

## ARTIFICIAL INTELLIGENCE

# PGNexus-AI: An Intelligent and Secure Natural-Language PostgreSQL Administration Framework

Shruti Mahendra Raskar, Brijendra Gupta

Department of Information Technology, Siddhant College of Engineering, Sudumbare (Savitribai Phule Pune University), Pune, Maharashtra, India

Published October 2026

## Abstract

Modern teams depend on PostgreSQL for mission-critical workloads, yet administration is still done by hand-writing SQL in disconnected tools such as psql, pgAdmin and DBeaver, while AI assistants such as GitHub Copilot have no live connection to the database and cannot read its schema, execution plans or error output. LLM-generated SQL also introduces new injection-style security risks. This paper presents PGNexus-AI, a PostgreSQL-only desktop IDE that combines Google Gemini, Gemma 4 and GitHub Copilot behind a schema-grounded orchestration layer. It provides natural-language-to-SQL generation, execution-plan-driven query optimization, automatic error diagnosis and correction, and AI-generated schema documentation with column-relationship mapping. A defense-in-depth safety layer (parameterization, query preview and confirmation, read-only and sandbox execution, role-based scoping and injection screening) protects production databases, and GitHub integration version-controls SQL and migrations. We describe the architecture, methodology and an evaluation plan covering efficiency, reliability, optimization and security.

**Keywords:** Database Security; Large Language Models; Natural Language to SQL; PostgreSQL; Query Optimization

## 1 Introduction

PostgreSQL is one of the most widely adopted open-source object-relational database management systems, relied upon by developers and database administrators (DBAs) for mission-critical applications ranging from transactional web back-ends to analytical data warehouses. Since its original design as the POSTGRES project, the system has been valued for its extensibility, standards compliance, and rich type system [1]. Despite this power, the day-to-day experience of interacting with PostgreSQL has changed remarkably little: users still compose Structured Query Language (SQL) by hand through tools such as the psql command-line client, pgAdmin, or DBeaver. These tools are mature and dependable, but they are fundamentally passive editors and result grids – they neither understand the intent behind a query nor actively help the user write, repair, or optimize it.

In parallel, large language models (LLMs) built on the transformer architecture [2] have demonstrated a remarkable capacity for few-shot generalization [3] and for translating natural language into executable code. Code-specialized models such as Codex, which underpins GitHub Copilot [4], and general-purpose model families such as Google Gemini [5], the openly released Gemma models [6], and Llama 2 [7] have made AI pair-programming a mainstream part of software development. However, this assistance is almost universally confined to the source-code editor. GitHub Copilot can autocomplete a SQL string embedded in an application file, but it has no live connection to the database that the string will ultimately run against. Consequently, it cannot inspect the real schema,

cannot see column names or foreign-key relationships, cannot read an execution plan, and cannot observe the actual error returned when a statement fails. The user is left to bridge this gap manually, copying suggestions back and forth and correcting them through repeated trial and error.

This disconnection has two consequences that motivate the present work. First, it produces inefficiency and a steep learning curve: less-experienced users struggle to discover the correct tables and join paths, while experienced DBAs waste time on repetitive schema look-ups and plan inspection. Second, and more critically, delegating SQL authorship to a probabilistic model introduces a new class of security risk. Because an LLM constructs SQL from language patterns rather than explicit logic, even innocuous-looking prompts can yield vulnerable or destructive queries, and adversarial prompts can be weaponized into prompt-to-SQL injection attacks [24, 25]. Any tool that places an LLM in the query-authoring loop must therefore treat security as a first-class design concern rather than an afterthought.

## 1.1 Related work

### 1.1.1 Large language models and natural-language-to-SQL

The task of translating natural language into SQL (NL2SQL, or text-to-SQL) has a long research history. Early neural approaches such as Seq2SQL framed the problem as sequence generation with reinforcement-learning rewards [8], and the release of the large, cross-domain Spider benchmark catalysed rapid progress by requiring generalization to unseen database schemas [9]. The arrival of general-purpose LLMs shifted the paradigm again: Rajkumar et al. showed that large models can produce competitive SQL with only a few in-context examples [10], while prompting strategies such as decomposed in-context learning with self-correction (DIN-SQL) closed much of the remaining gap to specialized models [11]. Benchmarks such as BIRD subsequently highlighted the difficulty of grounding generation in large, realistic databases with dirty values and performance constraints [12]. A recurring lesson across this literature, summarized in recent surveys [13], is that accuracy depends heavily on how well the model is grounded in the target schema. Retrieval-augmented generation (RAG), which conditions the model on retrieved context [14], is therefore a natural mechanism for supplying the relevant tables, columns and relationships to the model at generation time – a principle that PGNexus-AI adopts directly.

### 1.1.2 Learned and adaptive query optimization

Query optimization is one of the most persistent challenges in database systems. Traditional cost-based optimizers rely on statistical estimates and heuristics that frequently misjudge cardinality for multi-way joins, leading to sub-optimal plans. A large body of recent work applies machine learning to this problem. Kraska et al. demonstrated that learned index structures can outperform classical B-trees for specific workloads [17], while Neo showed that a learned optimizer can match or exceed commercial systems [18]. Reinforcement learning is a particularly natural fit for join-order enumeration, framing planning as sequential decision making [19, 20], and the Bao system made such learned optimization practical by steering an existing optimizer rather than replacing it [21]. Of direct relevance to this work, Ramadan et al. proposed RL\_QOptimizer, a reinforcement-learning join-order optimizer that reported a notable advantage over the native PostgreSQL optimizer in execution time [16], and Mandapaka described a framework for embedding gradient-boosting and neural models inside PostgreSQL, using its extensibility to improve cardinality estimation, adaptive indexing and workload management without modifying the core engine [15]. PGNexus-AI does not attempt to replace the PostgreSQL optimizer; instead, it makes the optimizer's own output – the EXPLAIN/

ANALYZE plan – legible to an LLM so that the system can explain and advise on performance in natural language.

