Schema-Aware NL2SQL: Adaptive Query Generation Across Databases

Architecture Overview

The **overall architecture** consists of a natural language front-end, a schema-aware NL2SQL engine (powered by LLMs), and a database execution layer. The system workflow is as follows (see steps below):

1. **User Query:** A user asks a question in natural language via a web interface.

2. **Schema Retrieval:** The back-end fetches the current database's schema (table names, columns, types, and relationships) dynamically. This schema serves as context for the NL2SQL model.

3. **NL2SQL Model (LLM):** An LLM-based module takes the user's question *along with the schema* as input, and generates an SQL query that answers the questionhuggingface.cohuggingface.co. The model is *schema-aware*, meaning it adapts to whatever tables/columns are present by conditioning on the schema in the prompt or input. This approach allows the model to generalize to databases it wasn't trained onhuggingface.co.

4. **SQL Execution:** The generated SQL query is executed on the target database. The system uses a database connector to run the query and retrieve results.

5. **Results Return:** Query results are returned to the user. The front-end can display these in a user-friendly format (e.g. a table). In some implementations, an LLM might also be used to convert raw results into a natural language answer or visualization, but by default we return the raw result set.

Figure: High-Level System Flow: *User question → [NL2SQL Engine (LLM + Schema)] → SQL Query → [Database Execution] → Results.* Each component is decoupled for modularity: the NL2SQL model is independent of the database engine, communicating via SQL strings.

To handle **dynamic or unknown schemas**, the system injects schema information at query time. That is, for each new database, the schema is retrieved (e.g. via `information_schema` queries or an ORM) and provided to the model. Large Language Models have made it feasible to translate NL questions into SQL using schema-aware prompts in a zero-shot or few-shot mannerarxiv.org. If the database has *many tables*, an initial step can narrow down relevant tables to include. For example, LangChain's SQL chain uses a two-step approach: *"1. Based on the query, determine which tables to use. 2. Based on those tables, call the normal SQL generation chain."*medium.com. This kind of **schema routing** or filtering is important when dealing with huge schemasarxiv.org – the system might first identify which subset of the schema is relevant (possibly via simple keyword matching or an auxiliary model) before prompting the NL2SQL model. Recent research (e.g. DBCopilot) suggests that automatically identifying the

relevant database and tables can make querying *massive* schemas feasible under LLM token limitsarxiv.org.

In summary, the architecture enables non-technical users to query arbitrary databases by combining an LLM-driven query generator with real-time schema knowledge and a live database connection. This avoids hardcoding any single schema. Notably, hooking an LLM up to a database provides factual grounding – the model's answers come from executing SQL on the DB, reducing open-ended hallucinationmedium.com.

Technology Stack

Back-End & Language: The back-end will likely be implemented in Python, given the rich ecosystem for ML and database connectivity. Python offers libraries like **Hugging Face Transformers** for loading/training LLM models, and **LangChain** for orchestration (prompt management, chaining steps). For database access, Python's **SQLAlchemy** is ideal – it can connect to *any* SQL dialect (MySQL, PostgreSQL, SQLite, SQL Server, etc.) via a unified APImedium.com. SQLAlchemy allows reflection to get table/column metadata, which we'll use to fetch the schema. If using LangChain, it has built-in classes (e.g. `SQLDatabaseChain`) that integrate with SQLAlchemy and even handle multi-step querying for large schemasmedium.com.

LLM Model: We can use a pre-trained sequence-to-sequence model (such as **T5**, **BART**, or **CodeLlama**) fine-tuned for text-to-SQL. These models can be run with Hugging Face Transformers in Python. For instance, T5 has been adapted to text-to-SQL by incorporating the schema in the inputhuggingface.co, and such models are available on HuggingFace (e.g. a T5-large fine-tuned on Spiderhuggingface.co). If leveraging LangChain in production, one could also integrate an OpenAI GPT-4 or GPT-3.5 model via API for query generation (prompted with the schema), but our plan emphasizes a self-hosted model to avoid relying on proprietary APIs.

Database: The underlying database can be any relational DB. For development, **SQLite** is convenient (no server needed) – one can easily load the Spider dataset databases into SQLite for local testing. In a deployed setting, the system could connect to a live **MySQL** or **PostgreSQL** database (or any supported DBMS). The app should be configured with the DB connection string, and using SQLAlchemy means the same code can connect to different DB types with just a change in the connection URLmedium.com.

