Database Context Engineering: Beyond Agentic RAG

Qazi Saad @ Threecolts

Abstract

Engineers working with large scale databases spend excessive time building context before they can even begin writing queries. With thousands of tables being legacy, poorly named, or undocumented the only reliable context often comes from engineers who created them, who may no longer be around. Query building becomes slow, error prone, and detached from business value, with verification loops taking days per metric. Vanilla Retrieval Augmented Generation (RAG) [1] attempts to address this but fails due to rigid workflows, poor handling of short queries, missing organizational context, and lack of feedback. Agentic RAG improves upon this by giving the model flexibility to use semantic search and other tools as needed, but it still struggles with hallucinations, misleading metadata, weak embeddings on schema, and unclear table usage. We present an improved Agentic RAG with Context Engineering. By profiling tables and columns, performing intelligent sampling for non profiler databases, generating LLM based multi aspect summaries, and enriching embeddings with structured context, we provide the model with the right information at the right time. Guardrails prevent unsafe queries, while human and LLM feedback loops refine accuracy iteratively. The result is a system that reduces query building from $\sim$ 2 days to 5 minutes per metric.

1 Introduction

In modern data teams, one of the most expensive and invisible costs is **context discovery**. An engineer given a business requirement say, to calculate monthly recurring revenue may have to navigate through thousands of tables, many deprecated, inconsistently named, or undocumented. Even once the relevant tables are found, nuances like discounts, subscription states, or custom churn definitions make query correctness difficult to establish. This process often requires back and forth between dashboards (verifying numbers) or subject matter experts, consuming entire days per metric.

Cataloging tables and columns could help, but in fast moving companies it is rarely prioritized, as it does not directly drive revenue. Instead, engineers inherit scattered tribal knowledge that leaves new hires or returning team members at a disadvantage. The outcome is wasted time: onboarding, rediscovering, and revalidating context.

Retrieval Augmented Generation (RAG) offered a promising direction: index table metadata, use embeddings for semantic search, and let an LLM generate queries. But **Vanilla RAG** [2] is rigid: it always retrieves regardless of query type, and cannot adapt when user queries are short, vague, or aspect specific. **Agentic RAG** [3] introduced autonomy, letting the model decide when and how to search, but even this approach falters without accurate metadata, strong embeddings, and guardrails against hallucination. Our work builds on Agentic RAG and advances it with **Context Engineering [4]:** the deliberate construction and injection of all relevant metadata, summaries, and constraints into the LLM's context window. Rather than leaving the model to

improvise on weak signals, we provide structured knowledge: table and column level statistics, sample values, multi aspect descriptions, and explicit cataloging outputs. Combined with human feedback and safe SQL execution, this approach makes query generation both faster and more reliable.

2 The Problem

Engineers tasked with building queries in large scale databases first need to build **context**. This means identifying which tables to use, how they relate, and what each column actually represents. In modern warehouses, there can be **thousands of tables**, including legacy or deprecated ones with unclear names. Many organizations lack a proper catalog, so the only context comes from engineers who created or once worked with those tables. If those engineers have left, the knowledge is effectively gone. For a new engineer, or even someone returning to the same project after months, this creates a costly cycle of rediscovery. They must navigate through unfamiliar schemas, experiment with queries, and verify results against dashboards or ground truth. Verification itself is non trivial. A query may look correct at first glance but miss important business rules. For example, calculating Monthly Recurring Revenue (MRR) from Stripe data is not as simple as the sum of monthly subscription amount and yearly subscription amount / 12. One must consider discounts, subscription states (active, trialing, past due), and how the company defines churn. Each of these details changes the outcome, and they are rarely encoded in schema names alone. This results in inefficiency between teams. Building and verifying a single metric can take **two days of engineer time or more**. This does not directly improve business metrics, yet it consumes resources from both the querying engineer and those who must help onboard them. The cost compounds as teams grow, databases evolve, and legacy systems persist. Traditional solutions, such as data catalogs, face resistance: they require ongoing manual effort, yet leadership sees them as overhead rather than value creation. In fast moving companies, the pace of change outstrips documentation, leaving engineers dependent on tribal knowledge and trial and error. This is the context in which Retrieval Augmented Generation (RAG) was introduced, promising automatic context retrieval. But as we will see, the **naive form of RAG** inherits its own set of problems.