### 1.1.3 SQL injection and the security of LLM-generated queries

SQL injection (SQLi) has remained among the OWASP Top 10 web-application risks for over a decade [27], and classical defenses such as randomized SQL keywords (SQLrand) [23] and, more durably, parameterized queries were developed for SQL authored by humans. Machine-learning and deep-learning detectors have pushed reported accuracy very high; Casmiry et al., for example, compared five architectures and found that a hybrid CNN-LSTM model reached 99.85% accuracy with a false-positive rate of only 0.06% on standard SQLi datasets [22]. The rise of LLM-integrated applications, however, has opened a new attack surface. Greshake et al. showed that indirect prompt injection can compromise real LLM-integrated applications [26], and this risk is now recognized in the emerging OWASP Top 10 for LLM Applications [28]. Pedro et al. characterized prompt-to-SQL (P2SQL) injections specifically, demonstrating vulnerabilities across seven LLMs and five real-world LangChain applications and proposing four defenses: database permission hardening, SQL query rewriting and parsing, an auxiliary LLM guard that inspects generated queries, and preloading in-prompt data to avoid over-broad access [24]. Most tellingly, Kim and Lee performed the first systematic evaluation of SQLi in LLM-generated SQL across fourteen models, executing expert-crafted attack prompts against SQLite, MySQL and PostgreSQL. They found that PostgreSQL was by far the most resilient back-end (only 1.7% of attacks succeeded, versus 8.1% for SQLite and 12.1% for MySQL) and, crucially, that conventional SQLi detectors degraded from roughly 98% accuracy on standard benchmarks to about 60% on LLM-generated queries [25]. These findings jointly justify PGNexus-AI's exclusive focus on PostgreSQL and its layered, detector-assisted safety design.

## 1.2 Research gap and contributions

The literature reveals a clear gap. AI coding assistants such as GitHub Copilot are powerful but database-blind: they operate inside the editor and never connect to a live server [4]. Established database clients such as pgAdmin, psql and DBeaver connect to the server but offer no LLM-driven reasoning, no schema-grounded natural-language interface, and no security layer specialized for AI-generated SQL. Research prototypes advance NL2SQL accuracy [10]-[13] and learned optimization [15]-[21] in isolation, and separate security studies expose the risks of LLM-generated SQL [24, 25] without offering an integrated, safety-first tool for practitioners. To the best of our knowledge, no dedicated desktop IDE integrates multi-model LLM assistance and GitHub Copilot directly with a live PostgreSQL server while remaining simultaneously schema-aware, optimization-aware, and security-hardened.

This paper addresses that gap by proposing PGNexus-AI. Its principal contributions are: (i) a layered architecture for a standalone, PostgreSQL-only desktop IDE that connects LLM reasoning to live database metadata and execution feedback; (ii) a schema-grounded natural-language-to-SQL pipeline that identifies the relevant tables and columns and maps column-to-column relationships through the foreign-key graph so that correct join paths are surfaced automatically; (iii) an execution-plan-driven optimization advisor that renders EXPLAIN/ANALYZE output into actionable, natural-language recommendations grounded in learned-optimization research [15, 16]; (iv) an automatic error-diagnosis and correction loop; (v) a defense-in-depth safety layer that operationalizes recent P2SQL defenses [24] and detector findings [22, 25]; (vi) GitHub-based version control of SQL and migrations; and (vii) a concrete evaluation methodology spanning efficiency, reliability, optimization and security. The remainder of the paper is organized as follows. Section II details the proposed

system and methodology; Section III presents an illustrative walkthrough, the expected results, and a comparative discussion; and Section IV concludes and outlines future work.

## 2 Methodology

### 2.1 System overview and architecture

PGNexus-AI is designed as a standalone Windows desktop application – packaged as a single executable – that presents a Visual Studio Code-style development environment dedicated exclusively to PostgreSQL. Restricting the tool to a single engine is a deliberate choice: it allows every layer, from metadata extraction to security hardening, to exploit PostgreSQL-specific features (the information schema, the system catalogs, EXPLAIN in JSON form, and role-based privileges) rather than settling for a lowest-common-denominator abstraction, and it aligns with the empirical finding that PostgreSQL is the most injection-resilient of the common back-ends [25]. The application follows a layered architecture, illustrated in Figure 1, in which each layer has a single, well-defined responsibility and communicates through explicit interfaces.

