Data-Oriented Design Integration Layer (DODIL): A Powerful Augmented Schema Framework for LLM-Powered Systems

Mohamed Rizk , Yousef Korayem , Wassim Alexan [*]

Posted Date: 22 January 2025

doi: 10.20944/preprints202501.1655.v1

Keywords: AI; API; applications; architecture; database; LLM

Article

Data-Oriented Design Integration Layer (DODIL): A Powerful Augmented Schema Framework for LLM-Powered Systems

Mohamed Rizk [1], Yousef Korayem [2] and Wassim Alexan [3,*]

[1] Circle Technologies PTE, Singapore, Singapore
[2] CSEN Department, Faculty of MET, German University in Cairo, Egypt
[3] Communications Department, Faculty of IET, German University in Cairo, Egypt
* Correspondence: wassim.alexan@ieee.org

Abstract: Large Language Models (LLMs) have emerged as powerful tools for natural language understanding and human-AI interaction. While LLMs excel at generating insights and responding to user queries, significant challenges arise when they are tasked with interacting with complex application data systems. Unlike basic database querying, real-world applications involve intricate architectures, non-standardized interfaces, and domain-specific semantics, making integration with LLMs a non-trivial task. This article highlights the key challenges faced by LLMs in effectively interfacing with modern application environments.

Keywords: AI; API; applications; architecture; database; LLM

---

1. Introduction

With the emergence of large language models like OpenAI's GPT and Meta's LLaMA, the state of application development in the past several years has changed drastically. Many modern applications seek to utilize these large language models - henceforth referred to as LLMs - as tools for advanced natural language processing and the facilitation of interaction between human users and AI [1]. LLMs are remarkably suited for these tasks and have seen an explosive rise in usage in recent years [2]. LLMs are extremely effective at generating insights and responding to user queries, but often fail when interacting with complex application data systems, making them difficult to adapt for each individual application developer's APIs and endpoints [3]. Database querying, which LLM integration is often likened to, rarely includes the challenges related to the complex architecture of unique applications and the semantics of each individual domain said applications may be involved in [4].

As it stands, the integration of applications with LLMs leaves something to be desired. Applications that rely on LLMs indirectly often require massive software overheads to ensure that their endpoints reliably interact with the models they utilize [5]. The reasons for this are multiple, including issues with data security and the management of the LLMs when taken as they are provided by their developers [6]. LLMs often have a tendency to misinterpret prompts or mutate data during queries, which requires additional software to manage and account for [7].

The research presented herein introduces a schema framework designed to mitigate most of the overheads and dangers that applications interacting with LLMs typically have to deal with, as well as streamline the additional software developed to handle those issues [8]. The schema, named Data-Oriented Design Integration Layer (DODIL), is an augmented data framework. Its main purpose is to accelerate the development of applications with a specific focus on those powered by LLMs. The framework integrates cleanly with any data source and supports any programming language.

This article is organized as follows: Section 2 discusses the key challenges that applications have to manage when interacting with LLMs. Section 3 proposes DODIL, the augmented schema framework. Section 4 discusses its key components. Finally, Section 5 concludes this article and suggests future research directions that could be further explored.

2. Key Challenges in Enabling LLMs to Interact with Application Data

A number of challenges quickly emerge in the face of any developer when attempting to interact with large language models during application development [8–11]. The four most impactful challenges are the following:

1. Diverse and Non-Standardized Interfaces
2. Context and Complexity of APIs
3. Lack of Shared Knowledge Representation
4. Safety, Reliability, and Operational Constraints

Each challenge comes with its own unique problems and possible solutions.

First, the diverse and non-standardized nature of interfaces. By definition, unique applications are unique: applications are built with unique structures, endpoints, and interaction patterns, often differing across systems and organizations. No universal protocol or standardized framework currently exists that LLMs can rely on to interpret and interact with arbitrary systems [12]. As such, teaching an LLM to comprehend every possible route, parameter, and authentication mechanism for each application is a laborious and error-prone process, which is also very much impractical at scale [13].

The context and complexity of application programming interfaces (APIs) is another major challenge. Real-world systems are often preconfigured to utilize APIs to interact with other applications and systems. These systems are often highly complex, involving multistep authentication mechanisms, rate-limiting policies, and deeply nested request-response structures [14]. APIs also usually include domain-specific vocabularies and semantics, requiring LLMs not only to identify available endpoints but also to understand their behavior, required parameters, and relationships with other parts of the system. This additional step requires additional configuration and computation from the LLM, which reduces efficiency and adds an additional overhead, making it a substantial challenge for developers. Figure 1 shows the typical integration of LLMs with application interfaces. Usual integration results in complex integration procedures that involve a large portion of the API, requiring extended developer involvement and more extensive software [15].

