the shift-left testing principle can be considered an adaptation of the "cost of quality" principle as originally described by Boehm and Basili [2]: namely, defects found earlier in the software development lifecycle are cheaper to fix. Shift-left has been applied to unit testing, static analysis of application-level code, and dependency vulnerability scanning, but not to SQL-specific quality gates. Manual review of SQL code by a database administrator (DBA), the most common mechanism of SQL quality assurance, does not scale to the velocity of modern continuous delivery pipelines and is subject to reviewer variability due to the DBAs' experience [3].

With LLMs' recent ability to syntactically and semantically understand SQL, a potential for automated, contextualized SQL QA emerges throughout the SDLC. LLMs used in GitHub Copilot, CodeRabbit or directly using OpenAI's GPT-4 through the OpenAI API and Anthropic's Claude have suggested SQL query optimizations, detected injection patterns, and evaluated migration impacts [4]. This gate-based integration of LLM-assisted SQL review into the SDLC from an ad hoc developer tool considerably improves quality and security consistency across all SQL code development activities.

This paper proposes the following: (i) a five-gate AI-enabled SQL review framework across the SDLC; (ii) a governance framework considering human override, data privacy considerations, and ruleset ownership; (iii) an implementation roadmap, including success criteria for each phase; and (iv) performance targets based on the pilot deployments. The remainder of this paper is structured as follows. Section 2 is a review of the state of SQL quality assurance and some systemic weaknesses. Section 3 describes a five-gate technical framework. On governance, look at Section 4; for the implementation roadmap, look at Section 5; and for the performance evaluation, look at Section 6. Threats to validity are discussed in section 7. Section 8 summarizes implications for research and practice.

2. Background and Related Work

2.1 Costs of SQL defects in production environments

The costs of SQL defects can vary widely according to a taxonomy of SQL defects. Missing an index definition may require a full table scan, thus reducing the throughput of a given query by multiple orders of magnitude against expected production volumes, cascading failures to downstream services [5]. The concept of the N+1 query anti-pattern describes the situation in which a parent query results in N queries to retrieve child rows from the back-end for each row returned by the parent query. This scenario is acceptable as a development error but disastrous at scale. SQL injection is the most well-known OWASP Top 10 security vulnerability, remaining on the list for over 20 years [6]. Using unsanitized input, an attacker seeks to manipulate the state of the database inside a DBMS. Destructive data definition language (DDL) migrations such as dropping a column or changing the type of the column without backfilling the values or dropping an index can affect integrity and require point-in-time restores.

The cost of fixing defects in later stages of the software lifecycle is well understood. According to Capers Jones, fixing defects in production might be from 30 to 100 times more costly than at the unit test stage [7]. Additionally, SQL-specific defects are especially affected by the operational complexity of production DBMS clusters; for example, rolling back a destructive migration on a production DBMS cluster often involves coordinated snapshots, replica resynchronisation and application-layer compatibility checks. The cost to the organization of a single P1 SQL database incident in terms of engineering time, infrastructure management and client risk has been conservatively estimated at $10000 to $50000 USD per incident [3].

2.2 Manual DBA Review Limitations

In customary SQL QA, DBAs review SQL at the pull request stage, usually by a limited number of DBA specialists, which has weaknesses in large-scale production SQL development. Throughput is limited: a DBA reviewing longer execution plans each day will have a limited number of pull requests they can accept, detrimentally affecting delivery velocity, or they will bypass the review entirely. Review quality is variable: a junior DBA or one unfamiliar with the schema surrounding a service may miss subtle anti-patterns within an execution plan. Third, a DBA review is essentially a reactive measure because it is performed after the code has been written and reviewed by application engineers, and the cost of any refactoring has already been incurred [8].

While static analysis tools like SQLFluff [9] can perform syntactic normalization and analysis of dialect-specific details, they lack the understanding to determine whether a query would be performant on a given database schema, if a migration is safe to roll back, or if a parameterized pattern sufficiently guards against injection in the context of a surrounding application. This is the space that LLM-based tools occupy.

