ChatGPT

Development Guide for a Grounded Natural-Language-to-SQL Python Tool Using LangChain and GitHub Copilot SDK

Executive summary

Natural-language-to-SQL (NL→SQL / Text-to-SQL) systems have progressed from early sequence-to-sequence generation toward *schema-aware*, *constraint-aware*, and *execution-aware* pipelines that are substantially more reliable in real deployments than "single-pass prompt → SQL" approaches. Key research directions include relation-aware schema encoding (schema linking at model time), constrained decoding (preventing syntactically or schema-invalid SQL during generation), execution-guided decoding (using database feedback to repair or select among candidates), and retrieval-augmented prompting using prior NL↔SQL examples and schema documentation. [1]

For a production-grade Python tool grounded on CSV schema files and a library of sample SQL queries + metadata, the most robust design is a **multi-stage, schema-grounded workflow**:

1) ingest and normalize schema + example-query corpora,
2) retrieve a *schema slice* (tables/columns/relationships) plus *few-shot query exemplars* relevant to the user question,
3) generate SQL in a **structured, validated format** (plan/AST-first or strict JSON schema),
4) compile to SQL, validate (dialect + schema + safety constraints),
5) execute in a bounded environment and optionally run an **execution-guided repair loop**. [2]

LangChain provides "off-the-shelf" SQL agents/chains (e.g., SQLDatabaseChain / SQLDatabaseToolkit / create_sql_agent) and the ecosystem components needed to build the above pipeline (retrievers, prompt templates, structured output parsers, callbacks, graph workflows). [3] The GitHub Copilot SDK (technical preview) can supply the LLM layer and/or an agent runtime via the Copilot CLI; it communicates with Copilot CLI over JSON-RPC and supports multiple languages including Python. [4]

Because multiple operational details are **unspecified** (DB engine/dialect, scale of schemas, latency targets, deployment environment), this guide presents implementation options and explicitly flags where environment-specific choices matter.

State of the art in NL-to-SQL and what it implies for your tool

Core technical challenges that modern systems explicitly address

Cross-domain benchmarks like Spider were designed to test generalization to *unseen databases* (database split) and complex multi-table SQL, making clear that NL-to-SQL is not merely language modeling—it is

structured semantic parsing with grounding to a schema, and often to database contents. [5] Conversational benchmarks (e.g., CoSQL) further add multi-turn context and ellipsis/underspecification. [6] More recent benchmarks (e.g., BIRD) emphasize realism and "value grounding," including harder schema/value linking, messy data characteristics, and external knowledge needs. [7]

A practical implication: **your CSV schema + query library must be treated as a first-class knowledge source**, and your system should be evaluated on both (a) exact SQL correctness and (b) execution correctness under realistic constraints. [8]

Transformer architectures and schema-aware modeling

Modern neural NL-to-SQL systems largely adopt transformer-based encoders/decoders (e.g., T5-style text-to-text generation), but high-performing solutions incorporate **schema structure and relations** rather than concatenating raw schema text. For example, relation-aware schema encoding (e.g., RAT-SQL) explicitly models relationships among question tokens, tables, and columns to improve schema linking. [9] Alternative paradigms include **bottom-up program construction** (e.g., SmBoP) for compositionality and complex query assembly. [10]

Implication for your tool: even if you rely on LLM prompting rather than fine-tuning, you should *mimic* these ideas by: - retrieving and presenting schema **relationships** (foreign keys / join paths / semantic hints), not only table/column lists,
- prompting the model to explicitly **select relevant schema elements first** (a schema linking stage), then generate SQL using only those elements. [11]

Constrained decoding and "make invalid SQL hard/impossible"

A persistent failure mode of unconstrained generation is producing invalid or unusable SQL. Constrained decoding methods (notably PICARD) use incremental parsing during decoding to reject tokens that would lead to invalid programs, improving usability and accuracy for text-to-SQL tasks. [12] More recent work also explores "template constrained decoding" (TeCoD), leveraging repeated query patterns in workloads to improve reliability for recurring questions. [13]

Implication: in LLM application development (especially via chat APIs where true token-level constraints may not be exposed), you can approximate constrained decoding with: - **structure-first outputs** (JSON AST/plan → compiler),
- strict **output parsing + validation** (reject/repair),
- optionally generate **multiple candidates** and keep only those that parse, satisfy schema constraints, and pass safety rules. [14]

Execution-guided decoding and database-in-the-loop repair

Execution-guided decoding (EGD) uses a database engine as a verifier: candidate SQL is executed (or partially executed), and feedback informs selection or iterative repair. A canonical reference is "Robust Text-to-SQL Generation with Execution-Guided Decoding," which formalizes how execution feedback can improve correctness. [15]

In the LLM era, this idea appears as: - “generate k candidates → execute → select by consistency / majority / heuristics,”
- “generate → execute → capture error → repair prompt → retry,”
- “test-suite accuracy” metrics (execute across multiple test databases/instances) to reduce overfitting to a single database state (prominent in SQL-PaLM). [16]

Implication: your system should implement **a bounded execution-guided loop** (with timeouts, row limits, and safety constraints), and your evaluation should track both exact match and execution correctness. [17]

Few-shot prompting, chain-of-thought, retrieval augmentation, and program-synthesis approaches

Chain-of-thought prompting improves multi-step reasoning in LLMs by eliciting intermediate reasoning steps. [18] Self-consistency improves CoT by sampling diverse reasoning paths and selecting the most consistent answer. [19]

In text-to-SQL specifically, prompt engineering studies (e.g., DAIL-SQL) show that **representation choices and example selection/organization** can materially improve execution accuracy on Spider, indicating that retrieval of relevant demonstrations is a major lever. [20] Similarly, “Few-shot Text-to-SQL Translation using Structure and Content Prompt Learning” proposes a divide-and-conquer framing that separates SQL structure prediction from filling in concrete identifiers/values, and includes constrained decoding ideas. [21]

SQL-PaLM frames NL-to-SQL as an LLM adaptation problem across prompting and fine-tuning, and studies: execution-based filtering, synthetic data augmentation, and *query-specific database content* as additional grounding signals—illustrating the breadth of “prompting vs fine-tuning” trade-offs. [16]

Implication: your tool should treat the sample-query library as a **retrieval corpus for demonstrations**, and it may also treat “common query templates” as constraints (especially for recurring enterprise analytics questions). [22]

Fine-tuning vs prompting vs synthetic data

Research trends increasingly use: - **prompt-only** pipelines (fast to iterate; depends on model strength and good retrieval),
- **instruction fine-tuning** (improves controllability and schema linking; needs data and MLOps),
- **synthetic data generation** to scale supervision (e.g., multi-million synthetic datasets like SynSQL-2.5M) to improve generalization and coverage. [23]

Implication: start with prompt + retrieval + validation (lowest operational overhead), and add fine-tuning only if (a) workloads are stable, (b) you can curate/validate enough training data, and (c) latency/cost or reliability needs justify it.