3 Naive RAG

Retrieval Augmented Generation (RAG) emerged as a way to reduce context discovery. The idea is simple:

1. Extract metadata from database tables.
2. Encode it into embeddings.
3. Store it in a vector database.
4. At query time, retrieve the top-k most relevant metadata and pass it to the LLM.

This pipeline promises that instead of searching manually through thousands of tables, the LLM can be guided toward the right subset. However, in practice, naive RAG is rigid and limited:

- **Always retrieves:** Even for simple tasks like converting a PostgreSQL query to MySQL syntax, the system still runs a retrieval step. This wastes compute and sometimes pollutes the context window with irrelevant information.
- **Short queries fail:** Human written query (e.g., "return all churned customers") gets passed as it is to the vector DB, as engineers often write shorthand queries to get quick answers. These are too vague for semantic search, which performs poorly without enough descriptive signal.
- **Missing organizational context:** Metadata alone does not have business logic behind analytics, or which tables are authoritative versus deprecated. Without this, retrieval may surface irrelevant or outdated tables.
- **Single query limitation:** If the user wants to compare two tables, vanilla RAG treats it as one retrieval step and often only surfaces one table. True comparisons require multiple retrievals, which the pipeline does not support.
- **No retry logic:** A poor retrieval result is accepted as final. The system does not attempt alternative searches.
- **No feedback loop:** Neither the LLM nor the human operator can guide the system to improve its search in subsequent iterations.

The result is that naive RAG provides only partial help. It reduces some discovery work but does not significantly change the overall cycle of building and verifying queries. Engineers still spend days iterating, because the model lacks flexibility and cannot adapt to the complexity of real world database tasks.

This led to the development of **Agentic RAG** a more autonomous approach.

4 Agentic RAG

To address the issues from naive RAG, **Agentic RAG** was introduced. Instead of a fixed workflow, the LLM is given **tools** such as semantic search or SQL execution and the freedom to decide how and when to use them.

This shifts RAG from a **workflow** to an **autonomous process.** In a workflow, the path is fixed meaning retrieve (using exact user prompt), then answer using the retrieved chunks/docs. In Agentic RAG, the model can choose a different path for each query. It may decide not to use retrieval at all, or to call it multiple times. For example:

- If the task is to compare two tables, the model can call the search tool twice, once for each table, then perform the comparison.
- If the query is simple, like reformatting SQL between dialects, the model may skip retrieval entirely.
- If the first retrieval is poor, the model can adjust and try again, rather than failing silently.

This flexibility solves many of the inefficiencies of vanilla RAG. It introduces:

- **Adaptive retrieval:** Use retrieval only when it is needed.
- **Multi step reasoning:** Handle tasks that require multiple lookups or tool calls.
- **Feedback loops:** Accept human feedback at the end of a response and retry with corrections.
- **Autonomous decision making:** Take different paths for different queries instead of one size fits all.

Agentic RAG is a significant improvement, but it is not a complete solution. Even with autonomy, the model is only as good as the context it receives. If table names are misleading, if embeddings are weak, or if metadata is incomplete, the model will still hallucinate or select the wrong tables.

The next challenge is to address these weaknesses directly by **engineering the context itself**, so the model always has the most relevant and reliable information available.