Web Interface: For the user interface, a simple web app can be built using a Python web framework or other tools:

- **FastAPI or Flask** (Python) can serve a REST API where the front-end sends the NL query and receives results.

- **Streamlit or Gradio** (Python) are high-level frameworks to quickly create an interactive UI for demos, which could be suitable for a prototype. (Indeed, Dataherald's open-source

project uses a Streamlit app front-endgithub.com).

- Or a custom front-end (React/HTML/JS) that calls the back-end API. Given the importance of smooth interaction, a modern web UI that shows the question, the SQL output, and possibly the results in a table is recommended.

Libraries & Resources:

- **Datasets:** The **Spider** text-to-SQL datasetmedium.comhuggingface.co (and others like WikiSQL) for training (discussed in the next section).

- **Hugging Face Datasets/Transformers:** to load data and model. The Spider dataset is available via HuggingFace's datasets hubhuggingface.co, and models like T5 or BART can be fine-tuned using the Transformers Trainer.

- **Example Repositories:** Projects like **Dataherald** (open-source NL2SQL engine) and **PremSQL** provide reference implementations. *Dataherald's engine* demonstrates a LangChain-based NL2SQL with a Python back-endgithub.com. *PremSQL* is an open-source library focusing on local deployment with small models for text-to-SQLblog.premai.io – it emphasizes privacy by avoiding third-party APIs and supports custom pipelines. These can be excellent references for architecture and best practices. Another example is **WrenAI** (GenBI), which showcases a full-stack approach (including a semantic layer on top of the raw schema to better map business terms to DB schema)github.com.

- **Utilities:** To handle SQL dialect differences (discussed later), a library like **SQLGlot** can be used for SQL transpilationcodecut.ai. Additionally, standard Python DB-API drivers (psycopg2 for Postgres, PyMySQL for MySQL, etc.) will be used under the hood (SQLAlchemy uses these). For logging and monitoring, standard Python logging or an APM tool can be integrated to trace queries and model outputs.

Overall, the stack is Python-centric for ease of integrating the ML model and database operations in one place. The choice ensures we can train models, run inference, and serve web requests all in a unified environment.

Model Training Pipeline

Building a robust NL2SQL model **from scratch** involves assembling training data, choosing a model architecture, and fine-tuning it to generate SQL from natural language. We outline the training pipeline below:

- **Data Collection & Preparation:** We will leverage public text-to-SQL datasets. The primary dataset is **Spider**, a large-scale benchmark of ~10,000 questions across 200 databasesmedium.com. Spider is specifically designed for cross-database generalization: the train and test sets have different database schemas, so a model can't just memorize one schema – it must truly learn to adapt to new schemasmedium.com. This aligns perfectly with our goal of an adaptive NL2SQL system. Each Spider example provides: a natural language question, the corresponding SQL query, and the database schema (tables, columns, foreign keys). Another dataset, **WikiSQL**, offers a huge number of simpler queries on single tablesmedium.com, which can be used for initial pre-training, but it lacks joins and complexity. We can use WikiSQL for a warm-up if needed, then move to Spider for multi-table queries and richness. (For completeness, newer benchmarks like **BIRD** focus on real-world messy schemas and could be considered for evaluationmedium.com, but Spider is the go-to for model training).

- **Input Formatting (Schema Integration):** The key challenge is feeding the *schema* into the model along with the question. Our training pipeline will construct input sequences that incorporate the database schema in a structured way. For example, the fine-tuned T5 model from the PICARD project used an input format: `[question] | [db_id] | [table1]: [col1], [col2], ... | [table2]: [col1], ...`, enumerating each table and its columnshuggingface.co. Another approach (used by some prompt-based methods) is to include CREATE TABLE statements or a list of tables and columns as contextmedium.com. We will choose a consistent format and apply it to all training examples. One proven format is to list each table with its columns and their types, and possibly primary/foreign keys, separated by special tokens (e.g. a `[SEP]` token or a newline)huggingface.co. For instance, the schema could be serialized as: *`table1 col1 type, col2 type, ... primary key(...); [SEP] table2 col1 type, ...; [SEP] ...`*. Including types and keys can help the model understand relationships (e.g. if a foreign key indicates a join path). The model then sees a concatenation like: `Question: ...? Schema: ...` and is trained to produce the correct SQL query as outputhuggingface.cohuggingface.co. This method explicitly teaches the model to *condition on the schema*, rather than ignoring it. As noted in one model card, *"we added the database schema to the question, as we wanted the model to generate a query over a given database"*huggingface.co. By learning the mapping between NL and schema elements during training, the model becomes adept at handling unseen schemas at inference timehuggingface.co.