Evaluation metrics and robustness tests you should plan for

Standard evaluation includes: - **Exact Match (EM)**: canonicalized SQL string structure match (common in Spider-style eval), [5]
- **Execution Accuracy**: whether the generated SQL produces the correct result when executed (used

prominently in LLM-era text-to-SQL evaluations like DAIL-SQL), [20]
- **Test-suite accuracy** (execute across multiple database instances / hidden tests), emphasized in SQL-PaLM to reduce "spurious correctness." [16]

Robustness testing is increasingly important because annotation ambiguity and dataset errors can distort perceived progress; recent analysis argues that pervasive annotation errors can "break" text-to-SQL benchmarks, underscoring the need for careful evaluation design and error analysis. [24] A practical response is to test not only on public benchmarks but also on: - schema perturbations (column renames/ synonyms),
- paraphrase/typos,
- adversarial values,
- real production query logs and known-good SQL templates. [25]

Mapping NL-to-SQL techniques to LangChain building blocks

LangChain's SQL agent tutorial describes a canonical multi-step flow: list tables, choose relevant tables, fetch their schemas, generate SQL, and double-check for common mistakes. [26] This aligns well with the research direction of "schema linking → SQL generation → verification," but you'll likely want to make the steps *more deterministic and auditable* than a fully autonomous agent for production analytics.

Key LangChain components you can assemble into a robust NL-to-SQL pipeline

LangChain offers: - **SQLDatabaseChain** (experimental): a chain for interacting with SQL databases; LangChain explicitly warns you to use narrowly scoped DB credentials because the chain may attempt destructive commands if prompted. [27]
- **SQLDatabaseToolkit**: toolkit for SQL databases with a DB connection and an LLM (used by query checking tools). [28]
- **create_sql_agent**: constructs a SQL agent from an LLM and toolkit/db; the reference notes that "tool-calling" agent types require an LLM supporting tool calling. [29]
- **Structured output and output parsers**: to force predictable JSON/Pydantic outputs instead of brittle text parsing. [30]
- **Retrievers** (BM25 and vector-store-backed retrievers) to implement hybrid retrieval over schemas/ examples. [31]
- **Callbacks + tracing** for observability and debugging. [32]
- **LangGraph workflows** for deterministic orchestration, streaming, loops, and "agentic" flows with explicit states. [33]
- **Runnables** interface (invoke/batch/stream + async variants) and built-in batching behavior that can reduce latency for I/O-bound pipelines. [34]
- **LLM caching** via global cache configuration (e.g., `set_llm_cache`). [35]

Practical mapping table: innovations → LangChain implementation patterns

Innovation area	Why it matters	LangChain-oriented implementation pattern	Notes / pitfalls
Schema linking	Prevents wrong tables/columns	Retrieval stage returning "schema slice" + prompt stage that *must* output selected tables/columns	Use metadata filters so the model cannot "invent" objects
Constrained decoding	Reduces invalid SQL	Structured output (SQL plan/AST), strict parser, schema validator, retry-on-parse-fail	Token-level constraints may be unavailable; rely on validation + repair [36]
Execution guidance	Repairs semantic errors	DB execution tool + retry loop (LangGraph conditional edges)	Add query timeouts/row limits, and isolate read-only roles [37]
Few-shot prompting	Captures styles/ templates	Retrieve top-k similar NL↔SQL exemplars; insert into prompt template	DAIL-SQL emphasizes example selection/ organization effects [20]
Retrieval augmentation	Grounds to local knowledge	Vector store + BM25 hybrid retrieval over schema docs and query corpus	Hybrid retrieval is often more robust than dense-only for identifiers [38]
Observability	Debug + compliance	Callbacks + structured logs + tracing (LangSmith or alternative)	Ensure logs redact sensitive values [32]

Grounding schemas and sample SQL with CSV-centered retrieval

This section provides a concrete, CSV-driven grounding strategy compatible with NL-to-SQL research practices (schema linking, exemplar retrieval) and with LangChain's retrieval abstractions.

Recommended CSV formats for schemas and sample queries

Because the exact schema CSV format is unspecified, prefer a **normalized, long-form schema table** rather than wide, human-oriented sheets. A robust minimum schema CSV contains one row per column:

schema_columns.csv (recommended minimum)
- `db_id` (string)
- `table_name` (string)
- `column_name` (string)
- `data_type` (string)
- `is_primary_key` (bool)
- `is_foreign_key` (bool)
- `references_table` (string nullable)
- `references_column` (string nullable)
- `column_description` (string nullable)

- `value_examples` (string nullable; safe samples only)
- `synonyms` (string nullable; comma-separated)

Add a second file for table-level metadata:

schema_tables.csv (recommended)
- `db_id`, `table_name`, `table_description`, `row_count_estimate`, `update_frequency`, `pii_category` (if relevant)

For "sample SQL queries with metadata," separate the SQL from descriptive fields:

sample_queries.csv (recommended)
- `db_id`
- `query_id`
- `nl_question` (human paraphrase(s) if available)
- `sql`
- `sql_dialect` (if multi-dialect)
- `tables_used` (list-like string)
- `columns_used` (list-like string)
- `tags` (e.g., "retention", "revenue", "cohort")
- `complexity` (simple/medium/complex)
- `description` (natural language explanation)
- `verified` (bool; human-validated)
- `source` (e.g., "handcrafted", "from logs", "synthetic")