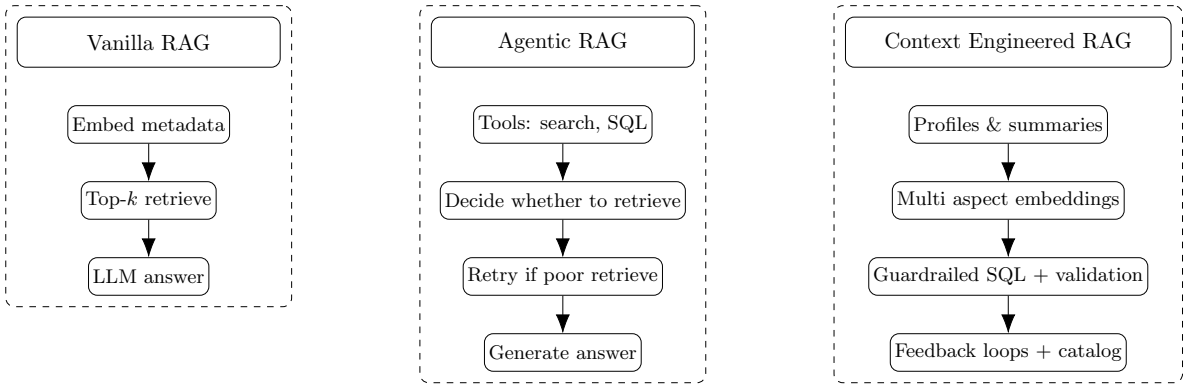

Figure 1: Comparing paradigms: Vanilla (fixed, shallow), Agentic (tool driven, flexible), and Context Engineered (structured knowledge, guardrails, feedback).

5 Remaining Issues

Even with the flexibility of Agentic RAG, key problems remain. Autonomy alone does not guarantee correctness especially if the underlying context is weak or misleading. In practice, engineers still encounter:

- **Hallucinated tables:** The model may invent a table name that sounds plausible but does not exist.
- **Misleading column names:** Without clarity, the model may use an incorrect column or use the correct column incorrectly for example, confusing region (geography) with region (business unit).
- **Null values and data quality:** If two similar tables exist, but one is sparsely populated, the model cannot distinguish which is authoritative.
- **Hallucinating Column Values:** Low cardinality columns can be queried with few fixed values ("US" instead of "USA" will not work if table has "USA"). On other hand for high

cardinality columns there might be a fixed pattern to it like user id's in that column following pattern: "user_{id}".

- **Missing catalogs:** Many enterprises lack up to date catalogs, so the model operates with incomplete metadata.
- **Weak embeddings:** Table and column names, along with numeric stats, are poor inputs for semantic search. Even with advanced stats like null ratios or distinct counts, embeddings struggle to capture meaningful relationships.

These gaps mean that Agentic RAG often produces results that look correct but are fundamentally wrong. For engineers, this translates into wasted verification cycles running queries, checking against dashboards, and iterating manually until accuracy is acceptable.

To move beyond this, we need more than just autonomy. We need a way to engineer the context itself, ensuring that what the model sees is accurate, detailed, and structured for retrieval. This is where our approach begins.

6 Our Approach

We improve upon Agentic RAG by applying **Context Engineering**: not just giving the model tools, but deliberately building and passing the most relevant context into its context window. This ensures the model can operate with accurate, detailed knowledge rather than guessing from incomplete metadata.

Our approach consists of four main components: Metadata Extraction, Intelligent Sampling, Profiling, and LLM Based Summarization.

6.1 Metadata Extraction

The foundation of our system is a rich metadata layer that goes beyond table names.

- **Table level stats:** schema name, table name, row count, number of columns, and index columns.
- **Column level stats:** column name, type, null percentage, distinct counts, and representative values.
- **Value distributions:** for low cardinality columns we store all unique values; for high cardinality columns we keep 3–5 representative values.

This gives the LLM exact signals about how data is represented. For example, if the user query asks for "United States," the system knows whether to filter by "US", "USA", or "United States" based on actual sampled values.