LLMs also suffer from an absence of shared knowledge representation systems [16]. Individual applications often rely individually on unique systems for representing data and storing information. Without a shared system to represent knowledge, each system's schema, data structures, and routes remain isolated, forcing LLMs to 'relearn' the specific language and logic of each new application they encounter. Without a standardized schema or an intermediary representation, LLMs cannot generalize knowledge or reasoning across systems, reducing efficiency and adaptability [17].

Finally, LLMs often suffer from issues related to safety, reliability, and operational constraints [18]. As models that work strictly on a language basis, LLMs do not inherently adhere to strict operational constraints that ensure safety, reliability, and compliance with business rules. When utilized as significant working parts of applications that do involve those strict constraints, LLMs then have to be managed to ensure that they do actually follow those rules. This results in additional overheads where developers build software to avoid unsafe or unintended operations, respect user-defined constraints, handle errors gracefully, and manage edge cases effectively [19]. Failure to enforce these constraints increases the risk of unsafe interactions, introducing reliability concerns within mission-critical systems. LLMs are powerful tools, indeed, but lack the regulation that hard-coded systems come designed with [20].

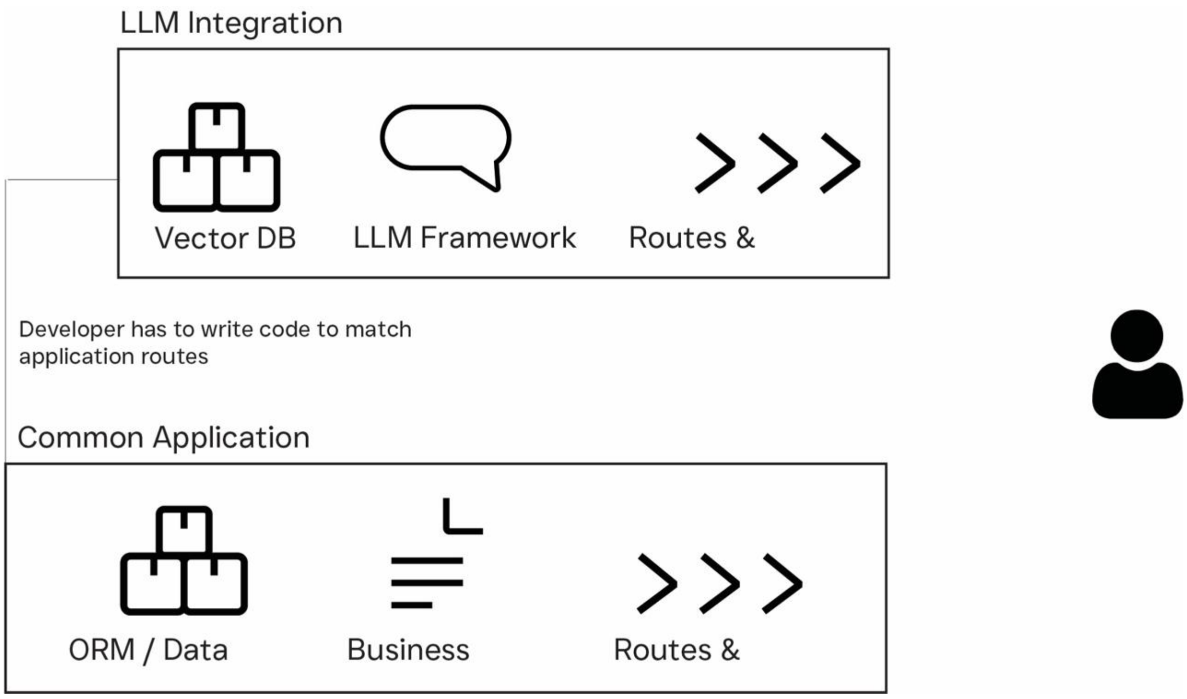

Figure 1. A simplified flowchart representing how most general applications get built and how LLM integration typically connects. LLMs will need to integrate with most of the API creating a complexity in integration.

