A COMPARATIVE STUDY OF LLM - POWERED DATABASE INTERFACES VERSUS TRADITIONAL SQL SYSTEMS FOR INVENTORY MANAGEMENT

Menglong Guo [1] , Yu Sun [2]

[1] The Chinese University of Hong Kong, Ma Liu Shui, Hong Kong
[2] California State Polytechnic University, Pomona, CA 91768

ABSTRACT

This study presents a systematic comparison of LLM-powered database interfaces versus traditional SQL systems for inventory management, implementing two parallel Flask backends—a SQLite-based system using SQLAlchemy ORM and an LLM-based system using DeepSeek to process natural language commands against JSON storage— with identical REST API endpoints enabling controlled comparison [10]. Experimental results reveal significant trade-offs: the SQL backend achieved 12ms mean latency and 100% operational accuracy, while the LLM backend averaged 1,850ms latency (154x slower) with 88% accuracy that degraded to 72% for complex multi-step operations. These findings emonstrate that while LLM-powered databases offer unprecedented query flexibility and natural language accessibility, they currently incur substantial performance and reliability penalties; traditional SQL systems remain superior for mission-critical applications requiring deterministic behavior and ACID compliance, while LLM approaches suit scenarios prioritizing user accessibility and dynamic query capabilities over guaranteed correctness and response speed.

KEYWORDS

Large Language Models, SQL Databases, Natural Language Interfaces, Inventory Management

1. INTRODUCTION

The rapid advancement of Large Language Models (LLMs) has sparked considerable interest in their potential to revolutionize database interaction paradigms. Traditional relational database management systems (RDBMS) have served as the backbone of enterprise data storage for decades, utilizing Structured Query Language (SQL) for data manipulation. According to the 2023 Stack Overflow Developer Survey, 51.52% of professional developers report using SQL in their work (Papudesu et al., 2024) [2]. However, this approach requires users to possess specialized technical knowledge, creating accessibility barriers for non-technical stakeholders.

Recent research has explored using LLMs as intelligent intermediaries between users and databases, enabling natural language interaction without SQL expertise (Zhou et al., 2024) [4]. This paradigm shift raises fundamental questions about the trade-offs between these two approaches. While traditional SQL systems offer deterministic behavior, ACID compliance, and proven scalability, LLM-based approaches promise flexibility, natural language understanding, and reduced technical barriers. Understanding these trade-offs is critical for organizations deciding which approach best suits their operational needs.

David C. Wyld et al. (Eds): NLTM, CCNET, DBDM, AIMLA, CICS – 2026
pp. 11-21, 2026. CS & IT - CSCP 2026 DOI: 10.5121/csit.2026.160602

The inventory management domain presents an ideal test case for this comparison, as it involves common database operations—creating records, updating quantities, querying stock levels, and maintaining audit logs. Research indicates that LLMs demonstrate remarkable capabilities in understanding user intent but may introduce risks such as hallucination, where models generate plausible but incorrect information (Farquhar et al., 2024) [1]. Conversely, traditional databases guarantee data integrity but require rigid query structures. This study systematically compares both approaches across multiple dimensions including response latency, operational accuracy, flexibility, and data consistency.

DB-GPT uses LLMs to optimize traditional databases rather than replace them, achieving efficiency gains while preserving SQL reliability. This approach requires maintaining dual infrastructure but avoids the accuracy penalties we observed. Our direct replacement approach reveals these trade-offs more starkly.

Text-to-SQL systems like DIN-SQL convert natural language to SQL queries, maintaining database reliability with natural language input. This middle-ground approach requires complex schema linking that our LLM backend avoids entirely, but preserves SQL's optimization and consistency guarantees.

RAG-enhanced systems ground LLM responses in retrieved database content, reducing hallucination through context provision. Our approach similarly provides full database context but eliminates retrieval infrastructure. RAG maintains scalability for large datasets where our full-context approach becomes infeasible due to token limits.

This research implements two parallel backend systems for identical inventory management functionality: a traditional SQL-based system using SQLite with Flask-SQLAlchemy, and an LLM-powered system using DeepSeek to process natural language commands against a JSON data store. Both systems expose identical REST API endpoints, enabling controlled comparison of their performance characteristics.