6.2 Intelligent Sampling

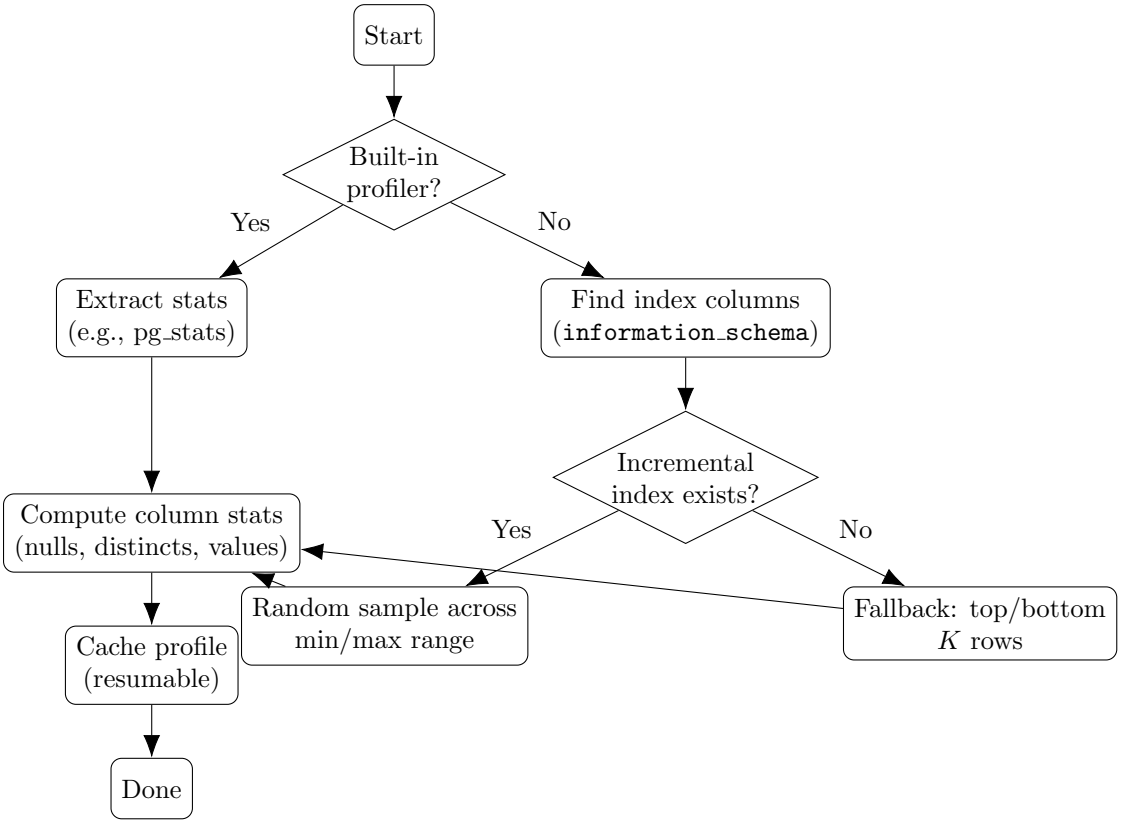

Figure 2: Intelligent sampling: use built-in stats when available; otherwise, prefer random sampling via incremental indices, with a safe top/bottom fallback.

For databases with built-in profilers (e.g., PostgreSQL via `pg_stats`), we directly extract stats. For databases without them (e.g., MariaDB), we build our own sampling mechanism to ensure accuracy and efficiency:

- **Filter deprecated tables:** Exclude tables not updated in over a year, reducing noise.
- **Detect index columns:** Using `information_schema`, identify fast queryable columns.
- **Check incremental patterns:** If a column has values like `user_1`, `user_2`, ... or numeric ranges, confirm by probing random values between min and max.
- **Random sampling:** If an incremental index column is found, generate random values across the full range and sample up to $\min(10{,}000, row\_count)$.
- **Fallback sampling:** If no incremental index column exists, take the top and bottom 5k rows to approximate distribution.

This ensures our sampled dataset preserves real world distributions instead of just the first few rows, which are often biased.

6.3 Profiling

With sampled data in hand, we compute statistics (e.g., in pandas):

- Null counts and ratios for each column.
- Distinct counts to measure cardinality.
- Most common values with frequencies.
- Unique value sets for low cardinality columns (< 20 values).
- Representative values for high cardinality columns.

Profiler outputs are stored locally per table, enabling resumability. If the system crashes or new tables are added, we only reprocess what is missing.