- **Model Choice:** We will fine-tune a sequence-to-sequence transformer. Good choices include **T5** (e.g. T5-base or T5-large) or **BART**, which have been used successfully for text-to-SQLhuggingface.co. For instance, a T5-large model adapted for text-to-SQL achieved strong performance on Spider by incorporating schema contexthuggingface.co. Starting from a pre-trained model (like T5) is crucial – the model already has broad language understanding and some coding ability, which speeds up convergence. We then train it on our formatted (question+schema → SQL) pairs. Training from scratch

(random initialization) would be data-inefficient given the complexity of SQL, so we leverage transfer learning. If resources allow, one could fine-tune a 2-3 billion parameter model (as done in PICARD with T5-3Bhuggingface.co) for best accuracy. Otherwise, smaller models (T5-base at ~220M or T5-large ~770M) can be fine-tuned on a single GPU. We'll also consider open source code-focused models like **CodeT5** or **LLaMA 2** (CodeLlama variant) if they are easier to fine-tune for SQL generation. Recent efforts like PremSQL are even exploring training small models from scratch for NL2SQLblog.premai.io – for example, a distilled model that could run on CPU for local deployment – but initially, fine-tuning a proven architecture on Spider is the straightforward path.

- **Training Process:** Using the **HuggingFace Transformers** framework, we will write a training script that:
    - Reads the Spider dataset (which includes the schemas and queries). We can use the `datasets` library to load ithuggingface.co.
    - For each training example, format the input string (question + schema) and the target string (SQL query).
    - Tokenize these with the model's tokenizer (taking care that the combined length is within model's limit, e.g. T5 can handle 512 tokens; we may truncate very large schemas or use schema filtering as a preprocessing step).
    - Train with teacher forcing to minimize the cross-entropy loss of generating the correct SQL. We'll use techniques like *label smoothing* and *learning rate scheduling* to help convergence. Because SQL generation is a structured task, it's helpful to train with the **exact match accuracy** metric in mind – we might use a custom callback to compute exact string match or execution accuracy on a validation set each epoch.
    - Typically, models are trained for a certain number of epochs on Spider (e.g. 20-50 epochs depending on batch size and learning rate) until performance on dev set stops improvinghuggingface.co. Early stopping on exact-match accuracy can be applied.
    - We must ensure the model doesn't overfit to the training schemas – an indicator is if it struggles on the Spider dev set which has unseen schemas. Data augmentation can help: Spider has variations like Spider-Syn (which paraphrases questions)medium.com, and we can include those for robustness. Also, we could synthesize additional training data (for example, generating NL-SQL pairs using large models or using the **GAP** approach to pretrain on synthetic data) if needed.

- **Model Evaluation:** After training, we evaluate on Spider's dev set (which contains unseen databases). Key metrics are:
    - **Exact Match Accuracy:** Does the generated SQL exactly match the ground truth string (ignoring formatting)?
    - **Execution Accuracy:** Does the SQL execute to the correct result on the database? Execution accuracy is often higher than exact-match, since there can be many SQL formulations for the same answer. We will use the official Spider evaluation script for this.
    - A well-trained model might achieve ~70% execution accuracy on Spider devhuggingface.co. For instance, the T5-3B model with constrained decoding got ~79% execution accuracyhuggingface.co. These benchmarks give us a sense of the performance to expect – NL2SQL is non-trivial, and errors will occur, so we'll need to handle them gracefully in the app (for example, if the model generates an incorrect query or one that fails to execute, the system should catch the SQL error and possibly return a failure message or ask for clarification).
- **Incorporating Constraints:** One enhancement during decoding is to enforce SQL correctness constraints. For example, the **PICARD** method can be applied at inference: it incrementally parses the generated SQL and ensures that each token conforms to SQL grammar and the given schemahuggingface.co. This prevents the model from producing invalid table or column names – any token not present in the schema can be rejected during generation. We can integrate a library or the PICARD code as part of the model's decoding step to improve reliability. This doesn't require model retraining, but uses the schema metadata to guide beam search. We mention this as a part of the training pipeline because it leverages training-time information (grammar) at inference.
- **Transfer Learning & Fine-Tuning:** If we want the model to handle multiple SQL dialects or particular domain data, we can further fine-tune it on additional data. For example, after Spider, we could fine-tune on **customer-specific queries** if such data is available, or on **synthetic datasets** (like the one from Gretel.aigretel.ai which provides thousands of synthetic NL-SQL pairs). The pipeline remains the same – just extend the training data and possibly incorporate an identifier of the SQL dialect in the input prompt (e.g. start the prompt with "dialect: MySQL" or use slightly different formatting per dialect).