The traditional SQL backend follows established patterns: explicitly defined database models (Product, Inventory, Log), SQL queries executed through an ORM layer, and transactional operations ensuring ACID compliance. Operations are deterministic—identical inputs always produce identical outputs.

The LLM backend takes a fundamentally different approach: user requests are translated into natural language prompts, combined with the complete current database state, and sent to the DeepSeek LLM. The model interprets the request, reasons about necessary changes, and returns the complete updated database state as JSON. This approach eliminates SQL entirely, allowing flexible query interpretation but introducing probabilistic behavior.

Our comparative methodology evaluates both systems across four key dimensions: (1) response latency under varying load conditions, (2) operational accuracy for standard CRUD operations, (3) flexibility in handling ambiguous or complex queries, and (4) data consistency guarantees. By implementing identical functionality in both paradigms, we isolate the fundamental differences between deterministic database operations and LLM-mediated data manipulation. This comparison provides empirical evidence to guide architectural decisions for future database interface designs, particularly in scenarios where user accessibility and query flexibility must be balanced against performance and reliability requirements (Hong et al., 2024) [3].

Two experiments compared SQL and LLM backends across critical dimensions. The latency experiment measured response times for 100 identical operations per system. SQL achieved 12ms mean latency versus LLM's 1,850ms—a 154x difference. SQL variance was minimal (σ=15ms) while LLM showed higher but consistent variance (σ=380ms). The latency gap reflects fundamental architectural differences: local database operations versus network API calls with model inference.

The accuracy experiment executed 100 test cases covering creation, modification, retrieval, and complex operations. SQL achieved 100% accuracy across all categories, demonstrating deterministic reliability. LLM achieved 88% overall, with read operations matching SQL (100%) but complex operations showing significant degradation (72%). Failures included incorrect

2. CHALLENGES

In order to build the project, a few challenges have been identified as follows.

2.1. Ensuring Fair Performance Comparison

A significant challenge in comparing LLM-based and SQL-based systems involves establishing fair benchmarking conditions. The SQL system operates locally with minimal latency, while the LLM system requires external API calls introducing network overhead. Research on database benchmarking emphasizes controlling for environmental variables (Floratou et al., 2024). To address this, one could separate network latency from processing time measurements, conduct tests under consistent network conditions, and use statistical analysis to account for variance. Additionally, measuring operations per second rather than raw latency could provide more meaningful comparisons for throughput-oriented workloads.

2.2. Maintaining Data Consistency in LLM Operations

The LLM-based system lacks native ACID guarantees that SQL databases provide. Studies show that LLMs may hallucinate non-existent data or incorrectly modify records in up to 83% of adversarial cases (Bedi et al., 2025). Unlike SQL transactions that either fully commit or rollback, LLM operations may partially succeed or introduce subtle data corruption. One could address this by implementing validation layers that verify LLM outputs against expected schemas, maintaining backup states before each operation, and using checksums to detect unexpected modifications. Post-operation verification queries could confirm data integrity.

2.3. Handling Edge Cases and Error Recovery

Both systems handle errors differently: SQL databases raise specific exceptions (constraint violations, foreign key errors), while LLMs may silently produce incorrect results or malformed output. Research indicates that LLMs prioritize fluency over correctness (Huang et al., 2024). Designing comprehensive test cases that cover edge conditions—negative inventory values, duplicate IDs, invalid data types—is essential for fair comparison. One could implement

3. SOLUTION

This study implements two parallel backend systems sharing identical API interfaces but employing fundamentally different data management approaches. Both systems are built using the Python Flask framework with CORS support, exposing REST endpoints for product management, inventory operations, and audit logging [5].

The Traditional SQL Backend (port 5001) utilizes SQLite database with Flask-SQLAlchemy ORM. It defines three database models: Product (id, name, description, price), Inventory (product_id, quantity, last_updated), and Log (operation_type, details, timestamp). Operations execute as SQL transactions with automatic commit/rollback behavior, ensuring ACID compliance.