These formats make it easy to build **schema linking**, **exemplar retrieval**, and **policy-driven filtering** (e.g., don't retrieve samples touching sensitive tables unless permitted). Value-grounding and realism are emphasized in newer benchmarks like BIRD, so capturing safe "value examples" and semantics is a major quality lever. [7]

Indexing strategy: schema-aware corpora rather than "one big blob"

Avoid indexing raw CSV rows without structure. Instead, create **multiple document types**, each with stable IDs and metadata:

- `doc_type="table"`: one document per table (description + column summaries + join hints)
- `doc_type="column"`: one document per column (name, type, description, synonyms, sample values)
- `doc_type="relationship"`: one document per FK relationship (A.col → B.col, plus semantic label like "customer_id")
- `doc_type="example_query"`: one document per verified example query (NL + SQL + explanation + tags)

This avoids over-chunking and supports metadata filtering (e.g., "only return schema for db_id=X"). LangChain retrievers operate over `Document` objects with metadata, and vector stores like Chroma explicitly support storing documents plus metadata in collections, enabling filtering and retrieval. [39]

Embedding and retrieval strategies tuned for schema + SQL workloads

A typical Text-to-SQL retrieval workload is *identifier-heavy* ("customer_id", "acct_bal", "order_date") and benefits from **hybrid retrieval** (lexical + semantic). Weaviate's documentation describes hybrid search as combining vector search with BM25-style keyword search and fusing result sets with configurable weighting. [40] Qdrant similarly describes hybrid approaches combining sparse and dense signals. [41] LangChain provides a BM25 retriever integration (sparse retrieval). [31]

Practical recommendations: - Use **dense embeddings** for semantic matching (descriptions, "what does this table mean?").
- Use **BM25** (or other sparse) for literal identifiers (exact column/table names, acronyms). [42]
- Prefer **hybrid fusion** or query-time ensemble if your environment supports it. [43]
- For schema and SQL, avoid aggressive chunking; table/column docs are usually already short enough.

If you want "schema-aware retrieval" beyond plain similarity, use metadata and lightweight symbolic re-ranking: - Increase scores when retrieved docs reference tables/columns already selected in schema linking.
- Boost documents whose `db_id` matches the active database.
- Optionally re-rank by join-path proximity (graph distance), approximating relation-aware encoding ideas from systems like RAT-SQL. [9]

Vector store choices and trade-offs

Below is a pragmatic comparison of popular vector store backends; the "best" choice depends on your scale and operational posture (unspecified).

Vector store	Strengths	Potential drawbacks	When to choose
Chroma	Developer-friendly "collection" abstraction storing embeddings + docs + metadata; easy local/dev usage [39]	Operational maturity depends on deployment mode; plan persistence/ backup explicitly	Local dev, single-node deployments, early prototyping
Weaviate	First-class hybrid search combining vector + BM25-style keyword search with tunable fusion [40]	Operational overhead (service) unless managed; schema/config learning curve	When hybrid retrieval must be first-class and scalable
Qdrant	Strong hybrid story via sparse+dense approaches; explicit discussion of sparse vectors + hybrid patterns [41]	Hybrid may require query/ API familiarity; operational overhead unless managed	When you need scalable vector search + hybrid experimentation

Vector store	Strengths	Potential drawbacks	When to choose
pgvector	Store vectors inside Postgres; keeps embeddings with relational data and leverages Postgres features (ACID/joins) [44]	Performance/tuning depends on Postgres setup; hybrid lexical search requires extra components	When your org already runs Postgres and wants fewer moving parts

Reference architecture integrating Copilot SDK with LangChain

Interface and runtime assumptions

- **Unspecified database engine/dialect**: This affects SQL syntax, quoting rules, and validation tooling. Consider supporting a single dialect first (e.g., PostgreSQL or SQLite) and expanding later. [45]
- **Unspecified production scale/latency**: This affects caching, batching, and streaming strategy.
- **LLM access via Copilot SDK**: The Copilot SDK communicates with Copilot CLI server mode via JSON-RPC, and requires the Copilot CLI to be installed. [4]
- **SDK maturity**: Copilot SDK is described as "Technical Preview." [46]

Authentication and model selection with Copilot SDK

The Copilot SDK documentation states: - SDKs exist for Python/TypeScript/Go/.NET, installed via package managers, [46]
- the SDK supports multiple authentication methods, and supports BYOK (bring your own key) to use provider keys; it explicitly notes BYOK uses key-based auth only and does not support Microsoft Entra ID / managed identities. [47]
- "All models available via Copilot CLI are supported" and the SDK can list available models at runtime. [46]
- by default, the SDK operates the CLI in an allow-all tools mode (equivalent of `--allow-all`), but tool availability can be customized—important for limiting capabilities in an NL→SQL product. [46]

This implies a recommended security posture: **disable file system / shell / web tools** unless your product explicitly needs them, and expose only tightly scoped DB-related tools.

Architecture diagram

```
flowchart LR
  user[User / API Client] --> app[NL→SQL Service API]
  app --> orch[LangChain Orchestrator / LangGraph Workflow]
  orch --> retr[(Schema + Query Retrievers)]
  retr --> vs[(Vector Store)]
  retr --> bm25[(BM25 Index)]
  orch --> llm[LLM via Copilot SDK (Copilot CLI JSON-RPC)]
  orch --> validator[SQL Validator + Policy Guard]
  validator --> db[(Database - Read-only Role)]
  db --> validator
```

```
validator --> orch
orch --> resp[Response: SQL + Results + Explanation + Audit]
```

The Copilot SDK roll-up docs describe the architecture as “your application → SDK client → Copilot CLI (server mode) via JSON-RPC,” which matches the “LLM via Copilot SDK” box above. [46]

Deterministic workflow vs “agent that figures it out”

For database querying products, prefer a **workflow** over an open-ended agent, because: - workflows have predetermined paths, while agents decide their own processes and tool usage; LangGraph documentation explicitly distinguishes these patterns and emphasizes benefits like persistence, streaming, and debugging. [33]
- SQL agents may hallucinate tables/columns or attempt unsafe operations unless tightly constrained; LangChain explicitly warns to scope DB credentials narrowly to prevent data corruption/loss. [27]