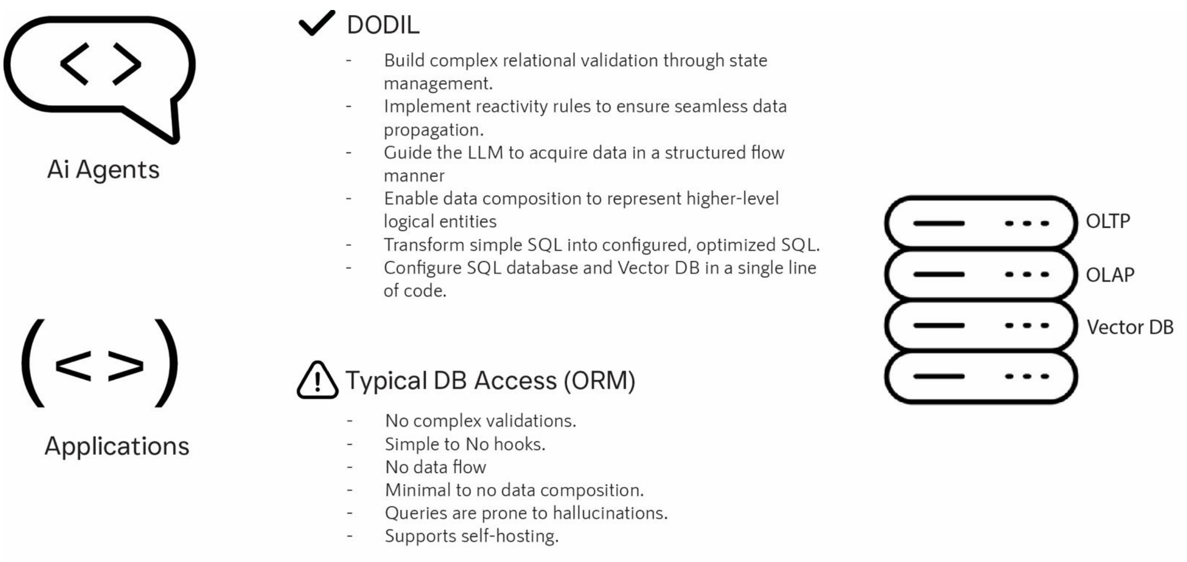
Ai Agents
Applications
DODIL
- Build complex relational validation through state management.
- Implement reactivity rules to ensure seamless data propagation.
- Guide the LLM to acquire data in a structured flow manner
- Enable data composition to represent higher-level logical entities
- Transform simple SQL into configured, optimized SQL.
- Configure SQL database and Vector DB in a single line of code.
Typical DB Access (ORM)
- No complex validations.
- Simple to No hooks.
- No data flow
- Minimal to no data composition.
- Queries are prone to hallucinations.
- Supports self-hosting.
OLTP
OLAP
Vector DB

Figure 2. The Smart Augmented Schemas (SAS) framework redefines schema management for modern applications by integrating multi-datastore support, advanced reactivity, relational state management, and access control. This enables real-time interaction between LLMs and applications, facilitating dynamic data manipulation and enriched application intelligence.

3. The Augmented Schema Framework: DODIL

All the aforementioned challenges can be tackled by employing a software integration framework. To that end, this article presents the Data-Oriented Design Integration Layer (DODIL) framework, a lightweight system capable of mitigating each of the challenges described in Section 2.

Unlike simple object-relation mapping (ORM) software, DODIL is a holistic-data framework that transforms how application models are designed and managed in an industry driven more and more by LLMs and other types of AI. DODIL is mainly programmed in Rust and offers support for

JavaScript and Python bindings, while integrating effortlessly with a wide range of data sources. DODIL is designed to allow for easy movement and migration of data with any language to any datastore including OLTP, OLAP and vector stores. It is programmed efficiently, requiring of the user less API code than when utilizing a competitive strategy. It is rule-driven and consistent, centralizing business constraints, validations, and reactive updates to create clean, maintainable architectures. It is also designed to effectively evolve with the user's software environment, scaling from a small size database application to a much larger, AI-driven, multi-service ecosystem. In essence, DODIL bridges the gap between AI prompts and real-world data, cutting down hallucinations and ensuring safe operations.

DODIL solves each of the challenges by tackling them both separately and in an integrated manner. The framework is designed as a number of modules, each module interacting with others but mainly being focused on a set of functionalities needed to ease and enhance the interaction between the LLM and application data.

