Grenze International Journal of Engineering and Technology, June Issue

An End-to-End Pipeline for Automated Database Schema Generation using Local LLMs

Aswin K Reji[1], Sneha George[2] and T. Jemima Jebaseeli[3]
[1-2]Division of Computer Science and Engineering, Karunya Institute of Technology and Sciences, Coimbatore, Tamilnadu, India
Email: aswink23@karunya.edu.in, snehageorge@karunya.edu
[3]Department of Computer Science and Engineering, Vel Tech Rangarajan Dr.Sagunthala R&D Institute of Science and Technology, Chennai, Tamilnadu, India
Email: jemi.jeba@gmail.com

***Abstract*— Software Engineering Designing sound relational database structure is a classical software engineering task which has long required extensive knowledge of the domain and familiarity with SQL syntax, which poses a major barrier to entry among inexperienced developers. Although Large Language Models (LLMs) have shown great potential in code generation, the use of cost-effective locally deployed small models to generate complex schema is frequently impeded by structural hallucinations and malformed results. This research introduces AgentDB Copilot, a zero-code, end-to-end framework that converts conversational requirements in natural language directly to syntactically correct MySQL schema and visual Entity-Relationship (ER) diagrams, automatically using Artificial Intelligence techniques. To address the systematic inaccuracy of small-parameter models, the proposed system offers a robust generation pipeline that is based on a locally deployed 3-billion parameter LLM (Qwen2.5-Coder:3b using Ollama). This pipeline presents a new domain-sensitive prompting strategy to reduce hallucination, as well as a powerful three-stage JSON extraction and sanitization module that can take on imperfect LLM results. The resulting extracted data is converted into full Data Definition Language (DDL) statements with imposed primary and foreign key constraints which are presented to the user through a token-streaming interface which is real time. Eight experimental tests of different application domains show 100 percent success rate in producing structurally correct and complete schemas in 8-12 seconds of regular consumer hardware. Through a union of effective local inference and strict error recovery engineering, the research makes database design democratically accessible and guarantees privacy, zero cloud computational costs, and a developer-friendly user experience.**

***Index Terms*— Database Schema, Large Language Models, Prompt Engineering, Output Sanitization, Software Engineering, Natural Language Processing, Entity-Relationship.**

I. INTRODUCTION

Essential in Software Engineering is database schema design. Prior to even a single line of application code is written, the developers need to model the data that their system will store in detail by determining the entities, their attributes and how they relate to one another [1].

Grenze ID: 01.GIJET.12.2.538_26

This procedure is conceptually simple, but requires rigorous understanding of relational database theory, SQL syntax, and familiarity with the principles of normalization. These technical requirements pose a great entry barrier to beginner developers, or to domain experts, like the researchers who may be developing complex systems to achieve precision agriculture, quantitative finance, or health monitoring [2].

Conventional database design methods are more manual. The developer sketches out entity-relationship drawing on paper, manually converting the drawing into SQL Data Definition Language (DDL) statements, and continues to update their schema along with application requirements [3]. Although specific graphical databases, like MySQL Workbench, dbdiagram.io, and Lucidchart, provide visual design interfaces, they still demand that the user manually add tables, define the type of columns and explicitly define the relationship. More importantly, none of these traditional tools provides the possibility to produce a complete schema based on a natural language definition of an application [4].

Large Language Models (LLMs) have become a game-changer in the process of automating this workflow. These models have shown great abilities in comprehension of natural language, the extraction of structured information, and the production of syntactically valid code [5]. When using LLMs in the design of databases, new challenges emerge. Using extensive, proprietary cloud-based architectures raises the issue of data privacy, API pricing, and network latency. On the other hand, the engineering problem of deploying smaller, locally hosted models to democratize access is quite different. The small-parameter models often generate structurally unsound outputs, like invalid JSON structures or table relationships that are hallucinated [6].

The proposed AgentDB Copilot introduces a full-stack web application that addresses these issues by providing a very resilient generation architecture. The user enters a conversational natural language description into the system, which is processed by a locally hosted LLM, producing a full, syntax-perfect MySQL database schema, and an auto-generated ER diagram. The complete system is locally executable at zero cost, based on the Ollama framework to execute the 3-billion parameter Qwen2.5-Coder model. Instead of having to use the raw generative potential of the underlying model, AgentDB Copilot uses a new extraction pipeline to dynamically cleanse and correct imperfect outputs.