2.3 LLMs in Code Review Contexts

Empirical studies have also provided evidence of the performance gains from code reviews assisted by LLMs. For instance, Sobania et al. [10] found that GPT-4 was able to automatically correct 31 out of 40 (i.e., 77.5%) simple bugs from a standard code repair benchmark. In a systematic mapping study about the use of LLMs in software engineering, Fan et al. [11] found code review, test generation, and code refactoring to be the most well-evidenced. While evaluations of SQL-focused LLMs are less common in the academic literature, vendor evaluations and evaluations by practitioners indicate state-of-the-art accuracy in the standard injection detection and index recommendation tasks for SQL [4]. The framework we propose builds off of these works, enacting LLM reasoning within a structured, gated SDLC process while ensuring that human oversight and governance accountability remain intact.

2.4 Gaps in Existing Approaches

Existing database change management systems such as Liquibase Pro apply deterministic rule simulation to DDL migrations but do not perform DQL or DML analysis, do not use language model scoring, and have no mechanism to adapt from production outcomes. Static analysis tools including SonarQube and Checkmarx apply rule-based pattern matching to SQL but provide no schema-context-aware probabilistic scoring. Commercial AI code review tools including CodeRabbit apply general LLM analysis to code diffs but lack the governed schema registry, multi-dimensional weighted scoring, and production incident feedback calibration described in this framework. To the authors' knowledge, this is the first framework to combine governed schema context injection with LLM-based probabilistic risk scoring and a formally-specified production feedback calibration mechanism.

3. The Five-Gate AI-Enabled SQL Review Framework

We suggest a framework of AI-based SQL analyzes at five incremental checkpoints in the SDLC, in which each type of analysis is layered onto the previous to detect a class of defects or quality dimensions expected to be most amenable to that stage's available artifacts and environment. Gates should take little weight relative to the defects being prevented in order to maintain developer flow and prevent the cost and implementation of automation from becoming a constraint on delivery.

Table 1. Five-gate AI-enabled SQL review framework: stages, functions, and tooling classes.

Gate	Stage	Primary Function	Tooling Class
1	IDE / Development	Real-time linting and anti-pattern detection	LLM-powered IDE plugin
2	Pull Request Review	Injection risk flagging, schema compatibility	AI-driven PR bot
3	CI Pipeline	Migration safety analysis, query cost estimation	Policy-as-code CI step
4	Staging / QA	EXPLAIN ANALYZE interpretation, baseline capture	Query plan analyser + LLM
5	Production Monitoring	Anomaly detection, regression alerting	ML-based observability platform

3.1 Gate 1 — IDE-Level Real-Time Analysis

The first gate applies SQL quality feedback at authoring time in the developer's integrated development environment (IDE). LLM-powered plugins such as GitHub Copilot with SQL workspace context or DataGrip's JetBrains AI Assistant, as well as organizations' own custom deployments of LLM APIs with injected database schema into the model context window, provide real-time linting, anti-pattern discovery and index recommendations while developers are typing [4]. Common issues include the use of SELECT * projections that make indexes ineffective, missing WHERE clause predicates that lead to full-table scans, and implicit type coercions that obstruct index seeks as well as the use of N+1 query patterns.

Because this approach provides immediacy to review and feedback (the author is working in the content at the time, as opposed to jumping back and forth to fix issues identified many hours or days earlier), and it is the simplest to implement (install the plugin and configure the schema context), Gate 1 is an ideal candidate for the first staged adoption. The main problem is the false positive rate, as over-linting valid queries leads to developer backlash and alert fatigue. Throughout pilot deployments, it was found that a false positive rate below an empirically-validated threshold is required before IDE gate findings are promoted to blocking [3].

3.2 Gate 2—Pull Request Automated Review

The second gate utilizes a generative artificial intelligence (AI) review bot that annotates SQL diffs in a pull request with severity levels. For example, CodeRabbit or GitHub Actions could invoke Claude or GPT-4 APIs and provide schema context to analyze SQL for a risk of code injection, schema compatibility with the target branch, execution plan predictions, and any policy violations, such as raw DDL in application service code [4]. Findings can be warnings (informational, non-blocking) or critical (blocking: the required check must be passed for the pull request to be merged). .