4. Key Modules of the Augmented Schema Framework

DODIL has seven unique modules. These are:

1. Relation
2. Augment
3. Compose
4. State Management
5. Reactivity
6. Flow
7. Prompt Management

Each component will be discussed in detail, but it is important to note the way DODIL represents data. In DODIL, data is represented in three forms:

1. **Datum**
   This is the singular, basic data form. It represents a single data point represented by a single *Schema*.
2. **DataSet**
   This is a list of *Datum* from the same *Schema*, with predefined powerful batch and math operations.
3. **DataComposite**
   This is an augmented data form of different *Schemas* connected via *Relationships* with the Eager or Lazy augmentation methods.
   For example, a composite of $[Order, OrderItem, DeliveryAddress]$ when considering an application involving ordering certain items.

It should also be noted that the following subsections describe in general the functionality of each specific module. The specifics of how each module operates may be found here, at the DODIL official documentation webpage.

4.1. Relation

The first principle is that of *Relation*. In many data models, especially those involving document stores or relational databases, "parent-child" relationships (or "one-to-many" relationships) are foundational. For instance:

- An Order (parent) can have multiple Order Items (children)
- A Category (parent) can have multiple Subcategories (children)
- A User (parent) can have multiple Addresses (children)

DODIL provides a concise way to define these relationships by referencing one schema from within another using the following pattern:

$$\{type: ''id'', refer: ''SCHEMA\_NAME''\} \tag{1}$$

Each type of relationship can be uniquely approached and defined for the appropriate data using DODIL. Additionally, constraints can be established at the database level (assuming the underlying datastore supports unique indices).

4.2. Augment

Augment allows the merger or population of related data from one schema into another without the manual writing of complex queries or mapping logic. In essence, it changes a *Datum* to a *DataComposite* or enhances a *DataComposite* with more *Datum*. It can be considered a higher-level join that deals not only with database fields but also with data manipulations, relationships, and DODIL's internal Datum objects. Its main benefits are as follows:

1. Less Repetitive Code: Instead of manually querying child records, adding augmentations rule or calling an augmentation function, it can dynamically fetch related data.
2. Consistent Data Structures: Augmentation converts multiple related *Datums* into a single *CompositeData*, letting you treat them as one cohesive unit.
3. Flexible Fetching: Depending on performance and domain requirements, augmentation can be Eager (all in one go) or Lazy (on demand).

Eager and lazy are the two designated sub-types of augmentation procedures. Eager augmentation involves the fetching or population of all related data in a singular operation, a heavy "preload" of sorts. It is a singular request, making data readily available but can be potentially expensive if the relation is large. Lazy augmentation only fetches data on demand. It only augments children when actually needed. It saves resources, especially if many related records exist but are rarely accessed. It does, however, require multiple calls if ultimately all related data is required.

4.3. Compose

Compose is the process of aggregating and orchestrating data from multiple related schemas into a unified logical structure. It ensures data integrity, consistency, and completeness before performing mutations or persistence in a data store. By combining individual data points into a cohesive schema, it simplifies the creation, validation, and mutation of complex entities in a systematic and efficient manner.

For example, creating an order often requires combining data from various schemas like $ORDER$, $ORDERITEM$, $DELIVERY\_ADDRESS$, and PAYMENT. DODIL's Compose feature streamlines this process by ensuring that every step in this multi-dimensional data collection is handled efficiently, consistently, and with integrity.

4.4. State Management

In DODIL, *State Management* allows the dynamic enforcement of rules on fields depending on conditions. For example:

- Locking an $immutable$ field if a related status is set to a certain value.
- Making a field required or optional based on other field values.
- Restricting valid choices $enumSubset$ for a field once certain criteria are met.

At runtime, DODIL computes these states into a special object, $'\$state'$, which indicates how each field behaves under current conditions. For example, when the status of some OrderItems is changed, the $quantity$ attribute of said OrderItems can be manipulated when the state of OrderItems is changed to $''['pending','inProcess']''$. The management of these states follows five key concepts:

1. Conditions:
   - Each condition checks a field or a parent's field using an operator and a value.

- If the condition is true, the specified effect is applied to target fields.