6.4 LLM Based Summarization

Metadata and stats are not semantically rich enough for embeddings. To solve this, we use the profiler outputs to prompt an LLM to generate structured summaries for each table. This concept follows a similar idea to Anthropic's Contextual Retrieval [5]. Each summary covers multiple aspects:

- **Summary**: a high level description of the table.
- **Purpose & granularity**: whether the table is fact/dimension, and what unit of data it represents.
- **When to use**: typical business questions and scenarios where the table is relevant.
- **Column descriptions**: one line natural language descriptions per column.

This summary transforms numeric stats into semantic knowledge that both enriches catalogs and provides meaningful content for embedding. Asking the LLM to cover multiple aspects (not just a single paragraph) helps create semantically rich embeddings that incorporate multiple factors.

6.5 Context Injection

At query time, the agent has access to:

- **Semantic search tool:** retrieves relevant summaries and stats.
- **SQL tool:** executes only `SELECT` queries (guardrails prevent `UPDATE`/`DELETE`).
- **Web search (optional):** pulls in external definitions or references.
- **System context:** list of all table names for global awareness.

This ensures that the LLM's context window is engineered not overloaded, not missing critical details. The agent sees only what is necessary and relevant for reasoning.

By combining metadata extraction, intelligent sampling, profiler stats, LLM based summaries, and multi aspect embeddings, our approach transforms Agentic RAG into a reliable Context Engineered RAG. The result is fewer hallucinations, faster query generation, and more accurate results aligned with business definitions.

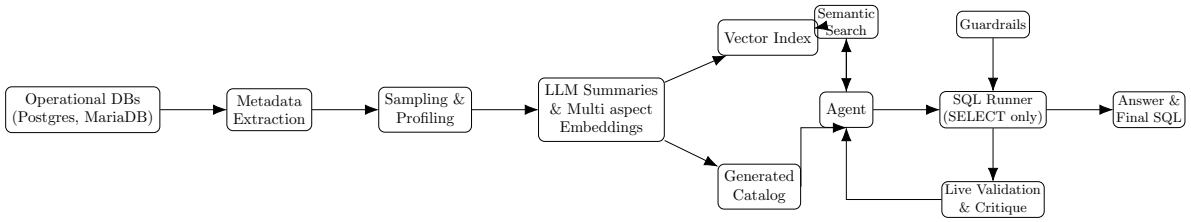

Figure 3: End to end Context Engineered Agentic RAG pipeline: metadata → profiling → summaries/embeddings; agent uses semantic search, guardrailed SQL, and validation/feedback; catalog is generated as a byproduct.

7 Context Engineering

Agentic RAG gives the model tools and autonomy, but autonomy without the right context still fails. The real improvement comes from **Context Engineering**: deliberately designing what goes into the model's context window so that it always has the right information at the right time. It optimizes by removing any unnecessary information, while adding all possible useful information for the task.

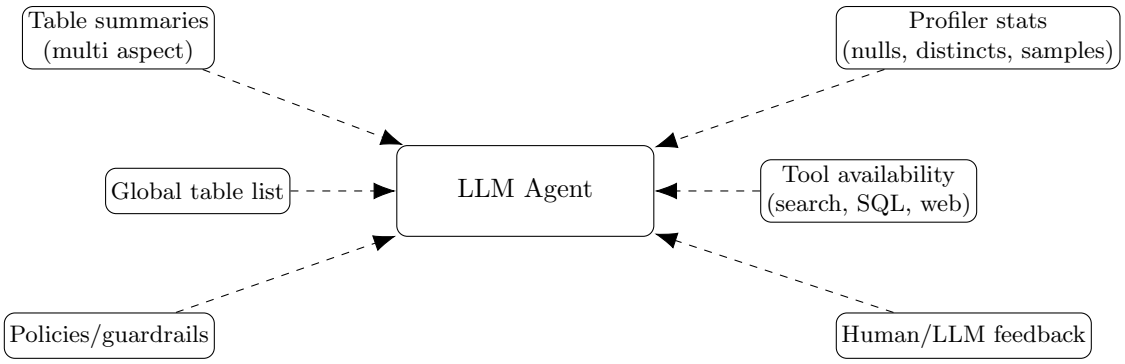

Figure 4: Context Engineering: curated inputs injected into the agent's context window. Dashed arrows indicate non destructive, read only context.