Finally, we save the fine-tuned model and tokenizer. This model will be loaded by the back-end to power the NL2SQL inference. We also save some example prompt templates and a mapping of how to feed schema into the model, as the serving code will need to do the same formatting for incoming user questions as was done during training.

Handling Schema Adaptation

Handling *unknown and dynamic schemas* is the crux of this project. The system should be able to take any new database schema at runtime and generate correct queries. Several strategies and best practices enable this:

- **Schema as Part of Input:** As discussed above, the model is trained to expect schema information with each query. At runtime, when a user selects or connects a new database, the application will retrieve the schema (table names, columns, types, and relationships) from the database's metadata. This can be done via SQLAlchemy's inspection (`db.inspect(engine)` in Python can list tables and columns) or querying information_schema tables. The schema is then formatted into the same structured representation that the model was trained on (e.g. list of tables and columns). By providing the *current schema* in every prompt (for prompting-based methods) or input sequence (for our fine-tuned model), we ensure the LLM only generates SQL elements that actually exist in that databasehuggingface.cohuggingface.co. The HuggingFace model card for a Spider-trained T5 emphasizes that exposing the database schema in the input *"allows the model to learn the mapping of the schema to the expected output"* and generalize to new schemashuggingface.co, which is exactly what we exploit.

- **Schema Linking:** Simply providing the schema textually is often enough for an LLM, but we can improve accuracy by emphasizing parts of the schema that are relevant. *Schema linking* refers to identifying which tables/columns are likely referenced by the user's question. For example, if the question is "Find the average age of French musicians", the term "musicians" likely links to a table or column in the schema (maybe a table `singer` with a `Age` column)huggingface.co. Our model, if trained well, will learn to pick up on substring overlaps or semantic similarities. In practice, we can assist this by ordering the schema or highlighting certain parts. One simple approach is to **reorder tables** in the prompt such that tables whose names (or column names) have keywords matching the question appear first. Another is to **include value examples**: some pipelines include a few sample rows or values for each table in the promptmedium.com, which can help the model understand table contents (e.g. a sample row in a `singer` table might show a Country value "France", indicating that table stores singers and their country). We might not always include data for privacy/performance reasons, but including just the schema is the minimum.

- **Adaptive Prompting:** If using a hosted LLM via LangChain (as an alternative approach), the prompt can be engineered to adapt to schemas. For instance, the system prompt can say: *"The database has the following tables: ..."* and list them, then *"Use only these tables to answer the question."* We should instruct the model to strictly stick to available schema and not make up table names. In fact, providing a clear instruction is important. For example, PremSQL's default prompt tells the model: *"You will be given schemas of tables of a database. Your job is to write a correct, error-free SQL query based on the question. Make sure the column names are correct and exist in the table. For column names with spaces, use backticks. Think step by step and always check the schema."*blog.premai.io. This kind of guidance, included in a system prompt or as part of

fine-tuning data, reinforces schema adherence.