Recommended pattern: use LangGraph (or LCEL) to implement a state machine with explicit nodes: 1) Retrieve candidate schema slice + exemplars
2) Schema-link (select tables/columns)
3) Draft SQL (structured output)
4) Validate SQL (parse + schema + policy)
5) Execute + postprocess
6) Repair loop if needed (bounded retries)

Latency, batching, caching, and streaming

LangChain runnables support `invoke/batch/stream` (and async variants) and note that default `batch` runs invoke calls in parallel via a thread pool; this can help when your workflow includes multiple retrieval calls and/or parallel candidate generation. [34] LangChain also supports global LLM caching via `set_llm_cache`. [35]

Copilot SDK supports real-time streaming (per its SDK materials and GitHub announcements about “real-time streaming”). [48] If you need end-user streaming, combine: - Copilot SDK streaming events →
- LangGraph streaming (for node updates) →
- your API streaming mechanism (SSE/WebSocket).

Rate limiting: GitHub documents that rate limits exist to protect the service and may be impacted by authentication and resource use; you should implement retries/backoff and identify which endpoints you will call heavily. [49]

Safety and hallucination mitigation layers

A robust NL→SQL tool should enforce multiple safety layers:

- **DB credentials and permissions**: use a read-only role and restrict accessible schemas; LangChain warns that permissive credentials can lead to destructive commands. [27]

- **Prompt-level constraints**: require SELECT-only SQL by default, ban DDL/DML unless a privileged mode is enabled.
- **Validator-level constraints**: parse SQL, confirm only allowed tables/columns, add a default `LIMIT`, enforce query timeouts.
- **Execution-guided correction**: use DB error messages as feedback (bounded retries), aligning with execution-guided decoding. [15]
- **Audit logging**: retain generated SQL + metadata for review, and integrate user feedback.

For data handling concerns: GitHub documentation indicates that (for individual subscribers) GitHub and affiliates "will not use your data, including prompts, suggestions, and code snippets, for AI model training," reflected in personal settings. [50] For enterprise adoption, consult the official "how Copilot handles data" guidance, which documents data pipeline and lifecycle considerations. [51]

Implementation playbook, evaluation, and operations

This section provides code-oriented patterns. The code is designed to be adapted because runtime details are unspecified (DB dialect, embeddings provider, deployment environment).

Query processing flowchart

```
flowchart TD
  A[User question] --> B[Retrieve schema slice + exemplars]
  B --> C[Schema linking: select tables/columns]
  C --> D[Generate SQL plan/JSON]
  D --> E[Compile to SQL]
  E --> F[Validate: parse + schema + policy]
  F -->|valid| G[Execute SQL (read-only, timeout)]
  G -->|success| H[Return SQL + results + explanation]
  G -->|error| I[Repair prompt with error + constraints]
  I --> D
  F -->|invalid| I
  I -->|max retries| J[Fail safely + ask clarifying question / fallback]
```

This "generate → validate → execute → repair" loop follows the core idea of execution-guided decoding while adding modern structured output validation. [52]

Code: ingest schema and sample queries from CSV into LangChain Documents

```
from __future__ import annotations

from dataclasses import dataclass
from typing import Any, Dict, Iterable, List, Optional

import pandas as pd
from langchain_core.documents import Document
```

```python
@dataclass(frozen=True)
class SchemaDocConfig:
    db_id: str
    schema_columns_csv: str
    schema_tables_csv: Optional[str] = None
    sample_queries_csv: Optional[str] = None


def _clean_str(x: Any) -> str:
    if x is None or (isinstance(x, float) and pd.isna(x)):
        return ""
    return str(x).strip()


def load_schema_documents(cfg: SchemaDocConfig) -> List[Document]:
    """
    Load schema CSV files and (optionally) sample queries into LangChain
Documents.
    Documents are typed via metadata['doc_type'] for schema-aware retrieval.
    """
    docs: List[Document] = []

    # 1) Columns
    col_df = pd.read_csv(cfg.schema_columns_csv)
    col_df = col_df[col_df["db_id"] == cfg.db_id].copy()

    for _, r in col_df.iterrows():
        table = _clean_str(r.get("table_name"))
        col = _clean_str(r.get("column_name"))
        dtype = _clean_str(r.get("data_type"))
        desc = _clean_str(r.get("column_description"))
        synonyms = _clean_str(r.get("synonyms"))
        examples = _clean_str(r.get("value_examples"))

        text = "\n".join([
            f"db_id: {cfg.db_id}",
            f"table: {table}",
            f"column: {col}",
            f"type: {dtype}",
            f"description: {desc}",
            f"synonyms: {synonyms}",
            f"value_examples: {examples}",
        ]).strip()

        docs.append(
            Document(
```

```
                page_content=text,
                metadata={
                    "doc_type": "column",
                    "db_id": cfg.db_id,
                    "table_name": table,
                    "column_name": col,
                    "data_type": dtype,
                    "is_primary_key": bool(r.get("is_primary_key", False)),
                    "is_foreign_key": bool(r.get("is_foreign_key", False)),
                    "references_table": _clean_str(r.get("references_table")),
                    "references_column": _clean_str(r.get("references_column")),
                },
            )
        )

    # 2) Tables (optional)
    if cfg.schema_tables_csv:
        tbl_df = pd.read_csv(cfg.schema_tables_csv)
        tbl_df = tbl_df[tbl_df["db_id"] == cfg.db_id].copy()

        # Precompute columns per table for richer table docs
        cols_by_table: Dict[str, List[str]] = {}
        for _, r in col_df.iterrows():
            t = _clean_str(r.get("table_name"))
            c = _clean_str(r.get("column_name"))
            cols_by_table.setdefault(t, []).append(c)

        for _, r in tbl_df.iterrows():
            table = _clean_str(r.get("table_name"))
            desc = _clean_str(r.get("table_description"))
            cols = cols_by_table.get(table, [])
            cols_preview = ", ".join(cols[:40]) + (" ..." if len(cols) > 40
else "")

            text = "\n".join([
                f"db_id: {cfg.db_id}",
                f"table: {table}",
                f"description: {desc}",
                f"columns: {cols_preview}",
            ]).strip()

            docs.append(
                Document(
                    page_content=text,
                    metadata={
                        "doc_type": "table",
                        "db_id": cfg.db_id,
                        "table_name": table,
```

```
                        "pii_category": _clean_str(r.get("pii_category")),
                        "row_count_estimate":
_clean_str(r.get("row_count_estimate")),
                        "update_frequency":
_clean_str(r.get("update_frequency")),
                    },
                )
            )

    # 3) Example queries (optional)
    if cfg.sample_queries_csv:
        q_df = pd.read_csv(cfg.sample_queries_csv)
        q_df = q_df[q_df["db_id"] == cfg.db_id].copy()

        for _, r in q_df.iterrows():
            query_id = _clean_str(r.get("query_id"))
            nl = _clean_str(r.get("nl_question"))
            sql = _clean_str(r.get("sql"))
            tags = _clean_str(r.get("tags"))
            complexity = _clean_str(r.get("complexity"))
            verified = bool(r.get("verified", False))

            text = "\n".join([
                f"db_id: {cfg.db_id}",
                f"query_id: {query_id}",
                f"nl_question: {nl}",
                f"sql: {sql}",
                f"tags: {tags}",
                f"complexity: {complexity}",
                f"verified: {verified}",
            ]).strip()

            docs.append(
                Document(
                    page_content=text,
                    metadata={
                        "doc_type": "example_query",
                        "db_id": cfg.db_id,
                        "query_id": query_id,
                        "tags": tags,
                        "complexity": complexity,
                        "verified": verified,
                        "sql_dialect": _clean_str(r.get("sql_dialect")),
                    },
                )
            )

    return docs
```