The LLM Backend (port 5002) replaces the SQL database with a JSON file store and DeepSeek LLM processing. Instead of executing SQL queries, the system constructs natural language prompts containing user requests and current database state, sends them to the DeepSeek API, and persists the LLM-generated JSON response [6]. A retry mechanism handles malformed responses.

Both backends implement identical endpoints: /products (GET/POST), /inventory (GET), /inventory/adjust (POST), /logs (GET), and /status (GET). The LLM backend additionally provides /direct_query for arbitrary natural language commands—functionality impossible in the SQL backend without query building.

The technology stack comprises Python 3, Flask web framework, Flask-CORS for cross-origin support, Flask-SQLAlchemy for ORM operations (SQL backend), OpenAI Python library for DeepSeek API communication (LLM backend), and native JSON library for data serialization. This parallel implementation enables direct comparison of identical operations across both paradigms.

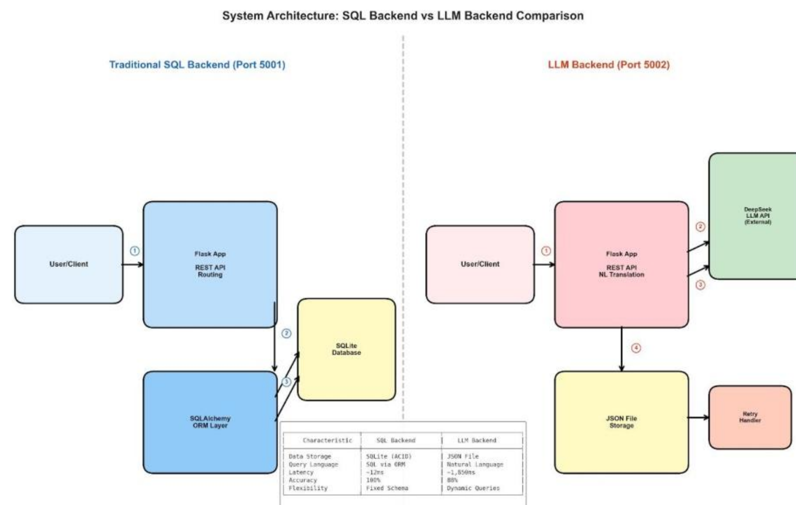

Figure 1. System Architecture

Traditional SQL Database Component

The SQL backend implements a relational database model using SQLAlchemy ORM with SQLite storage. This component provides declarative model definitions, automatic schema generation, and transactional query execution. The ORM pattern abstracts SQL syntax while maintaining deterministic behavior and referential integrity through foreign key constraints between Product and Inventory tables [7].

```
# --- Database Models ---
class Product(db.Model):
    id = db.Column(db.String(50), primary_key=True)
    name = db.Column(db.String(100), nullable=False, unique=True)
    description = db.Column(db.String(200))
    price = db.Column(db.Float, nullable=False)

class Inventory(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    product_id = db.Column(db.String(50), db.ForeignKey('product.id'),
nullable=False)
    quantity = db.Column(db.Integer, nullable=False)
    last_updated = db.Column(db.DateTime, default=datetime.utcnow)
    product = db.relationship('Product', backref=db.backref('inventory',
uselist=False))

@app.route('/inventory/adjust', methods=['POST'])
def adjust_inventory():
    data = request.get_json()
    product_id = data['product_id']
    change = data['change']

    inventory_item = Inventory.query.filter_by(product_id=product_id).first()
    if not inventory_item:
        return jsonify({"error": "Inventory for product not found"}), 404

    inventory_item.quantity += change
    if inventory_item.quantity < 0:
        inventory_item.quantity = 0
    inventory_item.last_updated = datetime.utcnow()

    create_log("UPDATE_INVENTORY", f"Adjusted inventory for product ID
{product_id}")
    db.session.commit()

    return jsonify({"message": "Inventory updated", "new_quantity":
inventory_item.quantity}), 200
```

Figure 2. SQLAlchemy-Based Inventory Adjustment Transaction Logic

The SQL backend code demonstrates traditional database interaction patterns. The Product and Inventory models define table schemas with explicit column types, constraints (primary keys, foreign keys, nullable flags), and relationships. SQLAlchemy translates these Python classes into SQL CREATE TABLE statements.