- **Large Schema Considerations:** If a database schema is very large (hundreds of tables), it may not be feasible to include the entire schema in one prompt due to token limitsarxiv.org. In such cases, the system should implement a **schema filtering** step as mentioned. One approach is a simple keyword match: find which table or column names overlap with question words (e.g. "age" -> find tables with an Age column). Another approach is using an embedding-based retrieval: treat each table schema description as a document, and use an embedding model to find which tables are semantically closest to the question. For example, one could encode the question and all table names with description, then choose the top-k tables to include. This is akin to a Retrieval-Augmented Generation (RAG) approach for text-to-SQLarxiv.org. LangChain's `SQLDatabaseSequentialChain` automates a bit of this by first selecting tables relevant to the querymedium.com. Our system can adopt a similar tactic: **Step 1:** identify relevant tables (and possibly columns) -> **Step 2:** feed only those (plus maybe any foreign key linked tables) into the model input. This keeps prompts concise and focused, avoiding confusion from unrelated tables.

- **Validation & Correction:** After the model generates SQL, the system can validate it against the schema. A straightforward validation is to attempt to prepare the query (not execute, just parse) via the database engine to see if it's syntactically valid. If it fails (e.g. references a non-existent column), we know the model hallucinated something outside the schema. In a robust system, one could implement a feedback loop: either attempt to automatically correct the query by removing or fixing the bad identifier, or prompt the LLM again with a message about the error. In initial development, we'll likely just notify the user that the question couldn't be understood well enough to generate a correct query. In advanced implementations, techniques like PICARD (which ensures validity during generation) or SQL parsers can be used to catch errors. Since our model was trained on many schemas, it may occasionally use a column name that "sounds" right but isn't exactly in the current schema (e.g. using `first_name` vs actual `fname`). We could do a closest-match suggestion: if the model's output column name has a Levenshtein distance of 1 to an actual column, maybe auto-correct it. But such heuristic fixes must be done carefully to avoid wrong queries. Often, re-prompting the model with a clarification is safer.

- **Dynamic Schema Updates:** If the underlying database schema changes (tables added, etc.), the system should detect this (perhaps by refreshing its schema cache or on each query fetch the latest schema) and update the context. The design should allow re-loading the schema without restarting the whole model. For example, if using an open model, we simply construct a new input string with the new schema. If using a prompt with a chat model, we include the new schema in the prompt anew. There's no fine-tuning needed for new schemas – that's the advantage of the schema-aware approach.

- **Multi-Database Handling:** If the application will connect to different databases (say the user can choose among multiple connections), the architecture should isolate the schema per database. For instance, the user might first select which database they want to query (or the question might implicitly refer to a certain domain). The system then loads that DB's schema and proceeds. In a more automated scenario, one could attempt to auto-detect which database a question is referring to (this is what the DBCopilot research calls *schema routing*arxiv.org). However, in most practical cases, the user or the app context will clarify which database is in use. We simply need to ensure the correct schema is fed to the model for the chosen database.

In essence, schema adaptation is achieved by always providing fresh schema context to the model and by training the model to utilize that context. By combining these with validation steps and prompt engineering, the system remains robust across different databases.

Dialect Handling

Handling different SQL **dialects** is an important consideration, since MySQL, PostgreSQL, SQLite, etc., have minor differences in SQL syntax and functions. There are several strategies to make our NL2SQL system dialect-aware or dialect-agnostic:

- **Unified SQL Subset:** One approach is to train (or prompt) the model to output a standardized SQL that is mostly dialect-neutral. Core SQL (SELECT/FROM/WHERE/JOIN/GROUP BY) is very similar across dialects (thanks to ANSI SQL standards). Spider's queries, for example, stick to a fairly standard style (the evaluation actually runs them on SQLite). If our model generates standard SQL, it should run on most databases with minimal tweaks. We should be careful about certain keywords: e.g. use `LIMIT` (SQLite/MySQL) vs `FETCH FIRST N ROWS` (Oracle) or `TOP N` (SQL Server) depending on the backend. If the model was trained primarily on one dialect (say, SQLite style), it might always produce `LIMIT`. We can then post-process for other dialects. For example, if the target is SQL Server, and the model output contains `LIMIT N`, we could transform that to `TOP N` in the SELECT clause. A simple way is to incorporate the *dialect name* into the prompt. E.g., prepend "**Dialiect: MySQL**" or have the system prompt say "You are an SQL assistant for a MySQL database." This might cue an LLM to produce ``` backticks for identifiers or proper function names for that dialect.