This ingestion strategy supports: - schema-aware retrieval via `doc_type` and `db_id` filters, - demonstration retrieval for few-shot prompting (DAIL-SQL-style example selection), [20]
- future extension to pattern libraries (TeCoD-style recurring templates). [13]

Code: vector store setup and a hybrid retriever (dense + BM25)

```
from typing import List

from langchain_community.vectorstores import Chroma
from langchain_community.retrievers import BM25Retriever
from langchain.retrievers import EnsembleRetriever  # commonly used import path
in LangChain setups


def build_retrievers(
    docs: List[Document],
    embeddings,  # any langchain Embeddings implementation
    chroma_dir: str,
    collection_name: str = "nl2sql_grounding",
):
    # Dense retriever
    vs = Chroma.from_documents(
        documents=docs,
        embedding=embeddings,
        persist_directory=chroma_dir,
        collection_name=collection_name,
    )
    dense = vs.as_retriever(search_kwargs={"k": 8})

    # Sparse retriever (BM25)
    sparse = BM25Retriever.from_documents(docs)
    sparse.k = 8

    # Hybrid fusion (RRF-like) using EnsembleRetriever
    hybrid = EnsembleRetriever(retrievers=[sparse, dense], weights=[0.45, 0.55])

    return vs, dense, sparse, hybrid
```

Rationale: - Hybrid retrieval is often superior for schema/identifier-heavy corpora, and Weaviate explicitly documents hybrid search combining BM25-like and vector search. [40]
- LangChain provides BM25 retriever integration and retrieval abstractions. [53]
- Chroma “collections” store embeddings + documents + metadata and enable retrieval plus filtering. [39]

If you choose Weaviate/Qdrant instead of Chroma, implement hybrid search natively at the vector DB layer using their official hybrid mechanisms. [43]

Prompt templates: schema linking + constrained generation

A modern pattern is "schema linking first" (select tables/columns), then SQL generation restricted to those selections. This mirrors the explicit schema linking emphasis in systems like RAT-SQL and many LLM-era pipelines. [54]

Schema linking prompt (structured output)

```
from pydantic import BaseModel, Field
from langchain_core.prompts import ChatPromptTemplate


class SchemaSelection(BaseModel):
    tables: List[str] = Field(...,
description="Selected table names, exactly as in schema.")
    columns: List[str] = Field(..., description="Selected columns, as
'table.column' with exact names.")
    rationale: str = Field(..., description="Short rationale to justify
selection.")
    open_questions: List[str] = Field(default_factory=list,
description="Clarifications needed from user.")


schema_link_prompt = ChatPromptTemplate.from_messages([
    ("system",

"You are a database expert. Your task is schema linking: select the minimal set
of tables "
     "and columns needed to answer the user question.\n"
     "Rules:\n"
     "- Use ONLY identifiers that appear in the provided schema context.\n"
     "- If the user request is ambiguous, list open_questions.\n"
     "- Do not generate SQL in this step.\n"
     "- Output must match the SchemaSelection JSON schema."),
    ("human",
     "User question:\n{question}\n\n"
     "Schema context (retrieved):\n{schema_context}\n")
])
```

Use LangChain "structured output" features to enforce the Pydantic schema at generation time. [55]

SQL generation prompt (constrained by selection + exemplars)

```
class SQLDraft(BaseModel):
    sql: str = Field(..., description="SQL query string in the target dialect.")
    dialect: str = Field(..., description="Dialect name (e.g., postgresql,
```

```
sqlite).")
    safety_notes: str = Field(..., description="Explain any safety limits
(LIMIT, filters) you applied.")
    assumptions: List[str] = Field(default_factory=list,
description="Assumptions made if user was ambiguous.")


sql_prompt = ChatPromptTemplate.from_messages([
    ("system",
     "You are an expert Text-to-SQL system.\n"
     "Generate a single SELECT-only query.\n"
     "Hard constraints:\n"
     "- Use ONLY the tables/columns provided in the schema selection.\n"
     "- No DDL/DML (no CREATE/DROP/INSERT/UPDATE/DELETE).\n"
     "- Add a LIMIT unless the question explicitly requires full results.\n"
     "- Prefer explicit JOIN conditions using known foreign keys.\n"
     "- Output must match the SQLDraft JSON schema."),
    ("human",
     "Target dialect: {dialect}\n\n"
     "User question:\n{question}\n\n"
     "Schema selection (tables/columns):\n{schema_selection_json}\n\n"
     "Retrieved example queries (for style; may not match schema 1:1):
\n{examples}\n")
])
```

This prompt style aligns with LLM-era prompt engineering results showing that careful representation + example selection can materially improve performance (DAIL-SQL). [20] For "structure-first then fill identifiers" ideas, see "Few-shot Text-to-SQL Translation using Structure and Content Prompt Learning." [21]

Output parsing + SQL validation (schema + safety + dialect)

LangChain's structured output can significantly reduce parsing fragility compared with free-form text. [30] Still, you should validate generated SQL before execution.

