# GRAMMAR: Grounded and Modular Methodology for Assessment of Closed-Domain Retrieval-Augmented Language Models

Anonymous ACL submission

## Abstract

Retrieval-Augmented Generation (RAG) systems are widely used across various industries for querying closed-domain and in-house knowledge bases. However, evaluating these systems presents significant challenges due to the private nature of closed-domain data and a scarcity of queries with verifiable ground truths. Moreover, there is a lack of analytical methods to diagnose problematic modules and identify types of failure, such as those caused by knowledge deficits or issues with robustness. To address these challenges, we introduce GRAMMAR (**GR**ounded **A**nd **M**odular **M**ethodology for **A**ssessment of **R**AG), an evaluation framework comprising a grounded data generation process and an evaluation protocol that effectively pinpoints defective modules. Our validation experiments reveal that GRAMMAR provides a reliable approach for identifying vulnerable modules and supports hypothesis testing for textual form vulnerabilities. An open-source tool accompanying this framework will be released to easily reproduce our results and enable reliable and modular evaluation in closed-domain settings.

## 1 Introduction

The emergent capabilities of Large Language Models (LLMs) have driven significant research and the widespread deployment of Retrieval-Augmented Generation (RAG) systems for closed-domain settings. This growing trend highlights the critical need for reliable evaluation methods tailored to RAG systems.

A key challenge in evaluating RAG systems is the collection of domain-specific data that includes accurate ground truths. Researchers often rely on data originally designed for human assessment, as seen in studies such as Santurkar et al. (2023); Wang et al. (2022); Zhong et al. (2022, 2023); Hendrycks et al. (2021); Choi et al. (2023). However, these data sources typically reflect open-domain commonsense and world knowledge, whereas industrial in-house RAG systems require data that captures closed-domain knowledge, such as details about company projects and employees. Although user queries can be gathered during system deployment, obtaining accurate ground truths remains a significant challenge. To address this, recent studies have explored reference-free LLM evaluators for cases where ground truths are unavailable (Chern et al., 2023; Min et al., 2023; Es et al., 2023). However, the reliability of these reference-free evaluation methods is still questionable.

Another critical challenge is assessing robustness in RAG systems, where a system may possess adequate knowledge for the semantics of a query but respond inconsistently to different query forms, as highlighted in Shen et al. (2023). Developers and researchers often hypothesize about factors that could impact robustness, such as vulnerable modules (e.g., retrieval mechanisms or language models) or specific input attributes. However, existing research, e.g., studies on adversarial robustness (Alzantot et al., 2018; Li et al., 2020, 2019), lacks effective tools for identifying non-robust modules and analytically testing these hypotheses.

Recognizing these challenges, the paper makes the following contributions:

**Proposing a grounded, controllable data generation process (GRAMMAR-Gen).** To address the issue of ground truth scarcity, we propose GRAMMAR-Gen, a grounded data-generation process designed to ensure reliable evaluation. This process leverages relational databases and LLMs to extract ground truths via SQL queries. Additionally, the template-based query generation enables scalable data creation. By leveraging the capabilities of LLMs, it allows the controlled generation of data with diverse linguistic attributes. Notably, a similar data-generation process for evaluating do-