- **LLM Dialect Few-Shot:** We can include examples of dialect-specific queries in few-shot prompts. For instance, show the model an example of a PostgreSQL query that uses `ILIKE` for case-insensitive match, or an example of a MS SQL query with `TOP`. However, controlling an LLM at that granularity can be hit-or-miss. Fine-tuning on a specific dialect is more reliable.

- **Post-generation Transpilation:** A pragmatic solution is to generate SQL in one canonical dialect (e.g. SQLite/Postgres) and then use a **SQL transpiler** library to convert it to the target dialect. **SQLGlot** is a powerful Python library that can parse a SQL query and output it in another dialectgithub.comcodecut.ai. For example, SQLGlot can translate a query with `LIMIT` into an equivalent one with `TOP` for T-SQL, or adjust date functions from one flavor to another. We could integrate SQLGlot in the execution pipeline: after the LLM produces SQL, parse it with SQLGlot, then `.transform()` to the dialect of our current database. This adds a reliable rule-based layer to handle syntax differences that the model might not handle. It's worth noting that SQLGlot supports 20+ dialects (including MySQL, Postgres, SQLite, Oracle, Snowflake, etc.)codecut.ai. It won't catch every subtle difference, but for many cases it works. Using such a tool helps us abstract over dialects – the NL2SQL model doesn't need to be perfect for each SQL flavor, as we have a converter.

- **Parameterized Queries and ORM Abstraction:** Another way to abstract differences is to not rely on raw SQL strings for everything. For instance, for safely executing queries and avoiding SQL injection, one should use parameterized queries (which our system can do by parameter binding, though if the model outputs a literal, we'd have to replace it with a parameter). ORMs like SQLAlchemy can construct queries in a dialect-agnostic way, but here the query is produced by the model as text, so we are essentially going around the ORM. However, we could use SQLAlchemy's SQL string compiler for certain adjustments if needed. A creative idea: interpret the model's output using SQLAlchemy's expression language. This would be complex and likely unnecessary; using SQLGlot is simpler.

- **Testing on Dialects:** We should test the system on a few databases. For example, set up a **PostgreSQL** instance with a sample schema and verify the generated queries run. Do the same with **MySQL** or **SQLite**. Common dialect differences to watch for:

    - Quoting style for identifiers: Postgres uses double quotes for case-sensitive identifiers, MySQL uses backticks. Our model might output quotes one way; if it outputs unquoted lowercase identifiers, that usually works on most unless reserved words.

    - String concatenation syntax (e.g. `CONCAT()` function vs `||` operator).

    - Date functions and formats (e.g. `DATE_TRUNC` in Postgres vs different in MySQL).

    - Limit/offset syntax as mentioned.

    - Case sensitivity in string comparison (MySQL by default case-insensitive for TEXT, Postgres is case-sensitive unless ILIKE is used).

- We might include in our prompt instructions something like: *"Use standard SQL. Avoid vendor-specific functions if possible."* Unless the question explicitly needs something like a full-text search or regex matching, which are highly dialect-specific, our focus is on standard analytical queries.

- **Dialects in Training Data:** If we wanted a truly dialect-aware model, we could augment training data with multiple versions of the same query in different dialects. However, this can confuse the model if not done carefully. A simpler approach: train on one dialect (say Spider which is akin to SQLite), and rely on conversion for others. If certain clients need precise dialect features, we fine-tune a separate model for that dialect.

- **Abstracting Over Dialects:** In the ultimate design, the user shouldn't have to worry about the SQL dialect. They just ask a question, and the system returns an answer. Whether internally it's MySQL or Oracle should be invisible. By incorporating one of the above strategies, we ensure the *same natural language question* can be answered regardless of the SQL engine underneath. This abstraction is a key feature of our project.

To illustrate, suppose a user asks *"What are the top 5 selling products last year?"*. The model might output `SELECT product_name, SUM(sales) AS total_sales FROM Sales JOIN Products USING (product_id) WHERE year = 2024 GROUP BY product_name ORDER BY total_sales DESC LIMIT 5;`. If running on MySQL or SQLite, this is fine. If running on SQL Server, our code could detect the dialect and transform that to use `SELECT TOP 5 ... ORDER BY total_sales DESC;` and drop the `LIMIT` clause. The rest of the query remains the same. This way, the user gets the correct result without knowing that, under the hood, the system adjusted the syntax for the specific database.