Instead of hoping the model guesses correctly, we provide it with a curated set of context sources:

7.1 Semantic Search Tool

The agent can call semantic search on demand, but unlike vanilla RAG, the results come from multi aspect embeddings of LLM generated summaries, not just raw column names or numeric

stats. This increases the chance that a vague or shorthand user query/prompt will still surface the correct tables.

7.2 SQL Query Tool with Guardrails

We allow the model to execute SQL directly, but only under strict controls. Every generated query passes through a policy check before execution:

- **Allowed:** `SELECT` queries.
- **Blocked:** `UPDATE`, `DELETE`, `DROP`, or any schema altering operation.

If the model attempts a disallowed query, the guardrail intercepts it and feeds back a correction. This prevents accidental or unsafe operations while still enabling live query validation.

7.3 Web Search Tool

For cases where company specific metadata is insufficient such as definitions of churn, revenue, or regulatory terms, the agent can query external sources. This augments local schema knowledge with domain knowledge, giving it the same context a human engineer would bring in from external documentation.

7.4 System Context

Alongside tools, we provide a global view of all table names in the system prompt. This ensures the model knows what tables exist, even if retrieval tools are not used. In our case, table counts were manageable, so listing them did not overwhelm the context window.

7.5 Why Context Engineering Matters

The difference between "vibe coding" and "context engineering" is precision.

- In vibe coding, the model generates based on incomplete or noisy context, leading to trial and error.
- In context engineering, we structure and inject the exact context needed: metadata, summaries, sample values, table lists, and guardrails.

This shift transforms performance. With all relevant context in place, the model no longer wastes cycles guessing. It retrieves efficiently, validates with live queries, and iterates with human or LLM critique when needed.

The result is an agent that reasons like an experienced engineer: it knows the schema, it understands business use cases, and it avoids dangerous mistakes.

8 Results

We evaluated our system on real engineering workflows, comparing it against the baseline of manual query building and against Databricks Genie. The results show significant improvements in both speed and accuracy.

8.1 Time Savings

Traditionally, building and verifying a single metric took ∼2 days of engineer time. Most of this effort was spent discovering relevant tables, validating assumptions, and iterating on queries. With our system, the same process can be completed in ∼5 minutes per metric. The LLM retrieves the correct tables, generates a candidate query, and validates it against the profiler outputs or live SQL. Engineers then review and provide feedback, iterating only if needed.

8.2 Accuracy

Databricks Genie achieves around 40% accuracy in generating correct queries in our testing. Our improved Agentic RAG with Context Engineering consistently reached ∼75% accuracy. The improvement comes from:

- Richer metadata (profiling, summaries, embeddings).
- Multi-step retrieval (agent calls search multiple times when needed).
- Guardrails preventing invalid SQL.
- Human in the loop feedback enabling quick correction.

Even when the first query is not perfect, the model typically converges to a correct solution within 2–3 iterations.

8.3 Feedback Loops

Unlike naive or vanilla RAG, our system is iterative by design. Engineers can reject a query and specify corrections, ask the agent to try a different table, or validate outputs against dashboards or ground truth. The agent incorporates this feedback, regenerates queries, and improves accuracy on the next attempt. This loop significantly reduces wasted effort compared to starting from scratch.

8.4 Practical Impact

The real value is not just accuracy but efficiency of discovery. Even when queries require manual adjustment, engineers now start from a strong baseline with the right tables and columns identified. This shifts the workload from hours of searching to minutes of refinement, freeing engineers to focus on business logic rather than schema archaeology.

8.5 Automated Cataloging

A secondary benefit of our approach is that it automatically generates a usable data catalog as a byproduct of profiling and summarization.

- Each table is enriched with human readable descriptions, usage guidance, and column level explanations.
- Low cardinality values and representative samples are recorded, making it easier for analysts to understand valid inputs.

- Multi aspect embeddings double as both search indexes and catalog entries.

This means that even teams without an existing catalog gain one "for free" through the system. Unlike traditional cataloging efforts which require manual documentation and quickly become outdated, our catalog is machine generated, consistent, and refreshable.