Gate 2 scans only this diff, creating a smaller input space than a whole codebase, with both the reduced computational cost and a smaller number of false positives in mind. Gate 2 also benefits from the availability of peer review context: the LLM can simultaneously reason about the intent of the change, the surrounding application logic, and the schema state of the target branch. This allows semantic analysis to be performed in addition to bare query evaluation [10].

3.3 Gate 3: Continuous Integration Pipeline Analysis

The third gate implements SQL quality analysis in the CI pipeline using the database migration scripts, which are often managed by a database migration tool such as Flyway, Liquibase, or Alembic. The migration SQL is parsed by a custom CI pipeline step and fed to the LLM API to classify it as high-risk or low-risk. This stage controls the deployment lifecycle through a Low, Medium, or High risk classification, which covers a column being deleted or a type changed to one not compatible with dependent services or an index being dropped without a deprecation period, as well as a query cost estimation when targeting a cloud data warehouse such as Snowflake, BigQuery or Amazon Redshift [5]. Other examples of specialized tooling include Bytebase with review rules baked in for database CI/CD workflows and Neon with migration safety APIs. By combining deterministic rule-based checks for known bad patterns with LLM-based risk scoring that assesses contextual safety, we can ensure a stronger gate than if we only relied on deterministic rules or LLM-based scoring alone. CI integration ensures that every proposed schema change is analyzed before being applied to the staging environment and avoids the type of incidents caused by unreviewed migrations.

3.4 Gate 4 - Staging Environment Query Plan Analysis

The fourth gate runs all queries against the staging database with EXPLAIN ANALYZE to capture the access plan and timing against production-like data distributions. An LLM processes the access plan, which requires domain expertise to parse, and produces human-readable recommendations to eliminate sequential scans on large tables, replace hash joins with index nested loop joins for selective predicates, and eliminate sort scans via carefully designed composite indexes [5].

Because Gate 4 uses real data distributions in production, it can detect performance regressions that only appear statistically meaningful in production. Additionally, it tracks the baseline of query performance for each service per release. Using these baselines, Gate 4 detects regressions: when a release changes the execution plan of an existing query to a less performant plan, the gate will fail the release. The human-readable report is added to the deployment ticket as a quality artifact, which the release management team can review.

3.5 Gate 5 cross-section runtime measurements

In the 5th gate, SQL performance quality checks are conducted in production using machine learning-based anomaly detection, comparing the observed performance of SQL vs. the traffic baseline. This requires telemetry architecture from tools such as Datadog Database Monitoring, Dynatrace, and Amazon RDS Performance Insights. This would match the AI components' query latency anomaly observations with deployments to identify the change in SQL that caused a regression [3]. When a similar pattern is identified based on query structure and frequency of execution, the gate would automatically inform the DBA team of a recommendation to create an index.

Gate 5 closes the quality loop. Gates 1 through 4 are how to avoid defects. Gate 5 understands what real-world coverage looks like, since a slow query anomaly event without any of the first four means a coverage gap upstream. The cause was either a defect class not covered by rules or a path through the query that was not executed during the staging. This feedback loop is how the ruleset grows and evolves.

4. Governance Architecture

4.1 Ruleset Ownership and Versioning

As the AI review rule set is a policy artifact, the ownership, versioning, and review frequency should be managed like any security policy or compliance control document. For example, the ownership of each rule category (security rules for injection, privilege escalation, and access pattern violation) could be assigned to the CISO and DBA Lead respectively with quarterly review frequency. Performance rules are owned by the Data Platform team, which reviews the rules monthly because of the frequency of changes to query patterns and underlying infrastructure capability. Style and maintainability rules are owned by Engineering Chapter Leads. Rules related to safe database migration are reviewed during major database framework upgrade cycles.

Every change to an AI review rule must be recorded in version control, along with the person who approved the change, the reason for the change, and a validation test that shows the changed rule behaves correctly for a set of positive and negative tests. Recording rule changes in this way prevents a failure mode of static analysis tooling programs known as "rule drift," where rules degrade in quality from incremental changes being made without undergoing review [9].