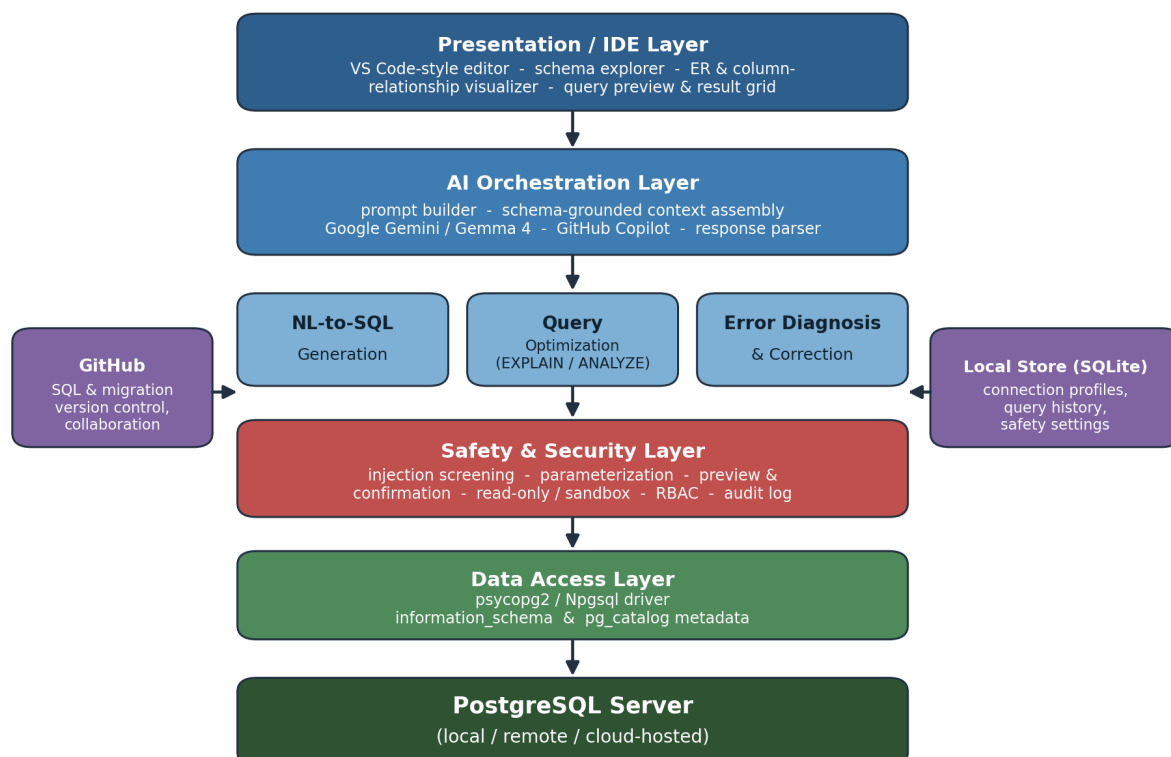


**Figure 1.** Layered architecture of PGNexus-AI, showing the presentation/IDE layer, the AI orchestration layer that coordinates google gemini, gemma 4 and GitHub copilot, the three core AI modules, and the safety, data-access and persistence layers around a connected PostgreSQL server

The presentation layer provides the editor, a schema explorer, an entity-relationship and column-relationship visualizer, and a query-preview and result grid. Beneath it, the AI orchestration layer is the heart of the system: it assembles schema-grounded prompts, routes requests to the appropriate model, and parses model responses into structured actions. Three core modules – natural-language-

to-SQL generation, query optimization, and error diagnosis and correction – draw on this orchestration layer. Every statement produced by these modules passes through the safety and security layer before it reaches the data-access layer, which uses the native PostgreSQL driver to execute statements and to read metadata from the information schema and system catalogs. A local SQLite store holds connection profiles, query history and safety settings, while a GitHub integration version-controls SQL scripts and migrations. This separation ensures that AI convenience is never allowed to bypass the safety controls that protect the database.

## 2.2 PostgreSQL connectivity and live metadata extraction

The data-access layer establishes live connections to local, remote, or cloud-hosted PostgreSQL servers using a mature driver (psycopg for the Python core, or Npgsql for a .NET build). Beyond executing statements, this layer continuously and non-invasively harvests metadata that becomes the factual basis for all AI reasoning. Table and column definitions are read from `information_schema.tables` and `information_schema.columns`; primary and foreign keys and other constraints are read from `information_schema.table_constraints` and the `pg_catalog` relations (`pg_constraint`, `pg_attribute`, `pg_class`); and lightweight statistics such as row-count estimates and index definitions are read from `pg_stat_user_tables`, `pg_stats` and `pg_indexes`. By sourcing this information directly from the server rather than inferring it, PGNexus-AI guarantees that the model always reasons over the true, current state of the database – precisely the capability that an editor-bound assistant lacks.

## 2.3 Schema intelligence and column-relationship mapping

A distinguishing feature of PGNexus-AI is that it does not treat the schema as an opaque list of tables. Instead, it constructs an in-memory schema graph in which nodes are tables and columns and edges encode relationships – most importantly the foreign-key edges that connect a column in one table to its referenced column in another. This graph is the mechanism by which the system delivers the practical goals of proper column identification and proper column-to-column connection. For any pair of tables the user wishes to combine, PGNexus-AI can traverse the graph to infer the correct join path and the exact key columns involved, eliminating the guesswork that makes multi-table queries error-prone. The same graph powers three user-facing features: an entity-relationship diagram that visualizes tables and their foreign-key links; a per-column inspector that reports a column's data type, nullability, default, participation in keys and indexes, and the other columns it references or is referenced by; and AI-generated schema documentation in which the LLM produces plain-language descriptions of each table and column, helping teams maintain an up-to-date understanding of their database as it evolves. Because the graph is derived from live metadata, it also enables lightweight impact analysis: before a column is altered or dropped, PGNexus-AI can list the constraints, indexes and dependent objects that reference it.