9 Limitations

While our approach improves both speed and accuracy, it is not without constraints. Some challenges remain open and must be acknowledged.

9.1 Catalog Freshness

Our profiler captures snapshots of table structure and statistics. If profiling is not refreshed regularly, the system may rely on outdated information. Tables may have grown, changed column distributions, or been deprecated since the last run. This can lead to misleading context and inaccurate queries.

9.2 Table Disambiguation

Enterprises often have multiple tables with nearly identical names and columns for example, `orders`, `orders_v2`, `orders_archive`. Without explicit signals about which tables are authoritative and which are legacy, the model may choose incorrectly. Human feedback mitigates this, but the ambiguity remains a source of error.

9.3 Resource Overhead

Sampling and profiling large tables, especially in systems without built-in statistics, requires compute and storage. Although we cache and resume profiles, there is still overhead in extracting and maintaining this metadata at scale.

9.4 Dependency on Feedback

The system relies on feedback loops to converge to correct queries. While this is faster than manual discovery, it is not always one shot. In cases where business definitions are unclear or undocumented, accuracy still depends on iterative human correction.

10 Future Work

While our system shows strong improvements over vanilla and agentic RAG, there are several directions to extend its capabilities and reduce the remaining limitations.

10.1 Retrieval Time Augmentation

We plan to enrich retrieval results with similar table context. For every table in the index, we will pre compute links to related tables based on embedding similarity, and add statistics such as read/write frequency, column null ratios, and creation dates for the linked tables. At query time,

when one table is retrieved, the system can surface its closest alternatives, giving the agent more options to disambiguate between active and deprecated tables.

10.2 SQL Script Integration

In many organizations, tables are created and maintained through SQL scripts. By parsing and embedding these scripts alongside profiler metadata, the system can capture intent and lineage knowledge that is often lost once a table exists in production. This will allow the LLM to reason not just about what the table contains, but *why* it was designed that way.

10.3 Continuous Profiling

To address catalog freshness, we aim to implement incremental profiling pipelines. Instead of full re profiling runs, we will detect schema changes or row count deltas and only reprocess affected tables. This will keep metadata up to date while reducing compute overhead.

10.4 Domain Aware Embeddings

Current embeddings are general purpose and struggle with domain specific nuances. Training or fine tuning embeddings on company specific queries and schemas could improve precision, especially for subtle distinctions like subscription states or revenue recognition rules.

10.5 Active Learning from Feedback

We plan to make human feedback persistent. Each time an engineer corrects the system, that correction can be stored as a training signal for both retrieval and query generation. Over time, the agent will not only adapt per query but also learn from the organization's collective corrections, reducing the need for repeated human guidance.

11 Conclusion

Engineers waste days rediscovering database context and validating queries that should take minutes. Vanilla RAG promised automation but was rigid and shallow. Agentic RAG improved flexibility but still failed on weak metadata and incomplete catalogs.

Our approach advances beyond this by combining profiling, intelligent sampling, LLM based multi aspect summaries, and context engineering. By deliberately constructing what enters the model's context window, we reduce hallucinations, improve retrieval quality, and align queries with business definitions.

The result is a system that cuts metric development from $\sim$2 days to $\sim$5 minutes and improves accuracy from $\sim$40% to $\sim$75%. It generates a living catalog as a byproduct, integrates human feedback for iterative improvement, and safeguards against unsafe operations.

Limitations remain particularly around catalog freshness, table disambiguation, and embedding precision but these can be addressed through incremental profiling, retrieval time augmentation, and domain aware learning. By engineering the context instead of leaving it implicit, we transform Agentic RAG into a practical solution for database discovery and query generation at scale.