In summary, we will implement dialect handling by a combination of careful prompting (or fine-tuning) for core SQL and leveraging a transpiler for the edge cases. The goal is to *write once, run anywhere* for the generated SQL.

Front-End and Back-End Integration

Integrating all components into a web application involves setting up the back-end server that interfaces with the LLM model and database, and a front-end UI for user interaction. Here's the plan for integration:

Back-End (Server): We will create a server (e.g. using **FastAPI** or **Flask** in Python) that exposes endpoints for the NL2SQL service. Key functionalities of the back-end:

- **Database Connector Initialization:** Using SQLAlchemy, the server will maintain a connection pool to the chosen database(s). At startup (or upon receiving a request if multiple DBs), it will load the schema via reflection. We might store the schema in an

in-memory structure or as a serialized string to avoid repeated overhead.

- **Model Loading:** The fine-tuned NL2SQL model (from the training step) will be loaded into memory (probably on server start). If the model is large and running on GPU, the server should be hosted on a machine with adequate GPU resources. Alternatively, we might run the model on CPU for smaller models or use quantization to speed it up if needed. We will also load the corresponding tokenizer. If using an external API (like OpenAI via LangChain) instead of a local model, then we just ensure the API keys and prompt templates are set up.

- **API Endpoint:** Define an endpoint (e.g. `POST /query`) that accepts a user's natural language question (and perhaps a parameter indicating which database or schema to use, if multiple are available). The request payload might include an identifier for the database, or we handle a single database in this project scope.

- **Processing Pipeline:** When a request comes in, the back-end will perform the steps:

    - Look up the relevant schema (if multi-DB, select the right one).

    - Format the model input (e.g. `"Question: ... Schema: ..."` as per training format).

    - Run the model to generate SQL. This may involve a `.generate()` call for seq2seq models, or LangChain's chain invocation if using that. We might set a cutoff on generation length (e.g. not more than 256 tokens output) to avoid runaway.

    - Post-process the SQL string: strip any excessive formatting, maybe fix common issues (like the model sometimes might produce a trailing semicolon or unnecessary explanation – our prompts will try to avoid this by instructing output to be just the query). Also apply dialect conversion here if needed using SQLGlot, based on the target DB.

    - **Safety check**: It's wise to ensure the generated SQL is a **SELECT** query (not a DDL or DML that could modify data). Our instruction strongly says to only generate queries that answer the question, but to be safe, we can parse the SQL (with sqlparse or SQLGlot) and verify it doesn't contain forbidden operations. If a malicious or faulty prompt caused a `DROP TABLE`, we should catch that and refuse. In practice, if using our fine-tuned model on Spider, it has only seen SELECT queries, so it should never produce a DELETE/UPDATE. Additionally, using a read-only database user for the connection is a good precaution.

    - Execute the SQL on the database using SQLAlchemy. This can be done with a connection or session. We will catch any SQL execution errors (e.g. if the query

is invalid or times out).

    - Fetch results. We may format the results as JSON or a list of records.

    - Return the results (and possibly the SQL itself) in the response.

- **Back-End Response:** The response to the front-end could look like: `{ "sql": "<generated SQL>", "results": [...], "error": null }`. Including the SQL in the response is useful for transparency (the user can see the generated query) and debugging. If the query failed, we might return an error message instead of results.

- **Front-End (UI):** The front-end can be a simple web page or a more sophisticated application. For a start, a basic interface with a text input box for the question and a submit button is enough. Upon submission, it calls the back-end (e.g. via an AJAX call or form submit) to get the results.

    - The UI should display the results in a readable format, e.g. an HTML table for table-like results. Many queries will produce a table of rows and columns, so showing that is natural. For aggregate or single-value results, it can still be a small table or just a highlighted number.

    - Optionally, display the SQL query that was run. This can build user trust and help in case the user wants to refine the question (they might spot that the query did something slightly different than intended).

    - If the system supports multiple databases or schemas, the UI could provide a dropdown to select the dataset or schema (for example, choose "Sports DB" vs "Employees DB"). Alternatively, if the user is technical, they might even type the schema name in the question (but that's not user-friendly; better to handle behind scenes).