## 2.4 Natural-language-to-SQL generation

The NL2SQL module turns a user's request in plain English into an executable, schema-correct PostgreSQL statement. Rather than sending the request to a model in isolation – the approach that produces plausible but wrong column names – PGNexus-AI follows a retrieval-augmented, schema-grounded strategy inspired by RAG [14] and by the grounding lessons of the text-to-SQL literature [11]-[13]. As shown in Figure 2, the module first retrieves the tables and columns most relevant to the request from the schema graph of Section II-C, together with the foreign-key edges that connect them. It then assembles a structured prompt containing these definitions, a small number of

illustrative examples, the relevant relationships, and explicit instructions to emit parameterized, PostgreSQL-dialect SQL. The prompt is dispatched to a configurable model backend – Google Gemini or the lighter-weight Gemma 4 for local or cost-sensitive deployments [5, 6], with GitHub Copilot available for developers already working inside a code project [4]. The model returns one or more candidate queries, which are parsed, validated against the known schema, and presented to the user with a natural-language explanation before any execution is considered.

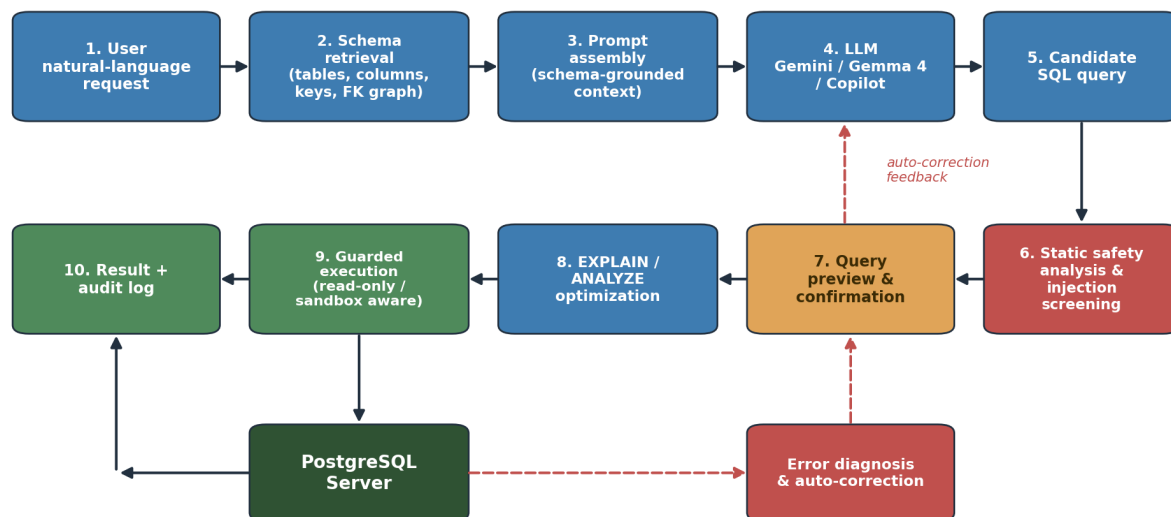


**Figure 2.** End-to-end query pipeline in PGNexus-AI: A natural-language request is grounded in retrieved schema and relationships, translated to SQL by the selected LLM, screened and previewed under the safety layer, optimized via EXPLAIN/ANALYZE, and executed under read-only or sandbox constraints, with an error-driven auto-correction feedback loop

## 2.5 AI-assisted query optimization

Once a query is validated, PGNexus-AI can act as an optimization advisor. It executes the statement under EXPLAIN (or EXPLAIN ANALYZE in a safe context) and captures the plan in JSON form, extracting structured features such as the join order, the physical operators chosen, the estimated versus actual row counts, and the most expensive nodes. These features – not the raw text – are supplied to the LLM, which is prompted to explain the plan in plain language and to propose concrete improvements: adding or adjusting an index, rewriting a correlated subquery as a join, avoiding an implicit type cast that defeats an index, or reordering predicates. This design is informed by, and complementary to, the learned-optimization literature. Whereas RL\_QOptimizer learns join orders that outperform the native PostgreSQL optimizer [16] and Mandapaka's framework embeds ML models to sharpen cardinality estimates and indexing inside the engine [15], PGNexus-AI operates one level up: it does not override the optimizer but makes its reasoning transparent and actionable to the human, and it can recommend the very indexing and rewriting changes that those systems exploit automatically. The captured plan features also give the model concrete evidence of cardinality misestimation, one of the best-documented causes of poor plans [17, 18].

## 2.6 Automatic error diagnosis and correction

When a statement fails, PGNexus-AI captures the full PostgreSQL error, including the SQLSTATE code, the message, and any position or hint the server provides. This structured error, together with the offending statement and the relevant schema context, is passed to the LLM, which diagnoses the cause – an unknown column, a type mismatch, an ambiguous reference, a violated constraint – and proposes a corrected statement. The correction is not applied silently; it is presented to the user through the same preview-and-confirm gate as any other generated query, and the loop can iterate if the corrected statement also fails. This closes the trial-and-error cycle that dominates manual debugging, but keeps the human in control of what actually runs.