4.2 Human Override Mechanism

Developers can suppress any AI findings with an inline comment that includes the rule identifier, reason for suppression, and authorizing person. Suppressions are preserved in the version control system and are reviewed monthly. However, if the same rule is violated multiple times, a rule governance review is triggered. During this review, the system determines whether the rule is miscalibrated for the team in question or whether the team's practice actually violates the rule. This prevents blocking gates from creating barriers to productivity and enables an audit trail of rule conformance.

Table 2. Defect cost profile by discovery stage.

Defect Type	Typical Discovery Stage	Estimated Fix Cost	Impact Category
Missing index / full table scan	Production (P1 incident)	4–8 hrs + downtime	Availability
SQL injection risk	Security audit / pen test	Sprint-level remediation	Security
Destructive migration	Post-deploy rollback	Data recovery + hotfix	Data integrity
N+1 query pattern	Load testing / prod scale	Refactor + re-release cycle	Performance
Unoptimised cloud query	Monthly billing review	Excess compute credits	Cost

4.3 Data Privacy Controls

Using external LLM APIs to review the SQL query may expose data by sending the schema context (the names of the tables and columns) to the model provider as part of the prompt. As there is no data to serialize, sanitization to remove the bind variable values from the SQL before calling the API is required. Schema context goes through a data classification review and approval process before being sent externally [6]. For organizations with strict data residency requirements (such as being subject to the Article 46 adequacy mechanism under the General Data Protection Regulation or financial regulatory data localization policies), on-premise deployment of LLMs using a platform like Ollama , or Azure OpenAI deployed using a virtual network integration allows the same technical capabilities.

Table 3. Governance RACI matrix for AI-enabled SQL review programme (R = Responsible, A = Accountable, C = Consulted, I = Informed).

Activity	Dev Lead	DBA	Security	CISO	Platform Eng.
Define SQL policy ruleset	C	C	A	I	R
Tool evaluation and selection	C	C	C	A	R
PR bot configuration	R	C	C	I	A
CI gate integration	R	C	I	I	A
Staging analysis setup	R	R	I	I	A
Production monitoring setup	R	R	C	I	A

5. Phased Implementation Roadmap

The framework is rolled out in four phases over 24 weeks. The use of phases is both for risk management (the blocking gates are implemented when the pilot data has established a satisfactory false-positive rate and developer comfort) and for organizational change management (the next team is only onboarded to the new framework when there is enough evidence within the phase to justify the next step).

5.1 Phase 0: Baselining (Weeks 1-3)

Phase 0 establishes a measurement baseline for the program, measuring the SQL defect rate in two ways: by auditing incident log data, reviewing pull request comments, and monitoring production alerts raised for SQL defects over the past year. Pilot participants are chosen based upon delivery speed (enough pull requests to allow statistically meaningful observations to be made) and stakeholder participation (the team lead agreed to give developer feedback on gate findings). A shortlist of candidate tools is built based on the criteria defined in Section 3, and a proof of concept is run for the candidates that score highest against those criteria. Data classification for schema context is finalized and approved by the CISO before invoking any external APIs.

5.2 Phase 1 - IDE and PR Gates, Pilot (Weeks 4-8)

Phase 1 deploys Gates 1 and 2 for the pilot team in non-blocking mode, installs and configures the IDE plugins with the organization's schema context, enables the PR review bot, and marks all PR review results as warnings but does not enforce blocking. The developer Net Promoter Score (NPS) is measured at the end of week 4 and week 8. The false positive rate is tracked weekly through the use of a feedback form where the developers agree or disagree with the correctness of the individual findings. Phase 1 pass criteria are that the false positive rate is below an empirically calibrated threshold (pilot data suggest low single digits) and the developer NPS score is neutral or

positive. Phase 1 rule tuning must be iterative and based on feedback data, and all changes must be reviewed through the governance process (Section 4.1).

5.3 Phase 2: CI Gate and Team Expansion (Weeks 8-12)

Phase 2 adds Gate 3 to the CI pipeline as well as three more development teams to the program. The PR gate changes from warning-only to blocking-on-critical, meaning that all pull requests that would trigger a critical severity finding are blocked until the finding is remediated or suppressed with a documented statement of the reason for not remediating the finding (e.g., risk of SQL injection or destructive DDL without explicit DBA approval). A suppression mechanism is defined and recorded for developer self-service. Weekly defect dashboards are connected to engineering leadership, and the program is held accountable based on empirical data. The exit criterion for Phase 2 is not having critical findings that would result in merging with the integration branch without approval.