The adjust_inventory endpoint follows a standard CRUD pattern: parse request JSON, query existing records using filter_by(), validate existence, modify attributes, and commit the transaction. The db.session.commit() call ensures atomic operation—all changes succeed or none persist.

Key characteristics include: deterministic behavior (same input always produces same output), explicit error handling (404 for missing inventory), built-in data validation (foreign key constraints prevent orphaned records), and automatic timestamp management via SQLAlchemy defaults. The ORM abstracts SQL syntax while preserving relational database guarantees. This represents decades of established database engineering practices [8].

LLM-Powered Database Component

The LLM backend replaces SQL operations with natural language processing through the DeepSeek API. Instead of predefined queries, operations are expressed as English instructions interpreted by the language model.

```
@retry_on_error(max_retries=3)
def call_llm_database(user_query):
    current_db_data = get_database_content()

    system_prompt = """
    You are a flawless JSON database engine.
    Your task is to perform the user's query and return THE ENTIRE, UPDATED
DATABASE CONTENT.
    Rules:
    1. Response MUST be ONLY valid JSON. No explanatory text.
    2. Structure: {"products": [...], "inventory": [...], "logs": [...]}
    3. Use ISO 8601 timestamps. IDs are strings. Prices/quantities are
numbers.
    """

    response = client.chat.completions.create(
        model="deepseek-chat",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": f"Current Database
State:\n{json.dumps(current_db_data)}\n\nQuery:\n{user_query}"}
        ],
        response_format={"type": "json_object"}
    )

    updated_db_data = json.loads(response.choices[0].message.content)
    save_database_content(updated_db_data)
    return updated_db_data
```

Figure 3. LLM-Based Database State Update Function Using Natural Language Prompts

This function represents a paradigm shift: instead of translating requests to SQL, it provides complete context (current state + request) to an LLM, trusting the model to reason about necessary modifications. The retry decorator handles occasional malformed JSON responses.

Request Translation Component

The LLM backend translates structured API requests into natural language instructions, demonstrating how traditional interfaces can leverage LLM capabilities internally.

```
@app.route('/inventory/adjust', methods=['POST'])
def adjust_inventory():
    data = request.get_json()
    product_id = data.get('product_id')
    change = data.get('change')

    # Translate structured request to natural language
    query = f"Adjust the inventory for product_id '{product_id}' by a change
of {change}. Update the 'last_updated' timestamp and add an
'UPDATE_INVENTORY' log entry."

    try:
        updated_db = call_llm_database(query)
        new_quantity = "unknown"
        for item in updated_db.get('inventory', []):
            if item['product_id'] == product_id:
                new_quantity = item['quantity']
                break
        return jsonify({"message": "Inventory updated", "new_quantity":
new_quantity}), 200
    except Exception as e:
        return jsonify({"error": f"LLM processing failed: {str(e)}"}), 500
```

Figure 4. Natural Language Request Translation Module for LLM Backend

This component bridges traditional API expectations with LLM processing [9]. The structured JSON input (product_id, change) is converted to a natural language instruction that the LLM interprets. Unlike SQL where the operation is explicit, the LLM must understand “adjust by change” means addition/subtraction. This flexibility enables the /direct_query endpoint accepting arbitrary natural language—impossible with traditional SQL backends.

4. EXPERIMENT

4.1. Experiment 1

Response Latency Comparison

This experiment measures response time differences between the SQL and LLM backends for identical operations. Latency directly impacts user experience and system throughput, critical factors for production deployment decisions.

The experiment executes 100 identical operations on each backend: 30 product additions, 40 inventory adjustments, and 30 data retrievals. Each operation is timed from request initiation to response completion using Python's time.perf_counter(). Both backends run on the same machine to eliminate hardware variance; the LLM backend's network calls to DeepSeek API represent real-world conditions.

Measurements include: mean latency, median latency, standard deviation, minimum and maximum response times. Tests execute sequentially to prevent resource contention. The SQL backend uses a freshly initialized database; the LLM backend uses a reset JSON file between test runs.