## 2.7 Security and safety layer

Because PGNexus-AI places a probabilistic model in the query-authoring loop, security is treated as a primary design concern rather than an add-on. The threat model draws directly on recent findings: prompt-to-SQL injection, in which a crafted natural-language input coerces the model into emitting a malicious query [24]; the broader risk that LLM-generated SQL is exploitable even for benign-looking prompts [25]; destructive statements such as an unqualified DELETE or DROP; and data exfiltration through over-broad SELECTs or over-privileged connections. In response, PGNexus-AI applies defense in depth. First, all generated data-manipulation queries are parameterized so that user-supplied values are never concatenated into SQL text. Second, every statement is subjected to static safety analysis and injection screening before it can run; the screening combines rule-based checks (for stacked queries, comment-based evasion, tautologies and destructive verbs) with an optional learned detector modelled on the hybrid CNN-LSTM classifier of Casmiry et al., which reached 99.85% accuracy on standard SQLi data [22]. Because such detectors are known to degrade on LLM-generated queries [25], screening is a defensive aid and never the sole control. Third, a query preview and explicit confirmation gate stands between generation and execution, and destructive operations require additional confirmation. Fourth, connections default to a read-only or sandbox mode, and least-privilege database roles restrict what any session can touch – the permission-hardening and role-restriction defenses recommended by Pedro et al. [24]. An optional auxiliary LLM guard, also proposed in that work, can inspect a generated query for malicious intent before it is shown to the user. Finally, every generated query, correction and execution outcome is written to a structured, tamper-evident audit log, supporting accountability and post-hoc review. Table 1 maps each identified threat to the corresponding PGNexus-AI mitigation.

**Table 1.** Threats to a live database from LLM-assisted querying and the corresponding PGNexus-AI mitigations.

| Threat                                                                    | PGNexus-AI mitigation                                                                                                                       |
|---------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| Prompt-to-SQL (P2SQL) injection via a crafted natural-language input [24] | Injection screening (rule-based and CNN-LSTM), parameterization, and an optional auxiliary LLM guard that inspects the generated query      |
| Exploitable SQL produced from a benign-looking prompt [25]                | Query preview and explicit confirmation gate before any execution; exclusive targeting of PostgreSQL, the most injection-resilient back-end |
| Destructive statements (unqualified DELETE, DROP, TRUNCATE)               | Static safety analysis flags destructive verbs; additional confirmation required; read-only or sandbox execution by default                 |
| Data exfiltration via an over-broad SELECT or an over-privileged session  | Least-privilege database roles and role-based scoping; in-prompt data preloading to avoid over-broad access [24]                            |
| Silent or unaudited changes to the database                               | Structured, tamper-evident audit log of every generated query, correction and outcome; GitHub version control of SQL and migrations         |

| Threat                                                                     | PGNexus-AI mitigation                                                                                                |
|----------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| Over-reliance on a single detector that degrades on LLM-generated SQL [25] | Defense in depth, so that no single control is authoritative: layered parameterization, roles, preview and screening |

## 2.8 GitHub integration and version control

Database logic is code, and it deserves the same version control as application source. PGNexus-AI integrates with GitHub so that SQL scripts, schema snapshots and migration files can be committed, diffed and reviewed. This provides an auditable history of how a schema evolved, supports collaboration and code review for database changes, and complements the audit log by preserving the intent behind each change. For developers working inside a repository, GitHub Copilot can be invoked for in-context completions, while PGNexus-AI retains responsibility for grounding, safety and execution against the live server.

## 2.9 Implementation and packaging

The reference implementation targets Windows and is packaged as a standalone executable. The core application logic and desktop user interface are built with a cross-platform stack (an Electron front-end over a Python service, or a .NET/C# implementation), and the application is bundled with PyInstaller or Electron Builder so that end users install a single artifact with no external runtime dependencies. Live connectivity, execution and metadata retrieval use psycopg or Npgsql; a local SQLite database stores connection profiles, query history and per-connection safety settings; and a structured logging module records AI-generated queries, corrections and execution outcomes for auditability. Model access is abstracted behind a provider interface so that Gemini, Gemma 4, or Copilot can be selected per connection without changing the rest of the system.

## 2.10 Evaluation methodology

PGNexus-AI is evaluated along four dimensions – efficiency, reliability, optimization quality and security – against representative baselines, as summarized in Table 2. Efficiency is measured as the time and number of interactions required to author correct queries for a fixed set of administrative tasks, compared with manual authoring in pgAdmin and DBeaver. Reliability is measured as the execution-success rate of generated queries and the success rate of the automatic error-correction loop. NL2SQL correctness is assessed with execution-match accuracy on established cross-domain benchmarks such as Spider [9] and BIRD [12]. Optimization quality is assessed by comparing the estimated and measured cost of queries before and after the advisor's recommendations are applied, using the native PostgreSQL optimizer as the baseline in the spirit of [16]. Security is assessed by re-playing tiered attack prompts modelled on the methodology of Kim and Lee [25] and measuring the proportion of malicious queries blocked by the safety layer together with the false-positive rate on benign queries, with the CNN-LSTM screening component evaluated as in [22]. Usability is captured through a System Usability Scale survey administered to developers, DBAs and students.