2. Effects:
   - $immutable : true$: You cannot change this field value after the condition is met.
   - $required : true$: This field must have a value.
   - $enumSubset$: Restricts the valid enum values for that field.
3. on Field:
   - Indicates which field(s) to apply the effect to (for example, quantity, status, or even * for all fields).
4. References to parent/child fields
   - You can reference a parent schema's field via $\$parent|parent(nth)|parent('schema', nth).fieldName$ or $\$parent|child(nth)|child('schema', nth).fieldName$.
   - Allows child/parent constraints based on the relation's state.
5. Computed $\$state$:
   - At runtime, each field's final state is compiled into a $\$state.fields.\{fieldName\}$ object.
   - This includes any combination of $immutable, required, hidden, enumSubset$, etc.

Multiple creation calls can be chained to add multiple conditions. When conditions overlap, their effects merge, potentially creating a combined or more restrictive state.

4.5. Reactivity

In DODIL, *Reactivity* defines event-driven rules that respond to changes in the application data. It always focuses on the effect location and not the origin of the trigger, similar in quality to a publish/subscribe model. It also automatically updates, propagating and synchronizing changes to related fields and records whenever a change happens to any individual field. DODIL also allows for coordination between multiple related schemas and between related schemas. Reactivity is also designed to be multi-phase, making it highly efficient and avoiding unnecessary database queries until conditions for them are met.

For example, if an Order status changes to *completed*, the application might automatically update all OrderItems to *completed*, assuming that they are children of the Order. Similarly, if an OrderItem's *quantity* attribute changes, the Order item's *total* attribute might be recalculated.

Reactivity is essential to DODIL because it ensures consistency, maintainability, and reduces boilerplate code. It ensures consistency by automatically keeping data in sync when changes occur, removing the burden of manual updates. It ensures maintainability by defining reactive rules in a clear, declarative manner, making application systems overall easier to understand and extend. Boilerplate code is reduced by centralizing logic that would otherwise be spread across multiple services or client-side code.

Reactivity is handled through a number of key concepts. These are:

1. **Reaction Rules**
   Reaction rules are defined using $schema.reaction.create(...)$. Each rule specifies which schema or target the rule belongs to, when to trigger the reactive rule (such as on creation, update, or deletion), which conditions apply to the rule, and what actions are to be done upon its execution.
2. **Event Object** $\$eventObject$
   This represents the record or fields causing the event. For example, if an OrderItem is edited, $\$eventObject$ would contain the newly edited fields and their values.
3. **Modified Fields** $\$modifiedFields$
   This object is the array of fields changed during an update. These are useful for identifying whether a specific field was altered before triggering a reaction.
4. **Multi-Phase Filtering**
   In the first phase, rules are filtered by schema and event type (creation, modification, or deletion).

In the second phase, conditions on $\$eventObject$ and $\$modifiedFields$ are satisfied. In the third phase, any needed database queries are performed if the first two filters are passed.

5. **Reaction Actions**
   An action defines how fields are updated or affected. Common patterns include $action : set$ which is setting a field in the current schema. The function $action : custom$ can be used to add additional functionality.

4.6. Flow

Data *Flow* refers to creating a step-by-step workflow or "flow" that orchestrates the collection/-mutation of all necessary data. *Flow* works hand in hand with Compose, which is useful in collecting complex data from LLMs or structured migrations. For example, while creating an order, the application might need to collect:

1. Basic order information
2. Multiple order items
3. A single delivery address

DODIL's *Flow* feature allows you to group these steps into a cohesive workflow:

- Compositional: Each step references a separate schema.
- Sequential or Parallel: Steps can be strictly ordered or configured for more flexible paths.
- Partial Data Handling: At each step, partial data can be saved or validated before moving on, minimizing errors and rework.

This modular system can significantly reduce complexity in large-scale data collection scenarios. DODIL allows the grouping of schemas in a $flow$, ensuring that the user or LLM must provide certain data in order before proceeding to the subsequent steps. For instance, the process of creating an order might look like the following:

1. Collect order details.
2. Collect multiple order item entries.
3. Collect a single delivery address.

Only once all the steps are complete is the order finalized.

4.7. Prompt Management

As discussed above, LLM prompting is a vastly imprecise system [21]. The concept of *prompt management* tackles this, attempting to create a safe, structured, and reliable way to connect LLM-based agents with real-world data systems. By utilizing DODIL's other concepts and integrating LLMs into existing database schemes, the strengths and advantages of LLMs can safely be utilized. Prompt management focuses on three specific goals, three dangers that often come about from poor integration of LLMs with applications:

1. Reduce Hallucination: LLMs often invent SQL queries or misinterpret schema [22]. With AI-Prompt-Management, queries are validated and generated using real DODIL schema definitions.
2. Enforce Business Rules: Augmentation and state management automatically apply to AI-generated read/write requests, ensuring data consistency [1].
3. Streamline AI-Driven Workflows: Whether the user wants to query, create, or update data, the module generates the necessary SQL or prompts to the user for missing info, without manually crafting complex logic [23].