A practical pattern is: 1) parse SQL into an AST (e.g., using sqlglot or another parser),
2) verify it is SELECT-only,
3) extract referenced tables/columns and confirm they exist in the schema slice,
4) enforce policy constraints (LIMIT, blocked tables),
5) optionally run `EXPLAIN` first (dialect-dependent).

```
from typing import Set, Tuple

import sqlglot


class SQLValidationError(Exception):
```

```
    pass


def validate_select_only(sql: str, dialect: str) -> None:
    try:
        expr = sqlglot.parse_one(sql, read=dialect)
    except Exception as e:
        raise SQLValidationError(f"SQL parse failed: {e}") from e

    if expr is None:
        raise SQLValidationError("Empty SQL expression after parsing.")

    # Very conservative: require top-level SELECT
    if expr.key.upper() != "SELECT":
        raise SQLValidationError(f"Only SELECT queries are allowed. Got:
{expr.key}")

    # Optional: enforce LIMIT (simple heuristic)
    if expr.args.get("limit") is None:
        raise SQLValidationError("Missing LIMIT. Add a LIMIT for safety.")


def extract_tables_and_columns(sql: str, dialect: str) -> Tuple[Set[str],
Set[str]]:
    expr = sqlglot.parse_one(sql, read=dialect)
    tables = {t.name for t in expr.find_all(sqlglot.expressions.Table)}
    cols = set()
    for c in expr.find_all(sqlglot.expressions.Column):
        # column string contains optional table qualifier
        cols.add(str(c))
    return tables, cols
```

Execution-guided loop with bounded retries

Execution-guided decoding uses database feedback to improve correctness. [15] In practice, you should implement a bounded loop:

- max retries (e.g., 2–4)
- per-query timeout
- row limit
- "safe mode" defaults to SELECT-only
- attach DB error message + schema slice + prior SQL to the repair prompt

```
from dataclasses import dataclass

@dataclass
class ExecResult:
```

```
    ok: bool
    rows: list
    error: str | None = None


def execute_sql_safely(db_conn, sql: str, timeout_s: int = 10) -> ExecResult:
    """
    DB execution stub: implement with your DB engine/driver.
    - enforce statement timeout
    - enforce max rows
    - run in read-only transaction if possible
    """
    try:
        # Example only: replace with DB-specific execution logic.
        cur = db_conn.cursor()
        cur.execute(sql)
        rows = cur.fetchmany(200)
        return ExecResult(ok=True, rows=rows)
    except Exception as e:
        return ExecResult(ok=False, rows=[], error=str(e))


def nl2sql_with_repair(
    *,
    llm_generate_sql,     # callable(inputs)->SQLDraft
    llm_repair_sql,       # callable(inputs)->SQLDraft
    db_conn,
    question: str,
    dialect: str,
    schema_selection_json: str,
    max_retries: int = 3,
):
    last_error = None
    draft = llm_generate_sql({
        "question": question,
        "dialect": dialect,
        "schema_selection_json": schema_selection_json,
        "examples": "",
    })

    for attempt in range(max_retries + 1):
        try:
            validate_select_only(draft.sql, dialect=draft.dialect)
        except SQLValidationError as e:
            last_error = f"VALIDATION_ERROR: {e}"
        else:
            exec_res = execute_sql_safely(db_conn, draft.sql)
            if exec_res.ok:
```

```
                return {"sql": draft.sql, "rows": exec_res.rows, "attempts":
attempt + 1}
            last_error = f"EXECUTION_ERROR: {exec_res.error}"

        # Repair step
        draft = llm_repair_sql({
            "question": question,
            "dialect": dialect,
            "schema_selection_json": schema_selection_json,
            "previous_sql": draft.sql,
            "error": last_error,
        })

    raise RuntimeError(f"Failed to produce executable SQL after retries. Last
error: {last_error}")
```

This loop is conceptually aligned with execution-guided decoding and SQL-PaLM's emphasis on execution-based filtering/selection, while remaining implementable with typical LLM APIs. [56]

Integrating Copilot SDK with LangChain in Python

The Copilot SDK is designed to embed the same agentic core as Copilot CLI, communicating via JSON-RPC to a Copilot CLI server. [4] Its Python SDK advertises streaming and provider configuration (BYOK). [57]

Because the SDK is in technical preview and exact APIs may evolve, the stable integration pattern is:

- implement a *thin* "LLM adapter" that exposes:
- `invoke(prompt) -> text`
- `stream(prompt) -> deltas`
- then wrap it behind LangChain's runnable interface or use it directly inside nodes.

A minimal sketch based on public examples (client start → session create → send and wait) appears in Copilot SDK issues. [58]

```
import asyncio
from typing import Optional

# Copilot SDK package naming can evolve; adjust import as needed.
from copilot import CopilotClient  # example seen in Copilot SDK discussions/
issues


class CopilotLLMAdapter:
    """
    Thin wrapper around CopilotClient/session for LangChain orchestration.
    Keeps Copilot session management outside LangChain to avoid hidden state.
```

```
    """

    def __init__(self, model: str, streaming: bool = False):
        self.model = model
        self.streaming = streaming
        self._client: Optional[CopilotClient] = None
        self._session = None

    async def start(self):
        self._client = CopilotClient()
        await self._client.start()
        self._session = await self._client.create_session({"model": self.model})

    async def invoke(self, prompt: str) -> str:
        if not self._session:
            raise RuntimeError("Call start() first.")
        # API shape may differ; adapt to the SDK's current methods.
        resp = await self._session.send_and_wait({"prompt": prompt})
        # Depending on SDK, resp may be structured; normalize to text.
        return str(resp)

    async def close(self):
        if self._client:
            await self._client.stop()
```