5.4 Phases 3 and 4: Full Rollout and Production Monitoring (Weeks 12-24)

Phase 3 includes setting up Gate 4 to run EXPLAIN ANALYZE in an automation and finishing scaling Gates 1, 2, and 3 organization-wide. Phase 3 also defines a baseline for per-service query performance, enabling the next service release to pick up regressions. Phase 4 has us bring up Gate 5 production monitoring, add anomaly detection alerts in the on-call runbook, derive JIRA tickets from slow query events pre-filled with AI-generated root cause hypotheses, and review the data six months later against the KPI baseline created in Phase 0.

6. Evaluation Methodology and Key Performance Indicators

Effectiveness is measured against a set of KPIs established prior to program implementation. The pre-post design, with measurements at baseline (Phase 0), controls for secular trends in delivery volume that may otherwise confound incident rate comparisons over time. Table 4 summarizes the full KPI set, baseline data collection methodology, six-month targets, and measurement instruments.

Table 4. Key performance indicators, targets, and measurement methodology.

KPI	Baseline	Target (Month 6)	Measurement Method
SQL-related P1/P2 incidents per month	Measured at M0	60% reduction	Incident tracker
Mean time to detect SQL issues	Measured at M0	70% decrease	PR analytics + logs
SQL defects caught pre-production	Measured at M0	>90%	JIRA / defect tracker
Mean time to SQL review (PR)	Measured at M0	<5 minutes	CI pipeline analytics
False positive rate from AI bot	N/A (new metric)	below an empirically-calibrated threshold (pilot data suggests low single digits)	Bot feedback mechanism
Repos with AI SQL review active	0%	100% by Month 7	CI/CD dashboard

The primary outcome measure is the SQL-attributable P1/P2 incident rate, extracted from the incident management platform and defined as the count of incidents for which root cause analysis identifies a SQL query, schema migration, or index definition as the triggering cause. The target of a 60% reduction by Month 6 is derived from published outcomes of shift-left static analysis programs applied to application source code [2], adjusted conservatively for the relative immaturity of SQL domain-specific tooling. Mean time to detect SQL issues—measured through pull request analytics and incident logs—targets a 70% decrease from the Phase 0 baseline,

reflecting the expected compression of detection latency as findings migrate from production into earlier stages. The proportion of SQL defects caught pre-production targets greater than 90%, measured via the organization's defect tracker. Mean time to SQL review at the pull request stage targets under 5 minutes through CI pipeline automation compared to the hours-long DBA review cycles characteristic of the current state. The false positive rate from the AI review bot—a new metric with no prior baseline—is targeted below an empirically calibrated threshold, monitored via a structured bot feedback mechanism through which developers classify individual findings as valid or erroneous. Finally, repository coverage targets 100% of repos with AI SQL review active by Month 7, tracked through the CI/CD pipeline dashboard; coverage begins at 0% at program inception.

Secondary measures track the directional shift in the detection distribution across gates. As the program matures, the proportion of SQL defects detected at the IDE and pull request gates should increase, while production-stage discoveries should correspondingly decline. This distributional shift is the operational hallmark of a successful shift-left program and provides a more granular quality signal than aggregate incident rate alone. The false positive rate KPI serves as a continuous programme health indicator: a rising rate signals rule set degradation or stale schema context and constitutes an early warning of impending developer adoption erosion that must be addressed before it compounds.

7. Discussion and Threats to Validity

One of the factors potentially inhibiting generalizability of predictions made by this framework is that LLM performance on SQL analysis tasks is sensitive to the quality of schema context provided in the prompt. Organizations with complex, poorly documented schemas, such as many legacy enterprise systems, will experience significantly worse finding accuracy compared to organizations with well-documented schemas. It may be reduced by a schema documentation hygiene initiative prior to Gate 1 and 2, which will increase implementation complexity beyond that shown on the roadmap timeline.

Second, the framework assumes that the production monitoring infrastructure required for Gate 5 is already available or can be provisioned in time for the program of work. For organizations with on-premise databases and no existing APM tooling, Gate 5 may not be practical without important investment in infrastructure. In such cases, the phased roadmap should either be extended to cover deployment of the APM in Phase 4 , or Gate 5 should be deferred.