**Table 2.** Evaluation dimensions, metrics and baselines for PGNexus-AI.

| Dimension   | Metric                                                    | Baseline / method                       |
|-------------|-----------------------------------------------------------|-----------------------------------------|
| Efficiency  | Time and number of interactions to author a correct query | Manual authoring in pgAdmin and DBeaver |
| Reliability | Execution-success rate; auto-correction success rate      | First-attempt generated queries         |

| Dimension            | Metric                                                           | Baseline / method                                                          |
|----------------------|------------------------------------------------------------------|----------------------------------------------------------------------------|
| NL2SQL correctness   | Execution-match accuracy                                         | Spider [9] and BIRD [12] benchmarks                                        |
| Optimization quality | Estimated and measured cost before vs. after advice              | Native PostgreSQL optimizer [16]                                           |
| Security             | Malicious queries blocked; false-positive rate on benign queries | Tiered attack prompts per Kim & Lee [25]; CNN-LSTM per Casmiry et al. [22] |
| Usability            | System Usability Scale (SUS) score                               | Survey of developers, DBAs and students                                    |

### 3 Results and discussion

PGNexus-AI is presented here as a system design with an accompanying evaluation plan; the results in this section are therefore expected outcomes and illustrative analyses, grounded wherever possible in quantitative findings reported by the cited literature. We first walk through a representative end-to-end scenario, then discuss the efficiency, optimization and security outcomes we anticipate, and finally compare PGNexus-AI with existing tools and state the limitations of the approach.

#### 3.1 Illustrative end-to-end scenario

Consider a DBA who asks, in plain English, "show the total amount invoiced to each customer in the last quarter, most valuable first." PGNexus-AI retrieves the customers, invoices and invoice\_items tables from the schema graph, recognizes through the foreign-key edges that invoices.customer\_id references customers.id and that invoice\_items.invoice\_id references invoices.id, and assembles a schema-grounded prompt. The selected model returns a parameterized query that joins the three tables along exactly those key columns, filters on the invoice date, aggregates with SUM, groups by customer and orders the result descending. The safety layer screens the statement, finds no injection or destructive pattern, and presents it in the preview pane with a plain-language explanation. The user runs EXPLAIN; the advisor notes a sequential scan on a large invoices table and recommends an index on invoices(customer\_id, invoice\_date). After the user accepts, the query is executed read-only, the result is returned, and the whole interaction – prompt, query, plan and outcome – is written to the audit log and, if the user chooses, committed to GitHub. What would have required manual schema look-ups, hand-written joins, and separate plan inspection is completed in a single grounded, guarded interaction.

#### 3.2 Expected efficiency and reliability gains

Because PGNexus-AI supplies the model with true schema and relationship context, we expect a substantial reduction in the time and interactions needed to author correct multi-table queries relative to manual composition, and a marked reduction in the failed-execution rate that follows from mistaken column or join assumptions. The automatic error-correction loop is expected to resolve the majority of first-attempt failures without manual intervention, converting the conventional edit-run-read-error cycle into a single confirm step. These expectations are consistent with the demonstrated ability of well-grounded LLMs to generate executable SQL on unseen schemas [10, 11] and with the accuracy gains that schema grounding provides in the text-to-SQL literature [13].

### 3.3 Expected optimization improvements

The optimization advisor's value rests on making the optimizer's reasoning actionable. The underlying opportunity is well established: learned and reinforcement-learning optimizers have repeatedly improved on classical cost-based planning, with RL\_QOptimizer reporting a notable execution-time advantage over the native PostgreSQL optimizer for join ordering [16], and in-engine ML frameworks improving cardinality estimation and adaptive indexing [15, 18]. PGNexus-AI surfaces the same categories of improvement – better indexing, join and predicate rewriting, and correction of cardinality-defeating patterns – as human-readable recommendations, so that a user who is not an optimization expert can still realize a large share of the available gains. Because the advice is derived from the actual plan rather than from heuristics, it adapts to the specific data distribution of the connected database.

### 3.4 Expected security outcomes

The security posture of PGNexus-AI benefits first from its platform choice. Kim and Lee found PostgreSQL to be markedly more resilient to SQLi in LLM-generated queries than the alternatives, with only 1.7% of attacks succeeding against it compared with 8.1% for SQLite and 12.1% for MySQL [25]; Figure 3(a) reproduces these figures. By targeting PostgreSQL exclusively, PGNexus-AI starts from the most defensible baseline. On top of that baseline, the layered controls of Section II-G are expected to block the large majority of attempted injections while keeping false positives on benign queries low. It is important to be realistic about detector limitations: as Figure 3(b) shows, conventional SQLi detectors that reach roughly 98% accuracy on standard benchmarks fall to about 60% on LLM-generated queries [25], even though a hybrid CNN-LSTM detector attains 99.85% on conventional data [22]. PGNexus-AI therefore treats learned screening as one layer among several, relying on parameterization, least-privilege roles, read-only and sandbox defaults, preview and confirmation, and the auxiliary LLM guard [24] so that no single point of failure can expose the database.