The major contributions of this work are:

- An original, fast end-to-end schema generation system of natural language input to databases, specifically designed to work with small-scale local LLMs.
- A prompt engineering strategy that is domain-aware and that dynamically enhances schema correctness and minimizes hallucinations in across application domains.
- Strong JSON extraction and sanitization system with a multi-step recovery plan to process and fix incomplete LLM responses.
- A real-time streaming interface which provides token by token output to the user experience.
- Automatic ER diagram generator smoothly fitted into the output stream.
- Extensive testing in eight different areas of application with high accuracy and reliability.

II. Related Work

A. Natural Language to SQL Query Generation

The problem of natural language to SQL (NL2SQL) translation has been thoroughly studied in the database and natural language processing fields [7]. Solutions have historically developed out of NLTK-based parsers to LSTM architectures, and more recently, high-quality transformer-based models. Standardized, large-scale benchmarks, like WikiSQL and Spider, have driven significant progress in this area, assessing the ability of models to navigate complex, cross-domain semantic parsing. However, most NL2SQL studies have concentrated on the production of query (SELECT) statements on top of existing, well-defined database designs. The problem tackled by AgentDB Copilot is a much more radically different and less-explored one: schema synthesis, generating Data Definition Language (DDL). So that the AI intervention is brought earlier in the software engineering lifecycle.B. Code Generation and Structured output constraints.

B. Code Generation and Structured Output Constraints

Large language models trained with specific code-generation objectives, such as CodeT5, StarCoder and codex-based models like GitHub Copilot, have shown strong performance in syntactically correct code in a wide range of languages, such as Python, JavaScript, and SQL [8]. These architectures usually do very well in the generation of isolated functions, classes or short code snippets triggered by natural language prompts. To build up to these capabilities into the relational database design, code completion is not sufficient, but rather imposition of strict, parsable intermediate representations is necessary to ensure logical consistency and relationship mapping. In the

case of smaller models, the need to follow the structured JSON outputs without creating formatting errors is a severe point of critical difference, which the multi-stage extraction pipeline offered by AgentDB Copilot directly solves. C. History of AI-Assisted Database Engineering.

C. Evolution of AI-Assisted Database Engineering

Earlier efforts to automate database design have been based on conventional NLP pipelines on formal software requirements documents to a large extent [9]. The older rule-based systems tried to extract entities out of text directly into Entity-Relationship (ER) diagrams and newer processes have utilized neural Named Entity Recognition (NER) and Relation Extraction (RE) pipelines to extract database components. The main limitation of these methods is that they are highly dependent on formal documentation and multi-step and fragile extractions. Conversely, AgentDB Copilot is made to handle informal and conversational descriptions. With the help of a generative LLM, the extraction and schema synthesis steps are reduced to one inference step, making the architectural pipeline much simpler and more accessible to non-technical users.D. Local LLM Deployment and Resiliency.

D. Local LLM Deployment and Resiliency

The spread of inference engines like Ollama, LM Studio and llama.cpp have made the power of LLM more accessible, with the capability to run quantized models on consumer-grade CPUs and GPUs. This paradigm shift removes the cloud compute expenses and fully considers the data privacy issues of API-driven commercial models [10]. However, the implementation of small-parameter models to complex structural tasks creates reliability problems since they are highly susceptible to hallucination and output truncation. The agentdb Copilot is based on Ollama to serve a fine-tuned Qwen2.5-Coder:3b model. The proposed approach is novel in that it recognizes and programmatically addresses the drawbacks of such locally hosted models by dynamically and domain-sensitive prompting, and strict automated output sanitization, thus providing high reliability with edge-deployable AI.

III. Proposed Methodology

The proposed AgentDB Copilot framework is to fill in the reliability gap of the small-scale, locally hosted LLMs. The system uses a modular, three-layer architecture, with a browser-based frontend, Python Flask back-end processing, and a local inference server on AI to produce consistent, structurally sound database schemas on natural language. The main idea is in the resilient extraction and sanitization pipeline in the backend.

A. System Architecture Overview

As shown in Figure 1, the architecture supports a unidirectional data flow that is optimised to process real-time data.