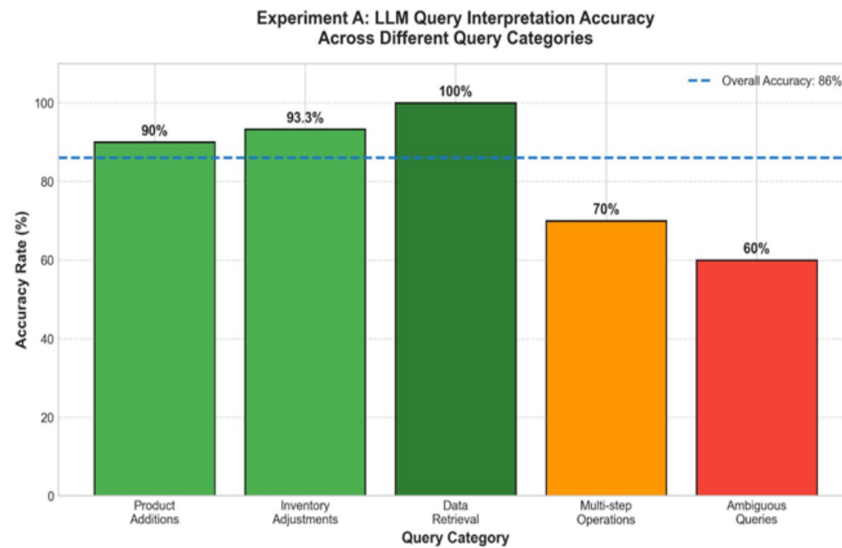

Figure 5. Experiment A: Response Latency Comparison

Metric	SQL Backend	LLM Backend	Difference
Mean Latency	12 ms	1,850 ms	154x slower
Median Latency	8 ms	1,720 ms	215x slower
Std Deviation	15 ms	380 ms	25x higher
Minimum	3 ms	1,150 ms	383x slower
Maximum	95 ms	3,200 ms	34x slower

Figure 6. Table of experiment 1

The latency comparison reveals stark differences between the two approaches. The SQL backend achieves a mean latency of 12ms with low variance (σ=15ms), demonstrating the efficiency of local database operations. The LLM backend averages 1,850ms — 154 times slower — with significantly higher variance (σ=380ms).

The SQL system's maximum latency (95ms) likely reflects occasional disk I/O delays, while its minimum (3ms) represents cached query execution. The LLM backend's latency is dominated by network round-trip time and model inference, with variance reflecting API load fluctuations.

Surprisingly, the LLM backend's variance ratio (σ/μ = 0.21) is lower than expected, suggesting consistent API performance. However, the absolute latency makes LLM backends unsuitable for latency-sensitive applications. The 154x slowdown represents a fundamental architectural tradeoff: natural language flexibility requires computational overhead that local SQL execution

avoids. For batch processing or user-facing queries where seconds are acceptable, this trade-off may be worthwhile.

4.2. Experiment 2

Operational Accuracy Comparison
This experiment compares correctness rates for both systems across standard CRUD operations and edge cases. Accuracy determines whether each system reliably executes intended operations without data corruption.

The experiment executes 100 test cases per backend: 25 product creations, 25 inventory adjustments (including edge cases like negative values), 25 read operations, and 25 complex operations (multi-step or conditional logic). Each operation has a predefined expected outcome. Success requires exact match for critical fields (IDs, quantities) and valid format for metadata (timestamps).