Firstly, hallucination. Part of DODIL's functionality inolves the parsing of prompts before they are sent to integrated LLMs. Those prompts are then used to consult DODIL's known schemas and output a relevant, valid SQL. This reduces hallucinations because of its grounding in actual schema defintions held by DODIL, making it less likely for the LLM to produce invalid or fictitious fields or queries.

Prompt management also seamlessly applies the same rules defined in the augmentation and state management modules of DODIL. For augmentation, prompt management ensures that any create or read requests from the LLM match requirements of any possible child records in accordance with the schema. As for state management, if a field is immutable when an LLM attempts to edit its field, an error or prompt is triggered for further action. This ensures that the LLM respects all constraints designed for the application's data, without requiring custom-coded logic in the AI layer.

For example, consider the scenario where a user wants to view a particular order along with its items. The LLM is prompted with the user's natural language request. The prompt management module first parses the prompt and finds the relevant schema, checking the important augmentation rules that are relevant. It then generates a valid SQL query. Only then does the module return the data to the LLM, which can present the output to the user in a natural language format.

On the other hand, the user could request the changing of a quality of an object in an immutable state. In that case, the state management module would inform the prompt management to ensure the LLM returns nothing and reports an error to the user, instead of hallucinating an output.

In general, *Prompt Management* acts as a companion to DODIL, ensuring that any LLM-driven queries or updates conform to schema design and relationships, honor augmentation, reduce hallucination, and prompt the user for missing data or disclaimers when appropriate constraints are not met.

5. Conclusions and Future Works

In general, DODIL introduces significant advancements over traditional schema models:

1. Bridging LLMs and Application Data: Enhances LLMs' ability to interact with complex data, moving beyond static queries to dynamic, context-aware interactions.
2. Automation and Dynamic Validation: Supports conditional immutability, field computations, and recursive logic validation for enhanced data consistency and automation.
3. Semantic Augmentation: Enables the seamless integration of related schemas, ensuring relational data is dynamically fetched and processed.
4. LLM Integration: By embedding schema and pipelines, the framework enables LLMs to understand system logic, automate workflows, and interact intelligently with data.

DODIL is a schema framework designed specifically to tackle the challenges modern application developers face in integrating LLMs into their work [11]. By applying the five modules of Relation, State Management, Reactivity, Augmentation, and Prompt Management, the challenges of LLM integration are mitigated and managed, easing the development process for many different kinds of applications.

Author Contributions: Conceptualization, M. R.; Methodolody, M. R.; Writing–original draft preparation, Y. K.; Visualization, M. R. and Y. K.; Validation, M. R. and Y. K.; Writing–review and editing, W. A.; Resources, W. A.; Supervision, W. A. and M. R.; Project Administration, Y. K. and W. A.

Funding: This research received no external funding.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this article:

AI	Artificial Intelligence
API	Application Programming Interface
DB	Database
DODIL	Data-Oriented Design Integration Layer
GPT	Generative Pre-trained Transformer
LLaMA	Large Language Model Meta AI
LLM	Large Language Model
NoSQL	Not Only SQL
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
ORM	Object-Relation Mapping
SAS	Smart Augmented Schemas
SQL	Structured Query Language