A Appendix A. Sampler Pseudocode

```
# Identify candidate tables (last updated within 1 year)
tables = filter_recent_tables(information_schema)

for table in tables:
    index_columns = get_index_columns(table)

    # Detect incremental patterns
    incremental_candidates = []
    for col in index_columns:
        sample_rows = fetch_sample_rows(col)
        if detect_incremental_pattern(sample_rows):
            incremental_candidates.append(col)

    # Choose best incremental index
    chosen_col = select_highest_match(incremental_candidates)

    if chosen_col:
        # Random sampling across min/max range
        values = generate_random_values(
            chosen_col, n=min(10_000, row_count)
        )
        sample = fetch_rows_by_values(table, chosen_col, values)
    else:
        # Fallback: top and bottom 5k rows
        sample = fetch_top_bottom_rows(table, 5000)

    save_sample(table, sample)
```

B Appendix B. Profiler Outputs

Table Stats

- timestamp
- schema_name
- table_name
- table_comment
- row_count

Column Stats

- column_name
- data_type
- is_nullable

- null_frac
- n_distinct_ratio
- most_common_vals
- most_common_freqs

Profiler outputs are cached per table, enabling incremental refresh and crash recovery.

C Appendix C. Summary Prompt

```
def build_table_analysis_prompt(table, columns, column_statistics,
                                relationships, row_count):
    context_prompt = f"""
You are a senior database engineer...
TABLE: {table.schema}.{table.name}
Type: {getattr(table, 'type', 'Unknown')}
Row Count: {row_count or 'Unknown'}
Comment: {getattr(table, 'comment', 'No comment')}

ANALYSIS REQUIREMENTS:
1. summary
2. purpose_and_granularity
3. when_to_use_this_table
4. columns_description[]
"""

    # (Optional) Group columns by type, append sections, and relationships
       ...
    return context_prompt
```

D Appendix D. System Prompt

```
from typing import Optional

def get_system_prompt(tables_list: Optional[str] = None) -> str:
    """Get the main system prompt for the SQL agent."""
    base_prompt = """You are a SQL database expert assistant specializing
in database for analysis and query generation.

##INSERT COMPANY CONTEXT HERE##

Your capabilities include:
1. Analyzing database schemas and providing insights
2. Generating SQL queries for MariaDB/PostgreSQL
3. Explaining query logic and performance considerations
4. Recommending data exploration strategies
```

```
IMPORTANT GUIDELINES:
- Always use retrieve_database_context to get schema info
- Use each tool once, except retrieve_database_context (up to 3 times)
- Retrieve table details before SQL generation
- Write only SELECT queries unless explicitly asked otherwise
- Never produce queries that modify or delete data
- Validate that tables/columns exist before use
- Include comments in SQL
- Warn about potential performance issues
- Always ask clarifying questions before proceeding

Response Format:
- Be conversational and helpful
- Provide context for decisions
- Explain assumptions
- Offer alternatives when relevant

Available Tools:
- retrieve_database_context: retrieve top 20 matching tables
- get_table_details: detailed schema for a given table
- search_web: for external context (sparingly)
- run_sql_query: Runs and returns the results from a SQL query

Remember: you're helping users query their database effectively and safely
    .
"""

    if tables_list:
        base_prompt += (
            f"\n\n**DATABASE CONTEXT:**\n{tables_list}\n\n"
            "Use this list to understand available tables."
        )
    return base_prompt
```

References

[1] LanceDB. (n.d.). *Vanilla RAG*. In *LanceDB documentation*. Retrieved September 4, 2025, from https://lancedb.github.io/lancedb/rag/vanilla_rag/ Accessed: 2025-09-01.

[2] Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, M., & Wang, H. (2024). Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv. https://doi.org/10.48550/arXiv.2312.10997

[3] Singh, A., Ehtesham, A., Kumar, S., & Talaei Khoei, T. (2025). Agentic retrieval-augmented generation: A survey on agentic RAG. arXiv. https://doi.org/10.48550/arXiv.2501.09136

[4] Karpathy, A. [@karpathy]. (2025, August 14). I prefer "context engineering" over "prompt engineering" because it's about carefully filling the LLM's context with the right info for each task [Tweet]. X. https://x.com/karpathy/status/1937902205765607626 Accessed: 2025-09-01.

[5] Anthropic. (2024, September 19). Introducing Contextual Retrieval. Anthropic. https://www.anthropic.com/news/contextual-retrieval Accessed: 2025-09-01.