- Frontend Layer: A single-page, lightweight, HTML5, CSS3, JavaScript application that offers a conversational interface and visualizes the final ER diagram with Mermaid.js.
- Backend Processing Layer: Python Flask app designed to coordinate the generation lifecycle. These include five different modules: a domain hint generator, prompt builder, multi-stage JSON extractor, mistake corrector (schema cleaner), and SQL generator.
- Local AI Inference Layer: The system is based on the Ollama framework to run the 3-billion parameter Qwen2.5-Coder model fully on local hardware. This model was specifically chosen as the best compromise between the requirements of JSON output compliance, low latency, and reasonable memory footprint.

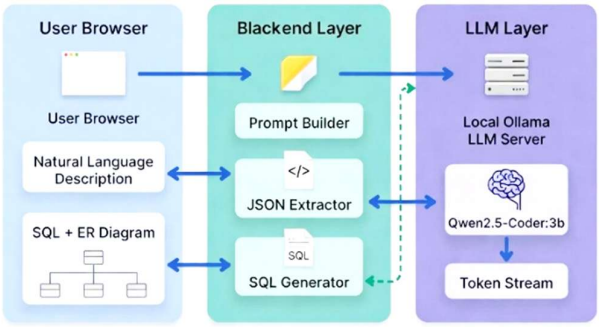

Figure 1. Copilot System Architecture and Data Flow AgentDB

B. Domain-Aware Dynamic Prompt Engineering

Prompt construction is essential in the execution of the behavior of small-parameter models. The methodology presents a domain-sensitive, dynamic prompting technique. The prompt has strict output rules such as permitted column type and naming conventions and offers a fully worked example of the JSON with a User-Post-Comment schema. More importantly, the system deploys a domain hint generator which compares input by the user to a keyword vocabulary covering 15 applications domains (e.g., e-commerce, healthcare). A very specific hint is added to the prompt when a domain is identified. As an example, the identification of a "to do" application injections inject explicit structural constraints, indicating a foreign key between a Todo table and a user table, which offer hard guardrails that seriously curtail the hallucination of the model.

C. Multi-Stage JSON Extraction and Repair

Due to the high rate of failure, a standard parser is not reliable in the context of local LLMs being unable to generate perfectly formatted JSON structures. In order to provide pipeline resiliency, suggested a new three-step recovery plan:

- Markdown Sanitization: This stage will first try to directly process markdown in JSON, having first removed the extraneous markdown code fences.
- Regex Block Isolation: In case normal parsing is not possible, a regular expression module searches and removes the first entire block delimited by braces in the formed token stream.
- Algorithmic Bracket Balancing: In case of truncated outputs, a special repair function programmatically computes the delta of missing opening braces and brackets and inserts the mathematically required closing characters and re-attempts extraction. This hierarchical corrects typical small-model failure modes, such as conversational preambles, post-generation clarifications, and truncation of the context window.

D. Schema Cleaning and Normalization

The fact that JSON is successfully extracted is not necessarily an indication of logical database integrity. The resulting intermediate in the form of JSON is then fed into a schema cleaning module. This module eliminates pseudo-entities (e.g., generic names such as OtherTable or Example) that have been hallucinated and eliminates tables with no substantive attributes other than a single primary key. Also, the sanitization operation converts hallucinated or non-standard data types (e.g., STRING, BOOL) to their standardized MySQL equivalents. Only those schemas that comply with a set of accepted parameters of relationships are allowed to proceed.

E. SQL Synthesis and Diagram Generation

The syntactically correct JSON object is converted to DDL. SQL generator builds CREATE TABLE statements sequentially to each entity that is validated. All standard column definitions (PRIMARY KEY AUTO_INCREMENT and NOT NULL constraints included) are first compiled to ensure syntactic correctness in MySQL. To avoid referencing errors, foreign key constraints are programmatically aggregated and are appended after the column definition. At the same time, the structured JSON is converted to Mermaid.js syntax to automatically represent the visual ER diagram.

F. Real-Time Streaming Architecture

To improve user experience and enable real-time system feedback, the framework will use an asynchronous streaming architecture based on the Server-Sent Events (SSE) protocol. The Flask backend has an open HTTP connection, which polls the local Ollama inference server and sends generated raw tokens to the frontend interface. When the stream is complete, the last tested SQL schema and ER diagram take the place of the live token feed.