References

1. Yao, Y.; Duan, J.; Xu, K.; Cai, Y.; Sun, Z.; Zhang, Y. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing* **2024**, p. 100211.
2. Liang, W.; Zhang, Y.; Wu, Z.; Lepp, H.; Ji, W.; Zhao, X.; Cao, H.; Liu, S.; He, S.; Huang, Z.; et al. Mapping the increasing use of llms in scientific papers. *arXiv preprint arXiv:2404.01268* **2024**.
3. Ganesh, A. Program Scalability Analysis for LLM Endpoints: Ahmdal's Law Analysis of Parallelizability Benefits of LLM Completions Endpoints*. In Proceedings of the 2024 4th International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME), 2024, pp. 1–6. https://doi.org/10.1109/ICECCME62383.2024.10796792.
4. Madden, S.; Cafarella, M.; Franklin, M.; Kraska, T. Databases Unbound: Querying All of the World's Bytes with AI. *Proceedings of the VLDB Endowment* **2024**, *17*, 4546–4554.
5. Chen, Y.; Cui, M.; Wang, D.; Cao, Y.; Yang, P.; Jiang, B.; Lu, Z.; Liu, B. A survey of large language models for cyber threat detection. *Computers & Security* **2024**, p. 104016.
6. Jahić, J.; Sami, A. State of Practice: LLMs in Software Engineering and Software Architecture. In Proceedings of the 2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C). IEEE, 2024, pp. 311–318.
7. Weber, I. Large Language Models as Software Components: A Taxonomy for LLM-Integrated Applications. *arXiv preprint arXiv:2406.10300* **2024**.
8. Raiaan, M.A.K.; Mukta, M.S.H.; Fatema, K.; Fahad, N.M.; Sakib, S.; Mim, M.M.J.; Ahmad, J.; Ali, M.E.; Azam, S. A review on large Language Models: Architectures, applications, taxonomies, open issues and challenges. *IEEE Access* **2024**.
9. Yang, J.; Jin, H.; Tang, R.; Han, X.; Feng, Q.; Jiang, H.; Zhong, S.; Yin, B.; Hu, X. Harnessing the power of llms in practice: A survey on chatgpt and beyond. *ACM Transactions on Knowledge Discovery from Data* **2024**, *18*, 1–32.
10. Patil, R.; Gudivada, V. A review of current trends, techniques, and challenges in large language models (llms). *Applied Sciences* **2024**, *14*, 2074.
11. Jin, H.; Huang, L.; Cai, H.; Yan, J.; Li, B.; Chen, H. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479* **2024**.
12. Yang, J.; Prabhakar, A.; Narasimhan, K.; Yao, S. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems* **2024**, *36*.
13. Patil, S.G. Teaching Large Language Models to Use Tools at Scale. PhD thesis, University of California, Berkeley, 2024.
14. Gupta, A.; Panda, M.; Gupta, A. Advancing API Security: A Comprehensive Evaluation of Authentication Mechanisms and Their Implications for Cybersecurity. *International Journal of Global Innovations and Solutions (IJGIS)* **2024**.
15. Ganesh, S.; Sahlqvist, R. Exploring Patterns in LLM Integration-A study on architectural considerations and design patterns in LLM dependent applications. Master's thesis, Chalmers University of Technology, 2024.
16. Xu, R.; Qi, Z.; Guo, Z.; Wang, C.; Wang, H.; Zhang, Y.; Xu, W. Knowledge conflicts for llms: A survey. *arXiv preprint arXiv:2403.08319* **2024**.

17. Pawade, P.; Kulkarni, M.; Naik, S.; Raut, A.; Wagh, K. Efficiency Comparison of Dataset Generated by LLMs using Machine Learning Algorithms. In Proceedings of the 2024 International Conference on Emerging Smart Computing and Informatics (ESCI), 2024, pp. 1–6. https://doi.org/10.1109/ESCI59607.2024.10497340.
18. Majeed, A.; Hwang, S.O. Reliability Issues of LLMs: ChatGPT a Case Study. *IEEE Reliability Magazine* **2024**.
19. Deng, Y.; Xia, C.S.; Yang, C.; Zhang, S.D.; Yang, S.; Zhang, L. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In Proceedings of the Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–13.
20. Morales, S.; Clarisó, R.; Cabot, J. A DSL for testing LLMs for fairness and bias. In Proceedings of the Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, 2024, pp. 203–213.
21. Patkar, N.; Fedosov, A.; Kropp, M. Challenges and Opportunities for Prompt Management: Empirical Investigation of Text-based GenAI Users. *Mensch und Computer 2024-Workshopband* **2024**, pp. 10–18420.
22. Martino, A.; Iannelli, M.; Truong, C. Knowledge injection to counter large language model (LLM) hallucination. In Proceedings of the European Semantic Web Conference. Springer, 2023, pp. 182–185.
23. Singh, A.; Shetty, A.; Ehtesham, A.; Kumar, S.; Khoei, T.T. A Survey of Large Language Model-Based Generative AI for Text-to-SQL: Benchmarks, Applications, Use Cases, and Challenges. *arXiv preprint arXiv:2412.05208* **2024**.