    - Provide guidance in the UI: e.g. a placeholder text like "Ask a question about the data...". One can also list what tables are available (essentially a schema preview) in the interface so that users know what terms to use. This is something some NL2SQL interfaces do – showing the schema to the user can help them phrase questions correctly. However, our system is aimed to not require the user to know the schema, so it's optional.

    - Error handling on UI: if the back-end returns an error (e.g. "I couldn't understand the question" or "Query failed"), show that message. Possibly prompt the user to rephrase.

- **Iterative Refinement and Interaction:** In a more advanced web app, you could allow the user to have a dialogue (like follow-up questions, or corrections). This would require

maintaining context or prior queries. LangChain and similar frameworks can maintain conversational memory, but handling follow-ups is beyond the core scope. Our initial interface will be one question = one answer. If needed, the user can always ask a new question.

- **Performance Considerations:** The back-end will have latency mainly from the LLM generation step. A smaller model on a GPU can generate the SQL in a second or two. Execution on the database is usually fast unless the query is complex on a huge DB. We should consider setting a timeout for both the model and the SQL execution. If using a large model (billions of parameters) on CPU, it could be slow (10s of seconds) – not ideal. So deployment likely needs at least a decent GPU or use of an optimized inference engine (like ONNX Runtime or FasterTransformer for the model). We can also cache results for repeated questions if needed, though in an interactive setting that may not be crucial.

- **Logging and Monitoring:** We should log queries asked and the SQL generated, both for debugging and for possibly improving the model. If we find systematic errors (like always misinterpreting a certain phrase), we can address that in the model or prompt. Logging must of course be done in a way that doesn't log sensitive data (if any). But since this is user-provided questions, it's usually fine to log for internal debugging.

- **Guidance for Deployment:** Once the app is working, containerization with Docker would be useful to reproduce the environment. The Docker image would include the model weights (which might be large, so perhaps mount a volume for those), the code, and necessary dependencies (transformers, pytorch, SQLAlchemy, etc.). For a cloud deployment, ensure the machine has a GPU if needed and proper security (no open database ports, etc., only the API endpoint exposed).

- **Security:** As mentioned, use a read-only database user for executing the NL2SQL queries to avoid any chance of destructive commands. The API should also have some rate limiting or authentication if it's a public service, to prevent abuse (like someone spamming heavy queries). These are standard web app security considerations.

To summarize, the front-end/back-end integration will result in a seamless experience: a user asks a question on the front-end, the back-end uses the LLM (with schema awareness) to generate SQL, executes it, and returns the answer. The modular design (separating the model, the executor, and the interface) ensures that each part can be improved or scaled independently. For instance, we could swap out the model for a more powerful one in the future without changing the front-end, or we could connect to a different database by just pointing the back-end to a new connection string and updating the schema context.

References and Resources:

- Spider Dataset (Yale) – a complex text-to-SQL benchmark with multiple databasesmedium.com.

- LangChain Documentation – for using `SQLDatabaseChain` and agents to link LLMs with databasesmedium.commedium.com.

- Hugging Face model *tscholak/t5-3b-finetuned-spider* – example of a T5 model fine-tuned on Spider that generalizes to unseen schemashuggingface.co.

- SQLGlot Library – a Python SQL parser/transpiler for converting SQL dialectscodecut.ai.

- Dataherald NL2SQL (GitHub) – an open-source project integrating LLMs with SQL DBsgithub.com.

- PremSQL (GitHub/Blog) – open-source local text-to-SQL pipelines focusing on small models and privacyblog.premai.ioblog.premai.io.

- "Text-to-SQL Benchmarks and SOTA" – overview of WikiSQL, Spider, and new challengesmedium.commedium.com.

- Microsoft NL2SQL Architecture Guide – discusses different architectures (few-shot vs fine-tuned vs agent-based) for NL→SQL appstechcommunity.microsoft.comtechcommunity.microsoft.com.

- PICARD (ServiceNow Research) – method for constrained decoding to ensure syntactically correct and schema-valid SQLhuggingface.cohuggingface.co.

These resources provide a wealth of guidance and examples to assist in building the schema-aware NL2SQL system from scratch. With a careful design following this plan, we can create an application that enables users to query databases in natural language, with the power of LLMs bridging the gap between human questions and SQL.