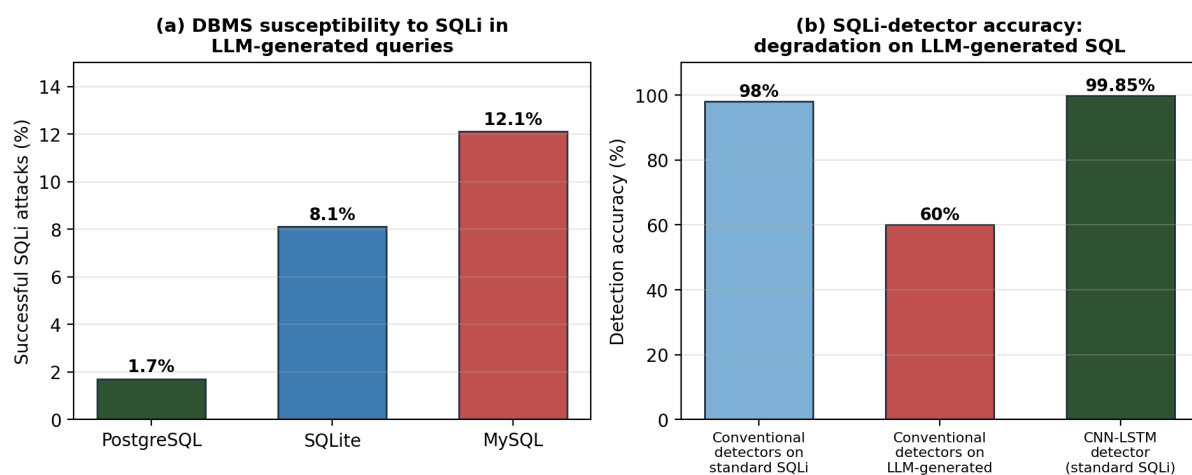


**Figure 3.** Security evidence motivating PGNexus-AI's design: (a) proportion of successful SQL-injection attacks in LLM-generated queries by database engine, showing PostgreSQL as the most resilient; and (b) degradation of conventional SQLi detectors on LLM-generated queries relative to standard benchmarks. values are as reported by kim and lee [25] and casmiry et al. [22]

### 3.5 Comparison with existing tools

Table 3 compares PGNexus-AI with the tools practitioners use today. The command-line client and graphical administration tools connect to the server but provide no AI reasoning, no schema-grounded natural-language interface, and no injection-aware safety layer for generated SQL. GitHub Copilot provides powerful in-editor assistance but is database-blind. PGNexus-AI is distinguished by combining live connectivity, multi-model AI reasoning, column-relationship visualization, plan-based optimization advice, automatic error correction, an integrated safety layer, and version control in a single dedicated PostgreSQL IDE.

**Table 3.** Comparison of PGNexus-AI with existing PostgreSQL interaction tools.

| Capability                               | psql | pgAdmin | DBeaver | GitHub Copilot | PGNexus-AI |
|------------------------------------------|------|---------|---------|----------------|------------|
| Live PostgreSQL connection               | Yes  | Yes     | Yes     | No             | Yes        |
| Graphical IDE experience                 | No   | Yes     | Yes     | Partial        | Yes        |
| Natural-language-to-SQL                  | No   | No      | Partial | Partial        | Yes        |
| Schema-grounded AI context               | No   | No      | No      | No             | Yes        |
| Column-relationship visualization        | No   | Partial | Yes     | No             | Yes        |
| Execution-plan-based optimization advice | No   | Partial | Partial | No             | Yes        |
| Automatic error diagnosis and correction | No   | No      | No      | Partial        | Yes        |
| Injection-aware safety layer for AI SQL  | No   | No      | No      | No             | Yes        |
| Query preview, confirmation and sandbox  | No   | Partial | Partial | No             | Yes        |
| GitHub version control of SQL/migrations | No   | No      | Partial | Yes            | Yes        |

### 3.6 Limitations and threats to validity

Several limitations should be acknowledged. First, PGNexus-AI depends on external LLMs whose behaviour can change with model updates and whose latency and cost must be managed; the provider abstraction and the option to use the lighter Gemma 4 model mitigate but do not eliminate this dependency. Second, the expected results reported here are grounded in the cited literature and in a designed evaluation rather than in a completed user study; the quantitative efficiency, optimization and security outcomes must be confirmed empirically. Third, learned injection detectors degrade on LLM-generated SQL [25], which is precisely why the safety layer never relies on any single control. Fourth, exclusive support for PostgreSQL, while beneficial for depth and security, limits applicability for organizations that also operate other engines; extending support is left to future work. Finally, sending schema metadata to a hosted model raises confidentiality considerations for sensitive databases, which the local-model option and on-premises deployment are intended to address.

## 4 Conclusion

This paper introduced PGNexus-AI, a standalone desktop IDE that integrates Google Gemini, Gemma 4 and GitHub Copilot directly with a live PostgreSQL server to make database administration more intelligent, efficient, and reliable. By grounding every AI action in metadata read from the information schema and system catalogs, and by mapping column-to-column relationships through the foreign-key graph, PGNexus-AI delivers natural-language-to-SQL generation that produces correct join paths, an optimization advisor that turns EXPLAIN/ANALYZE plans into actionable recommendations, and

an automatic error-correction loop – all behind a defense-in-depth safety layer that responds to the specific risks of LLM-generated SQL. The design is motivated by, and consistent with, recent evidence that PostgreSQL is the most injection-resilient common back-end, that conventional detectors degrade sharply on model-generated queries, and that learned optimization can improve on classical planning. In addressing the objectives set out for the project – connectivity, natural-language querying, optimization, error correction, schema documentation and safety – PGNexus-AI closes the gap between AI-assisted coding and AI-assisted database administration. Future work will complete the empirical evaluation with a user study, extend support to additional engines such as MySQL and SQL Server, add role-based access control for team deployments, provide a cloud-hosted edition, and investigate fine-tuning a PostgreSQL-specialized model to further improve accuracy and reduce dependence on hosted APIs.

## Acknowledgment

The authors gratefully acknowledge the guidance of the project supervisor and the Department of Computer Engineering, TSSM's PVPIT, Savitribai Phule Pune University, Pune. This research received no external funding, and the author declares no competing interests.