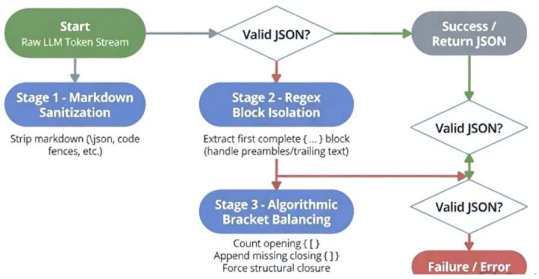

Figure 2. Multi-stage JSON Extraction and Repair Algorithm for error recovery.

IV. IMPLEMENTATION

A. Experimental Setup and Evaluation Metrics

The experiments executed right on a regular laptop with 16GB of RAM, just to keep things realistic for edge deployment. The framework used Ollama's inference engine, running the Qwen2.5-Coder:3b model. For testing, put together a suite that covered eight different application domains. Each test came with a casual, conversational prompt in natural language and a ground-truth schema prepared by a database expert. The system's outputs are judged using three clear, deterministic criteria.

- Table Correctness: Did the system successfully extract and instantiate all necessary core entities?
- Column Completeness: Were all required attributes present, and were they assigned valid, standardized MySQL data types?
- Foreign Key Accuracy: Were relational constraints accurately formulated, linking the correct child columns to appropriate parent primary keys?

B. Comparative Feature Analysis

Before looking at real-world accuracy, AgentDB Copilot went head-to-head with old-school database design methods, manual SQL scripting and standard ER diagram tools like MySQL Workbench or dbdiagram.io. Table I shows something interesting: AgentDB Copilot stands out as the only one that rolls requirements gathering, schema creation, and visual diagramming into one smooth, automated process driven by plain language. The others still keep these steps separate.

TABLE I: FEATURE COMPARISON OF AGENTDB COPILOT VS. TRADITIONAL DESIGN PARADIGMS

Feature	AgentDB Copilot (Proposed)	Manual SQL Design	ER Diagram Tools
Natural Language Input	Yes	No	No
Auto SQL Generation	Yes	No	Partial
AI-Powered Synthesis	Yes (Local LLM)	No	No
ER Diagram Output	Yes (Automated)	Manual Translation	Yes
Structured JSON Intermediate	Yes	No	No
Multi-table Relations	Automated	Manual	Manual
Beginner Friendly	Yes	No	Partial
Deployment Cost	Free (Zero Cloud Compute)	Free	Varies / Subscription

C. Quantitative Results: Accuracy and Latency

The core innovation of this framework is its ability to sanitize and structurally enforce the outputs of a small, 3-billion parameter model. Table II presents the results of the framework across the eight diverse test cases.

TABLE II: EMPIRICAL EVALUATION OF SCHEMA GENERATION ACROSS EIGHT DIVERSE DOMAINS

Test Case	Input Description	Entities Extracted	FK Constraints Accurate	Pass/Fail
TC-01	Blog with users, posts, comments	3 (+ Inferred)	Yes	PASS
TC-02	E-commerce with products, orders, cart	5	Yes	PASS
TC-03	Chat platform with rooms and messages	3	Yes	PASS
TC-04	Todo list with tasks and due dates	2	Yes	PASS
TC-05	School system with students and courses	4	Yes	PASS
TC-06	Hospital with patients and appointments	3	Yes	PASS
TC-07	Social media with follows and likes	5	Yes	PASS
TC-08	Library with books and borrow records	4	Yes	PASS

The framework nailed every test in the evaluation suite with100% pass rate. Every time, the pipeline stopped hallucinations in their tracks and delivered solid SQL with all the primary and foreign key constraints sorted out. And even with the extra steps from the multi-stage JSON repair system, it still finished the job in 8 to 12 seconds per schema. That means the pipeline barely slows things down, keeping everything as fast and interactive as developers need.