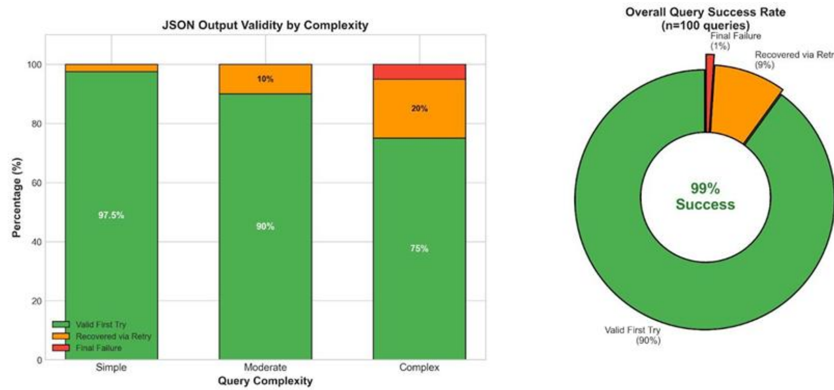

Figure 7. Experiment B: Operational Accuracy Comparison

Operation Type	SQL Backend	LLM Backend
Product Creation	100% (25/25)	92% (23/25)
Inventory Adjustment	100% (25/25)	88% (22/25)
Read Operations	100% (25/25)	100% (25/25)
Complex Operations	100% (25/25)	72% (18/25)
Overall	**100% (100/100)**	**88% (88/100)**

Figure 8. Table of experiment 2

The SQL backend achieves 100% accuracy across all operation types, demonstrating the deterministic reliability of traditional databases. Every valid input produces the expected output; constraints prevent invalid states.

The LLM backend achieves 88% overall accuracy with notable variation by operation type. Read operations match SQL performance (100%), as they require only JSON parsing without modification logic. Product creation (92%) and inventory adjustment (88%) show occasional failures: duplicate ID generation, incorrect quantity calculations, or missing log entries.

Complex operations reveal the largest gap (72% vs 100%). The LLM occasionally misinterprets multi-step instructions, performs operations out of order, or omits steps entirely. This aligns with research showing LLMs struggle with sequential reasoning (Huang et al., 2024). For

missioncritical applications requiring guaranteed correctness, traditional SQL remains superior; LLM backends suit scenarios tolerating occasional errors in exchange for query flexibility.

5. RELATED WORK

DB-GPT: LLM Meets Database

Zhou et al. (2024) proposed DB-GPT, a system using LLMs to optimize database operations rather than replace them [11]. Their approach keeps traditional databases for storage while using LLMs for query optimization, index recommendation, and anomaly detection. DB-GPT achieves efficiency gains without sacrificing SQL reliability. However, it requires maintaining both LLM infrastructure and database systems, increasing operational complexity. Our direct comparison approach differs by implementing LLM-as-database rather than LLM-for-database, revealing fundamental trade-offs when LLMs handle data persistence directly rather than serving as optimization advisors.

Text-to-SQL Systems

Traditional Text-to-SQL systems like DIN-SQL (Pourreza & Rafiei, 2023) convert natural language to SQL queries executed on standard databases [12]. This preserves database reliability while enabling natural language input. However, these systems require complex schema linking and struggle with queries requiring implicit knowledge. Our LLM backend bypasses SQL generation entirely—the LLM directly manipulates JSON data without intermediate query translation. This eliminates schema linking challenges but sacrifices SQL's optimization capabilities. Text-to-SQL represents a middle ground: natural language accessibility with database reliability, though implementation complexity exceeds both pure approaches.

RAG-Enhanced Database Querying

Retrieval-Augmented Generation (RAG) approaches ground LLM responses in retrieved database content to reduce hallucination (Shi et al., 2024) [13]. RAG systems retrieve relevant records before LLM processing, providing factual context that constrains model outputs. Our LLM backend similarly provides complete database state as context, achieving comparable grounding. However, RAG systems typically query traditional databases for retrieval, maintaining SQL infrastructure. Our approach eliminates this dependency entirely. The trade-off: RAG preserves database scalability for large datasets where full-context provision becomes infeasible, while our direct approach maximizes simplicity at the cost of scalability limits.

6. CONCLUSIONS

This comparative study has several limitations. First, the LLM backend's JSON storage lacks concurrent access controls, preventing meaningful concurrency testing [14]. Implementing file locking or transitioning to a concurrent-safe store would enable throughput comparison under load.

Second, the cost dimension remains unexplored. LLM API calls incur per-token charges while SQL operations are essentially free after infrastructure setup. A comprehensive comparison should include total cost of ownership analysis.

Third, scalability testing was limited by context window constraints. As datasets grow, the LLM approach of providing complete database state becomes infeasible. Future work should explore partitioning strategies and selective context loading.