Operational notes grounded in SDK documentation: - you must install Copilot CLI separately and have it available for the SDK to manage the CLI process lifecycle; [46]
- the SDK supports BYOK, but explicitly notes key-based auth only (no Entra ID / managed identities). [47]
- tool permissions should be narrowed from default allow-all for an NL→SQL service. [46]

Evaluation plan: datasets, benchmarks, and test strategy

Benchmark datasets to use

Use a staged evaluation plan:

- **Spider** (cross-domain, unseen DB split): strong baseline for structural correctness and schema linking. [5]
- **WikiSQL** (single-table, simpler): useful early scaffolding and regression testing. [59]
- **BIRD** (harder realism/value grounding): closer to enterprise analytics needs. [7]
- **UNITE** (unified benchmark across multiple datasets and patterns): helps measure generalization breadth. [60]
- **Synthetic datasets** (optional) for coverage (e.g., SynSQL-2.5M) if you later explore fine-tuning. [61]

Be mindful that benchmark annotation errors and ambiguities can distort results; complement with manual error analysis and curated internal test suites. [24]

Metrics

Track at least: - EM (exact match) on canonicalized SQL where applicable. [5]
- Execution accuracy (primary "does it work?" metric). [62]
- Valid SQL rate (parses + passes safety constraints) as a first-line reliability metric (motivated by constrained decoding research). [12]
- Latency and token usage per query (operational).
- Repair loop rate (percent requiring retries), average retries, and failure reasons (observability). [15]

Unit and integration testing recommendations

A pragmatic test pyramid:

- **Unit tests** (fast):
- CSV ingestion correctness (schema docs and metadata),
- SQL validator correctness (SELECT-only enforcement, LIMIT enforcement, schema allowlist checks),
- prompt template determinism (golden snapshots of rendered prompts with fixed inputs).
- **Integration tests** (medium):
- ephemeral DB (SQLite file or containerized Postgres) with small schemas,
- "golden" NL queries with expected SQL semantics (validate results not only string),
- execution-guided repair tests using injected error cases.
- **CI patterns**:
- run integration tests on pull requests with a small DB fixture,
- run nightly benchmark regression (Spider dev subset) if compute budget allows,
- store evaluation artifacts (generated SQL, error traces) for diff-based debugging.

Deployment, monitoring, and governance

A production NL→SQL system must support reviewability and continuous improvement:

- **Structured logging and tracing**: log (a) user question, (b) retrieved schema/docs IDs, (c) generated SQL, (d) execution outcome, (e) repair attempts. LangChain callbacks provide a standard hook point. [63]
- **Query auditing**: store generated SQL and bind it to user/session IDs; include decision metadata ("selected tables," "applied limit") for compliance.
- **Drift detection**: track changes in schema CSVs, query failure rates, and retrieval distribution shifts (e.g., suddenly different tables are being selected).
- **User feedback loop**: allow users to mark answers correct/incorrect and capture corrected SQL when available; this becomes evaluation data and potentially fine-tuning data (careful governance). [64]
- **Access control**: enforce role-based access at the DB level; do not rely on LLM behavior for authorization. LangChain's security notes on SQL chains reinforce the need for scoped credentials. [27]

Security, privacy, and compliance considerations for grounding data

Key risks include: - leaking sensitive schema/value information through prompts or logs,
- executing unauthorized queries because the model inferred or hallucinated tables,
- prompt injection through user-provided text.

Mitigations: - **Minimize data in prompts**: retrieve only necessary schema slices; do not include raw PII values.
- **Metadata-based policy filtering**: label sensitive tables/columns (`pii_category`) and block unless explicitly authorized.
- **Redaction in logs**: store hashed identifiers; avoid storing full result sets.
- **Read-only DB roles**: enforce at the database layer; LangChain warns about destructive query risks if credentials are overly broad. [27]
- **Copilot data handling**: GitHub documentation states that (for individual subscribers) data including prompts and code snippets is not used for model training by default, and suggests consulting official data-handling documentation for the data pipeline lifecycle. [65]
- **BYOK governance**: if you use BYOK, ensure secrets management, key rotation, and provider-side policy controls; Copilot SDK supports BYOK but with key-based auth constraints. [66]

Prioritized implementation roadmap with milestones and risk mitigation

Because effort depends on unspecified factors (DB dialect count, schema sizes, latency SLOs, compliance posture), this roadmap prioritizes deliverable milestones rather than hard dates.

Milestone: Prototype (single dialect, local vector store) - Implement CSV ingestion → schema/query documents. - Implement dense retriever + BM25 retriever; add metadata filtering by `db_id`. - Implement schema linking step with structured output. - Implement SQL generation step with structured output + basic validator (SELECT-only + LIMIT). - Execute on a small read-only test DB fixture. - Risks: hallucinated identifiers; mitigate with strict schema validation and “selection must come from retrieved docs.” [67]

Milestone: Reliability hardening (execution guidance + hybrid retrieval tuning) - Add execution-guided repair loop (bounded retries, timeouts). - Add retrieval tuning (MMR, hybrid weights, curated exemplars). - Add unit/integration tests and CI fixtures. - Risks: repair loops mask deeper issues; mitigate with error taxonomy + logging, and measure “repair rate.” [68]

Milestone: Production controls (security + observability + governance) - Implement strict tool/permission model (disable non-DB tools). - Add audit logging, redaction, rate limiting, and caching. - Add per-user authorization mapping to DB roles or row-level security. - Risks: data leaks via logs/prompts; mitigate with redaction and minimized prompt context. [69]

Milestone: Benchmarking and continuous evaluation - Evaluate on Spider/WikiSQL/BIRD and internal workloads; store artifacts. - Add robustness test suite (paraphrases, schema perturbations). - Incorporate human review and feedback loop. - Risks: benchmark noise/annotation errors; mitigate with manual error analysis and internal gold sets. [70]

Milestone: Scaling and optional fine-tuning - If needed, explore fine-tuning or synthetic data augmentation (e.g., SynSQL-2.5M-style approaches) to improve schema linking and robustness. - Risks: high

MLOps complexity; mitigate by starting with prompt+retrieval (DAIL-SQL) and adding fine-tuning only with clear ROI. [71]