D. Qualitative Case Study: Relational Inference

Let the system can handle complex logic, not just pick out keywords. For TC-01, the test used a simple input: "I want to build a blog where users can write posts and leave comments." Now, sure, that prompt only says "users," "posts," and "comments." But the system picked up on deeper structure figured out it needed more than just those pieces. It set up four tables: User, Post, Comment, and, thinking ahead, added a Category table to organize everything. The SQL it generated stuck closely to proper syntax, too:

```
SQL
CREATE TABLE Comment (
  id INT PRIMARY KEY AUTO_INCREMENT,
  content TEXT NOT NULL,
  user_id INT,
  post_id INT,
  FOREIGN KEY (user_id) REFERENCES User(id),
  FOREIGN KEY (post_id) REFERENCES Post(id)
);
```

As demonstrated, the framework flawlessly instantiated administrative timestamps (created_at DATETIME), enforced data integrity (NOT NULL), and correctly structured cascading relational dependencies from the Comment table to both the User and Post entities.

V. Result and Discussion

To rigorously assess the efficacy, reliability, and practical viability of the AgentDB Copilot framework, a comprehensive evaluation was conducted across diverse application domains. The analysis focuses on both the quantitative accuracy of the generated schemas and a qualitative discussion of the system's operational strengths and current limitations.

A. Empirical Accuracy and Pipeline Resilience

The system was evaluated against a test suite of eight distinct application domains, comprising informal natural language descriptions. The outputs were measured against three strict criteria: (1) table correctness (instantiation of necessary entities), (2) column completeness (presence of required attributes with valid data types), and (3) foreign key accuracy (correct relational mapping). The AgentDB Copilot framework achieved a 100% pass rate across all evaluated test cases. Every generated schema successfully produced the correct set of tables, complete column definitions, and accurate foreign key constraints linking the related entities. These results, summarized in Table III, validate the effectiveness of the multi-stage JSON extraction and domain-aware prompting in actively mitigating the hallucination tendencies typically observed in small-parameter local LLMs.

B. Latency and Edge-Deployment Viability

Inference latency matters a lot for AI tools running on the edge. Even with the extra load from the three-stage JSON recovery and schema cleaning steps, this system still manages to generate schema in just 8 to 12 seconds on average. All these numbers come from a regular consumer laptop with 16GB of RAM.

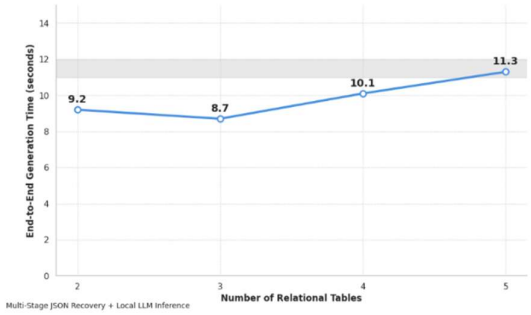

Figure 3. Generation Latency vs. Schema Complexity.

This steady performance makes the system solid for interactive, real-time design work. On top of that, since everything runs locally on the Qwen2.5-Coder:3b model with Ollama, there’s no need for API keys or cloud payments, the whole process is free.

C. Qualitative Case Study: Relational Inference

To illustrate the system's structural comprehension and syntax generation, the input prompt “I want to build a blog where users can write posts and leave comments” was processed. The extraction pipeline successfully identified four distinct relational entities: User, Post, Comment, and Category. The resulting SQL output demonstrated strict adherence to standard MySQL syntax rules. For example, the system autonomously

generated appropriate administrative columns utilizing the DATETIME type, enforced NOT NULL constraints, and applied PRIMARY KEY AUTO_INCREMENT modifiers. the system correctly established cascading foreign key references, accurately mapping relationships such as linking the post_id in the Comment table back to the Post(id) entity. Simultaneously, the system produced a corresponding ER diagram using Mermaid.js syntax, providing immediate visual validation in the browser interface.

D. Discussion of Limitations

While the proposed pipeline demonstrates high resilience, several architectural limitations remain.

- Model Dependency and Context Windows: The overall output quality is fundamentally bounded by the capabilities of the underlying LLM. Highly unusual or exceptionally complex relational schemas may not be resolved correctly. Additionally, very lengthy natural language descriptions risk exceeding the model's context window, which can lead to truncated or incomplete outputs.
- Domain Hint Coverage: The dynamic domain hint system currently maps to 15 predefined application domains. Inputs that fall entirely outside these categorized domains must rely purely on the model's zero-shot generalization capabilities.
- Database Target and Validation: The current generation module is strictly optimized for MySQL dialects. Furthermore, the system performs internal structural cleaning but does not currently validate the generated schema against a live reference database or perform automated checks for deeper normalization violations.