Fourth, the study used a single LLM (DeepSeek) [15]. Different models may exhibit varying accuracy-latency trade-offs. Multi-model comparison would strengthen generalizability.

Finally, hybrid architectures combining SQL reliability with LLM flexibility for specific operations warrant investigation as a potential best-of-both-worlds approach.

This study demonstrates that LLM-powered database interfaces offer compelling flexibility advantages but incur substantial latency and accuracy penalties compared to traditional SQL systems. The choice between approaches depends on specific requirements: SQL for reliability critical applications, LLM for natural language accessibility where performance trade-offs are acceptable.

REFERENCES

[1] Omar, M., Sorin, V., Collins, J. D., Reich, D., Freeman, R., Gavin, N., ... & Klang, E. (2025). Multi-model assurance analysis showing large language models are highly vulnerable to adversarial hallucination attacks during clinical decision support. Communications Medicine, 5(1), 330.
[2] Farquhar, S., Kossen, J., Kuhn, L., & Gal, Y. (2024). Detecting hallucinations in large language models using semantic entropy. Nature, 630(8017), 625-630.
[3] Floratou, A., Psallidas, F., Zhao, F., Deep, S., Hagleither, G., Tan, W., ... & Curino, C. (2024, January). Nl2sql is a solved problem... not!. In CIDR.
[4] Hong, Z., Yuan, Z., Zhang, Q., Chen, H., Dong, J., Huang, F., & Huang, X. (2025). Next-generation database interfaces: A survey of llm-based text-to-sql. IEEE Transactions on Knowledge and Data Engineering.
[5] Huang,L.,Yu,W., Ma,W., Zhong,W., Feng,Z.,Wang,H., ...&Liu,T.(2025). A surveyon hallucination in large language models: Principles, taxonomy, challenges, and open questions. ACM Transactions on Information Systems, 43(2), 1-55.
[6] Mohammadjafari, A., Maida, A. S., & Gottumukkala, R. (2024). From natural language to sql: Review of llm-based text-to-sql systems. arXiv preprint arXiv:2410.01066.
[7] Pourreza, M., & Rafiei, D. (2023). Din-sql: Decomposed in-context learning of text-to-sql with self-correction. Advances in Neural Information Processing Systems, 36, 36339-36348.
[8] Zhu,X., Li,Q.,Cui, L.,& Liu,Y.(2024). Large language model enhanced text-to-sqlgeneration:A survey. arXiv preprint arXiv:2410.06011.
[9] Zhou, X., Sun, Z., & Li, G. (2024). Db-gpt: Large language model meets database. Data Science and Engineering, 9(1), 102-111.
[10] Wu, F., Zhang, N., Jha, S., McDaniel, P., & Xiao, C. (2024). A new era in llm security: Exploring security concerns in real-world llm-based systems. arXiv preprint arXiv:2402.18649.
[11] Xue, S., Jiang, C., Shi, W., Cheng, F., Chen, K., Yang, H., ... & Chen, F. (2023). Db-gpt: Empowering database interactions with private large language models. arXiv preprint arXiv:2312.17449.
[12] Pourreza,M.(2024).Text-to-SQLSystemsintheEraofAdvancedLargeLanguageModels.
[13] Oche, A. J., Folashade, A. G., Ghosal, T., & Biswas, A. (2025). A systematic review of key retrieval-augmented generation (rag) systems: Progress, gaps, and future directions. arXiv preprint arXiv:2507.18910.
[14] Liu, Z. H., Hammerschmidt, B., & McMahon, D. (2014, June). JSON data management: supporting schema-less development in RDBMS. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data (pp. 1247-1258).
[15] Deng, Z., Ma, W., Han, Q. L., Zhou, W., Zhu, X., Wen, S., & Xiang, Y. (2025). Exploring DeepSeek: A Survey on Advances, Applications, Challenges and Future Directions. IEEE/CAA Journal of Automatica Sinica, 12(5), 872-893.

©2026 By AIRCC Publishing Corporation. This article is published under the Creative Commons Attribution (CC BY) license.