---

[1] [9] [54] https://github.com/microsoft/rat-sql
https://github.com/microsoft/rat-sql

[2] [12] [14] [36] [67] https://arxiv.org/abs/2109.05093
https://arxiv.org/abs/2109.05093

[3] [11] [26] https://docs.langchain.com/oss/python/langchain/sql-agent
https://docs.langchain.com/oss/python/langchain/sql-agent

[4] [46] [47] [66] https://github.com/github/copilot-sdk
https://github.com/github/copilot-sdk

[5] [8] [45] [70] https://arxiv.org/abs/1807.03100
https://arxiv.org/abs/1807.03100

[6] https://github.com/taesiri/ArXivQA/blob/main/papers/2306.00739.md
https://github.com/taesiri/ArXivQA/blob/main/papers/2306.00739.md

[7] https://aclanthology.org/2025.coling-main.256.pdf
https://aclanthology.org/2025.coling-main.256.pdf

[10] https://www.reddit.com/r/deeplearning/comments/141gzg7/sqlpalm_paper_summary_video/
https://www.reddit.com/r/deeplearning/comments/141gzg7/sqlpalm_paper_summary_video/

[13] [25] https://dl.acm.org/doi/pdf/10.1145/3769822
https://dl.acm.org/doi/pdf/10.1145/3769822

[15] [17] [37] [52] [56] [68] https://arxiv.org/pdf/1807.03100
https://arxiv.org/pdf/1807.03100

[16] [23] [64] SQL-PaLM: Improved Large Language Model Adaptation for Text-to-SQL (extended)
https://arxiv.org/abs/2306.00739?utm_source=chatgpt.com

[18] https://arxiv.org/abs/2201.11903
https://arxiv.org/abs/2201.11903

[19] https://arxiv.org/abs/2203.11171
https://arxiv.org/abs/2203.11171

[20] [22] [62] https://www.vldb.org/pvldb/vol17/p1132-gao.pdf
https://www.vldb.org/pvldb/vol17/p1132-gao.pdf

[21] Few-shot Text-to-SQL Translation using Structure and ...
https://dspace.mit.edu/bitstream/handle/1721.1/151087/3589292.pdf?isAllowed=y&sequence=1&utm_source=chatgpt.com

[24] https://vldb.org/cidrdb/papers/2026/p5-jin.pdf
https://vldb.org/cidrdb/papers/2026/p5-jin.pdf

[27] [69] https://reference.langchain.com/v0.3/python/experimental/sql/langchain_experimental.sql.base.SQLDatabaseChain.html
https://reference.langchain.com/v0.3/python/experimental/sql/langchain_experimental.sql.base.SQLDatabaseChain.html

28 https://reference.langchain.com/v0.3/python/community/agent_toolkits/langchain_community.agent_toolkits.sql.toolkit.SQLDatabaseToolkit.html
https://reference.langchain.com/v0.3/python/community/agent_toolkits/langchain_community.agent_toolkits.sql.toolkit.SQLDatabaseToolkit.html

29 https://reference.langchain.com/v0.3/python/community/agent_toolkits/langchain_community.agent_toolkits.sql.base.create_sql_agent.html
https://reference.langchain.com/v0.3/python/community/agent_toolkits/langchain_community.agent_toolkits.sql.base.create_sql_agent.html

30 55 https://docs.langchain.com/oss/python/langchain/structured-output
https://docs.langchain.com/oss/python/langchain/structured-output

31 38 42 53 https://docs.langchain.com/oss/python/integrations/retrievers/bm25
https://docs.langchain.com/oss/python/integrations/retrievers/bm25

32 63 https://reference.langchain.com/python/langchain_core/callbacks/
https://reference.langchain.com/python/langchain_core/callbacks/

33 https://docs.langchain.com/oss/python/langgraph/workflows-agents
https://docs.langchain.com/oss/python/langgraph/workflows-agents

34 https://reference.langchain.com/v0.3/python/core/runnables/langchain_core.runnables.base.Runnable.html
https://reference.langchain.com/v0.3/python/core/runnables/langchain_core.runnables.base.Runnable.html

35 https://reference.langchain.com/v0.3/python/core/globals/langchain_core.globals.set_llm_cache.html
https://reference.langchain.com/v0.3/python/core/globals/langchain_core.globals.set_llm_cache.html

39 https://docs.trychroma.com/
https://docs.trychroma.com/

40 43 https://docs.weaviate.io/weaviate/search/hybrid
https://docs.weaviate.io/weaviate/search/hybrid

41 https://qdrant.tech/articles/sparse-vectors/
https://qdrant.tech/articles/sparse-vectors/

44 https://github.com/pgvector/pgvector
https://github.com/pgvector/pgvector

48 https://github.blog/news-insights/company-news/build-an-agent-into-any-app-with-the-github-copilot-sdk/
https://github.blog/news-insights/company-news/build-an-agent-into-any-app-with-the-github-copilot-sdk/

49 https://docs.github.com/en/copilot/concepts/rate-limits
https://docs.github.com/en/copilot/concepts/rate-limits

50 65 https://docs.github.com/copilot/how-tos/manage-your-account/managing-copilot-policies-as-an-individual-subscriber
https://docs.github.com/copilot/how-tos/manage-your-account/managing-copilot-policies-as-an-individual-subscriber

51 https://resources.github.com/learn/pathways/copilot/essentials/how-github-copilot-handles-data/
https://resources.github.com/learn/pathways/copilot/essentials/how-github-copilot-handles-data/

57 copilot-sdk/python/README.md at main
https://github.com/github/copilot-sdk/blob/main/python/README.md?utm_source=chatgpt.com

58 https://github.com/github/copilot-sdk/issues/384
https://github.com/github/copilot-sdk/issues/384

59 https://aclanthology.org/2021.spnlp-1.2.pdf
https://aclanthology.org/2021.spnlp-1.2.pdf

60 https://arxiv.org/abs/2305.16265
https://arxiv.org/abs/2305.16265

61 71 https://www.vldb.org/pvldb/vol18/p4695-li.pdf
https://www.vldb.org/pvldb/vol18/p4695-li.pdf