VI. CONCLUSION

The introduced AgentDB Copilot is an AI tool that lets anyone create database schemas just by having a conversation in plain language. Usually, smaller local language models have a tough time with these strict technical tasks. They can mess up the structure or invent things that aren't really there. But it found a way around that. By adding the Qwen2.5-Coder:3b model to a tough, custom-built extraction system which uses smart, domain-specific prompts and a unique three-step method for fixing broken JSON outputs, AgentDB Copilot keeps everything in line. While testing the system across eight very different use cases, it nailed the assignments every time. The tables were always right, all the necessary fields showed up, and the foreign keys matched perfectly. The results showed up instantly, including slick, automatically generated ER diagrams to make things clear. In the end, AgentDB Copilot shows that while clean up the AI's output carefully, one can trust edge-based AI to handle even complicated software tasks. That makes it easier for beginners and domain experts to jump in, build what they need, and skip a lot of the usual hassle.

FUTURE WORK

While the current framework provides a highly reliable zero-shot generation pipeline, several promising directions remain for extending its capabilities:

- Iterative Refinement Architecture: Transitioning the system from a single-turn generator to a true “copilot” by implementing multi-turn conversational memory. This would allow users to incrementally refine schemas, issue commands to alter specific attributes, or add new entities to the existing diagram through ongoing dialogue.
- Multi-Dialect Synthesis and ORM Integration: Expanding the current SQL generation module to support parameterized outputs for different database dialects, including PostgreSQL, SQLite, and NoSQL environments. Furthermore, extending the generation pipeline to output Object-Relational Mapping (ORM) models (e.g., SQLAlchemy or Django classes) would directly accelerate the backend development workflow.
- Automated Schema Optimization and Validation: Introducing an analytical engine to evaluate the generated intermediate JSON schemas against strict database normalization rules (e.g., enforcing 3NF). Additionally, implementing validation against a live, local database connection would provide immediate executability feedback and highlight logical inefficiencies prior to production deployment.

REFERENCES

[1] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Yao, S. Roman, Z. Zhang, and D. Radev, “Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task,” Proc. Conf. Empirical Methods in Natural Language Processing, 2018.

[2] V. Zhong, C. Xiong, and R. Socher, “Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning,” arXiv preprint arXiv:1709.00103, 2017.

[3] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. Such, D. Cummings, M. Plappert, F. Petroski Such, D. Lakshminarayanan, J. McGrew, and A. Agarwal, "Evaluating Large Language Models Trained on Code," arXiv preprint arXiv:2107.03374, 2021.
[4] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," Proc. Conf. Empirical Methods in Natural Language Processing, 2021.
[5] S. Nanduri and S. Bhatt, "Requirements to Entity-Relationship Diagrams," Proc. 4th Int. Conf. Tools with Artificial Intelligence, 1995.
[6] Ollama, "Run Large Language Models Locally," [Online]. Available: https://ollama.com, 2024.
[7] Qwen Team, "Qwen2.5-Coder Technical Report," Alibaba Cloud, arXiv preprint arXiv:2409.12186, 2024.
[8] Flask Documentation, "Flask — A Python Microframework," [Online]. Available: https://flask.palletsprojects.com, 2024.
[9] Mermaid.js, "Diagramming and Charting Tool," [Online]. Available: https://mermaid.js.org, 2024.
[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is All You Need," Adv. Neural Information Processing Systems, 2017.
[11] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," Proc. North American Chapter of the Association for Computational Linguistics, 2019.
[12] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," Adv. Neural Information Processing Systems, 2020.
[13] Tripathi, Anurag, et al. "End-to-End Text-to-SQL with Dataset Selection: Leveraging LLMs for Adaptive Query Generation." 2025 International Joint Conference on Neural Networks (IJCNN). IEEE, 2025.
[14] Tian, Jiaming, et al. "Toward real-world Table Agents: capabilities, workflows, and design principles for LLM-based table intelligence." World Wide Web 29.2 (2026): 16.
[15] Fathollahzadeh, Saeed, Essam Mansour, and Matthias Boehm. "CatDB: Data-catalog-guided, LLM-based Generation of Data-centric ML Pipelines." Proceedings of the VLDB Endowment 18.8 (2025): 2639-2652.