Third, the governance model in Section 4 presumes the existence of institutional capacity to own the rules and cadence reviews. Small engineering organizations that do not have a dedicated DBA or security function may not be able to fulfill the RACI assignments. In these cases, the rule ownership model can be lightened. A single Technical Lead can share the security and performance rules. A semiannual review cadence would be appropriate. The suppression audit mechanism must not be omitted, regardless of organization scale, as its presence is the primary accountability signal of the program.

8. Conclusions

We present a five-gate SQL review framework based on AI to shift left database quality assurance in the software development life cycle. This framework uses LLM-based analysis to review SQL scripts in the integrated development environment (IDE), pull request, continuous integration (CI) pipeline, staging, and production. Each gate detects issues that it can best fix at that stage. Governance architecture includes human override mechanisms, data privacy controls, and formal ownership of rules to provide the organizational infrastructure necessary to scale an AI-assisted quality program across the enterprise.

The evidence on LLM capabilities in SQL analysis contexts continues to grow but is not yet mature. The performance goals in this paper are based on data from application code analysis programs and earlier pilot data, rather than randomized controlled trial data. Future work should focus on longitudinal testing of production deployments and generate peer-reviewed evidence for the effectiveness of the framework. A comparative evaluation of LLM provider accuracy when assessing SQL injection detection, execution plans, and migration risks across different schema types and database dialects would provide further evidence for practitioner adoption decision-making.

As AI-assisted development tooling becomes ubiquitous infrastructure, these patterns of process around gates, governance with accountability, and feedback loops will generalize to AI-assisted quality assurance for code beyond application logic that has been historically less automated. SQL code, having the closest ties to data accuracy, security, and the performance of the entire system; is the highest-value target for various classes of tooling; and is the most tractable place for organizations to show value through improved application quality with AI-assisted engineering.

REFERENCES

1. K. Schmid and M. Lindvall, "Assessing the reliability of software engineering methods and data," in Proc. 1st Int. Symp. Empirical Software Engineering (ISESE), pp. 103–113, IEEE, 2002. doi: 10.1109/ISESE.2002.1166928
2. B. W. Boehm and V. R. Basili, "Software defect reduction top 10 list," Computer, vol. 34, no. 1, pp. 135–137, Jan. 2001. doi: 10.1109/2.962984
3. M. Fowler, Refactoring: Improving the Design of Existing Code, 2nd ed. Boston, MA, USA: Addison-Wesley, 2018.
4. M. Chen et al., "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374
5. G. Moerkotte, "Query optimization in database systems," in Foundations of Databases, A. Silberschatz, H. F. Korth, and S. Sudarshan, Eds. New York, NY, USA: McGraw-Hill, 2019, ch. 16, pp. 574–622.
6. OWASP Foundation, "OWASP Top 10: 2021 — The Ten Most Critical Web Application Security Risks," OWASP, 2021. [Online]. Available: https://owasp.org/www-project-top-ten/
7. C. Jones, Applied Software Measurement: Global Analysis of Productivity and Quality, 3rd ed. New York, NY, USA: McGraw-Hill, 2008.
8. N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. Portland, OR, USA: IT Revolution Press, 2018.
9. A. Blundell et al., "SQLFluff: A dialect-agnostic SQL linter and formatter," GitHub, 2022. [Online]. Available: https://github.com/sqlfluff/sqlfluff
10. D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of ChatGPT," in Proc. IEEE/ACM Int. Workshop on Automated Program Repair (APR), pp. 23–30, 2023. doi: 10.1109/APR59189.2023.00011
11. A. Fan et al., "Large language models for software engineering: Survey and open problems," in Proc. Int. Conf. Software Engineering: Future of Software Engineering (ICSE-FOSE), pp. 31–53, IEEE, 2023. doi: 10.1109/ICSE-FOSE59343.2023.00008
12. T. Winters, T. Manshreck, and H. Wright, Software Engineering at Google: Lessons Learned from Programming Over Time. Sebastopol, CA, USA: O'Reilly Media, 2020.