## References

- [1] Stonebraker M, Rowe LA. The design of POSTGRES. *ACM SIGMOD Rec.* 1986;15(2):340–355. <https://doi.org/10.1145/16856.16888>
- [2] Vaswani A, et al. Attention is all you need. In: *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*. 2017. p. 5998–6008. <https://doi.org/10.48550/arXiv.1706.03762>
- [3] Brown TB, et al. Language models are few-shot learners. In: *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*. 2020. p. 1877–1901. <https://doi.org/10.48550/arXiv.2005.14165>
- [4] Chen M, et al. Evaluating large language models trained on code. *arXiv:2107.03374* [Preprint]. 2021.
- [5] Gemini Team, Google. Gemini: A family of highly capable multimodal models. *arXiv:2312.11805* [Preprint]. 2023.
- [6] Gemma Team, Google DeepMind. Gemma: Open models based on Gemini research and technology. *arXiv:2403.08295* [Preprint]. 2024.
- [7] Touvron H, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv:2307.09288* [Preprint]. 2023.
- [8] Zhong V, Xiong C, Socher R. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv:1709.00103* [Preprint]. 2017.
- [9] Yu T, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In: *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*. 2018. p. 3911–3921. <https://doi.org/10.18653/v1/D18-1425>
- [10] Rajkumar N, Li R, Bahdanau D. Evaluating the text-to-SQL capabilities of large language models. *arXiv:2204.00498* [Preprint]. 2022.
- [11] Pourreza M, Rafiei D. DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction. In: *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*. 2023. <https://doi.org/10.48550/arXiv.2304.11015>
- [12] Li J, et al. Can LLM already serve as a database interface? A big bench for large-scale database grounded text-to-SQLs (BIRD). In: *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*. 2023. <https://doi.org/10.48550/arXiv.2305.03111>
- [13] Katsogiannis-Meimarakis G, Koutrika G. A survey on deep learning approaches for text-to-SQL. *VLDB J.* 2023;32(4):905–936. <https://doi.org/10.1007/s00778-022-00776-8>
- [14] Lewis P, et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. In: *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*. 2020. p. 9459–9474. <https://doi.org/10.48550/arXiv.2005.11401>

- 
- [15] Mandapaka S. Adaptive machine learning techniques for PostgreSQL performance optimization: A PostgreSQL case study. *Int. J. Innov. Sci. Res. Technol.* 2025;10(10):3375–3381. <https://doi.org/10.38124/ijisrt/25oct1435>
  - [16] Ramadan M, El-Kilany A, Mokhtar HMO, Sobh I. RL\_QOptimizer: A reinforcement learning based query optimizer. *IEEE Access.* 2022;10:70502–70515. <https://doi.org/10.1109/ACCESS.2022.3187102>
  - [17] Kraska T, Beutel A, Chi EH, Dean J, Polyzotis N. The case for learned index structures. In: *Proc. ACM SIGMOD Int. Conf. Manage. Data.* 2018. p. 489–504. <https://doi.org/10.1145/3183713.3196909>
  - [18] Marcus R, et al. Neo: A learned query optimizer. *Proc. VLDB Endow.* 2019;12(11):1705–1718. <https://doi.org/10.14778/3342263.3342644>
  - [19] Marcus R, Papaemmanouil O. Deep reinforcement learning for join order enumeration. In: *Proc. 1st Int. Workshop Exploiting Artif. Intell. Techn. Data Manage. (aiDM).* 2018. p. 1–4. <https://doi.org/10.1145/3211954.3211957>
  - [20] Krishnan S, Yang Z, Goldberg K, Hellerstein JM, Stoica I. Learning to optimize join queries with deep reinforcement learning. *arXiv:1808.03196 [Preprint]*. 2018.
  - [21] Marcus R, et al. Bao: Making learned query optimization practical. In: *Proc. ACM SIGMOD Int. Conf. Manage. Data.* 2021. p. 1275–1288. <https://doi.org/10.1145/3448016.3452838>
  - [22] Casmiry EN, Sinde RS, Mduma NM. A hybrid deep learning model for SQL injection attack detection. *IEEE Access.* 2026;14:6450–6463. <https://doi.org/10.1109/ACCESS.2026.3651991>
  - [23] Boyd SW, Keromytis AD. SQLrand: Preventing SQL injection attacks. In: *Applied Cryptography and Network Security (ACNS), LNCS.* 2004. p. 292–302. [https://doi.org/10.1007/978-3-540-24852-1\\_21](https://doi.org/10.1007/978-3-540-24852-1_21)
  - [24] Pedro R, Coimbra ME, Castro D, Carreira P, Santos N. Prompt-to-SQL injections in LLM-integrated web applications: Risks and defenses. In: *Proc. IEEE/ACM 47th Int. Conf. Softw. Eng. (ICSE).* 2025. p. 1768–1783. <https://doi.org/10.1109/ICSE55347.2025.00007>
  - [25] Kim E, Lee S. SQL injection in LLM-generated queries: Systematic analysis of detection gaps and security risks. *IEEE Access.* 2026;14:12797–12815. <https://doi.org/10.1109/ACCESS.2026.3654822>
  - [26] Greshake K, Abdelnabi S, Mishra S, Endres C, Holz T, Fritz M. Not what you’ve signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In: *Proc. 16th ACM Workshop Artif. Intell. Security (AISec).* 2023. p. 79–90. <https://doi.org/10.1145/3605764.3623985>
  - [27] OWASP Foundation. OWASP Top 10:2021 – The ten most critical web application security risks. 2021. <https://owasp.org/Top10/>
  - [28] OWASP Foundation. OWASP Top 10 for large language model applications. 2023. <https://owasp.org/www-project-top-10-for-large-language-model-applications/>