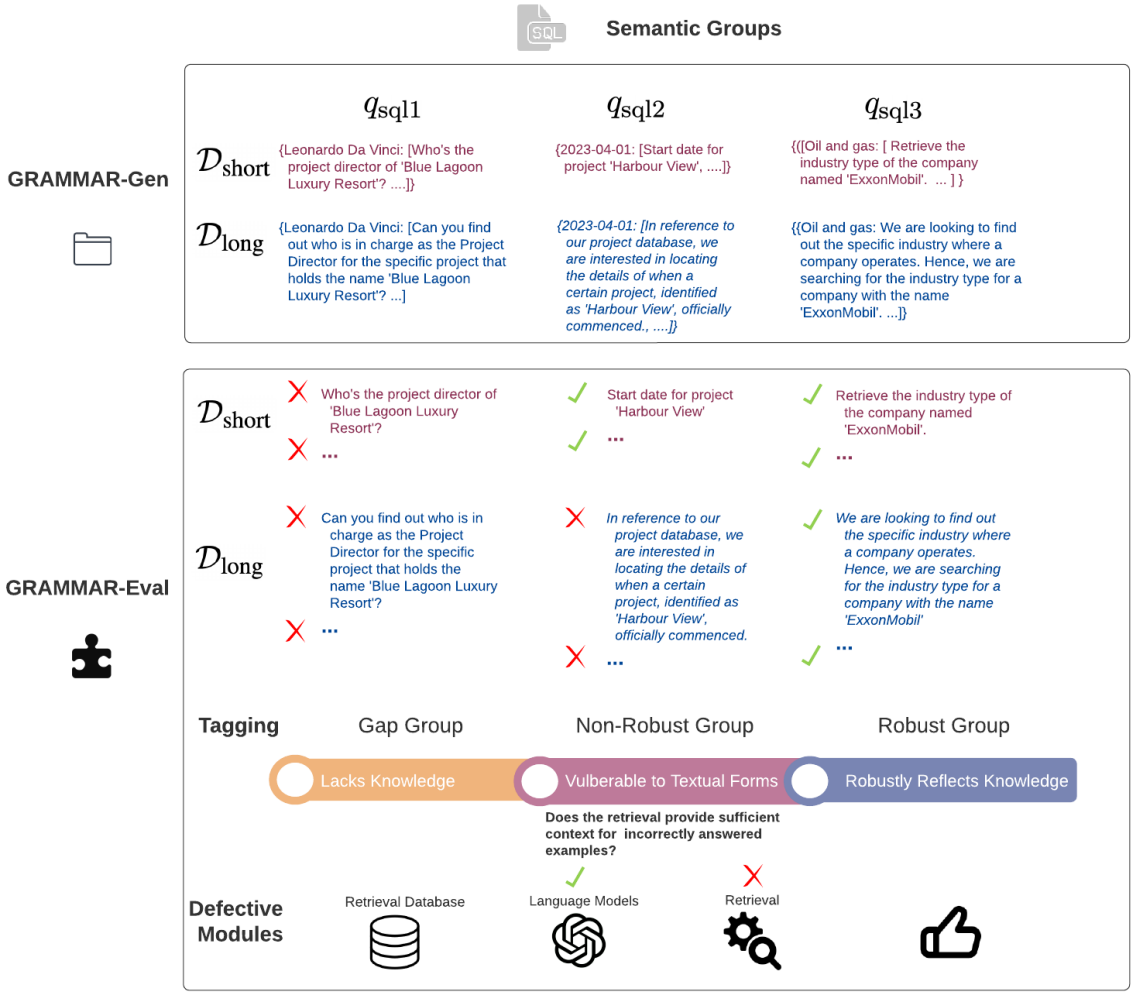



Figure 1: An Example of Applying the GRAMMAR Framework for Modular Evaluation and Hypothesis Testing. The upper section demonstrates the data generation process for creating sets of hypothetically robust ($\mathcal{D}$robust) and non-robust ($\mathcal{D}$non-robust) data. The lower section depicts the evaluation protocol that utilizes the generated data to identify defective modules and facilitate hypothesis testing.

main knowledge was proposed concurrently by Tu et al. (2024)[1].

**Proposing a protocol for modular evaluation and hypothesis testing (GRAMMAR-Eval).** GRAMMAR-Eval combines a grouping and tagging mechanism based on GRAMMAR-Gen with effective strategies to both identify knowledge gaps in the retrieval database and determine whether robustness issues stem from the retrieval mechanism or the language models (LMs). Our empirical results demonstrate the effectiveness of GRAMMAR-Eval in identifying deliberately bugged retrieval systems and accurately testing hypotheses about which input attributes contribute to robustness issues. An example of this process is illustrated in Figure 1.

[1]It was published after the initial preprint/submission of this work.

## 2 Background

Evaluating the performance of Retrieval-Augmented Generation (RAG) systems, especially in closed-domain settings, poses unique challenges, leading to an absence of reliable reference-based evaluation protocols. This section outlines existing approaches and highlights the gaps addressed by our proposed framework, GRAMMAR.

**Retrieval-augmented Generation (RAG)** RAG systems augment LLMs with retrieval for domain-specific use where extensive databases can potentially provide necessary information. A retrieval can be used for retrieving relevant passages or documents. A dense retrieval (Mialon et al., 2023; Lewis et al., 2020; Borgeaud et al., 2022) consists of an embedding model to produce a vector $v_q$ for any given query $q$. This vector $v_q$ is then used to

identify pertinent text segments normally via dot product calculations with the vectors of text chunks. The most relevant chunks are aggregated to form an additional context $c$, which serves as the factual basis for the response generation, until adding another chunk would surpass the maximum allowable context length. Subsequently, an LLM will process this aggregated context alongside the original query $q$ to generate a predicted answer $\hat{a}$ for it.

In contrast, sparse retrieval computes similarity without parametric embedding models. For instance, keyword-matching approaches (Manning, 2009) select documents that share the highest number of common words for context retrieval. Additionally, methods like TF-IDF (Chen et al., 2017) factor in the inverse document frequency, emphasizing the significance of less common words which are likely to be more indicative of pertinent content.

**Reference-Free Evaluation** Reference-free evaluation methods (Es et al., 2023) assess model performance without requiring predefined ground truths. These methods can be categorized according to their applicable systems: 1) Protocols for LLMs: FactScore (Min et al., 2023) and FacTool (Chern et al., 2023) use retrieval combined with LLMs as validators, but these are not directly applicable to RAG systems as retrieval is reserved for evaluation. 2) Protocols for RAG: ARES and RAGAS focus on evaluating the faithfulness of generated answers against retrieved contexts, with LLMs scoring the relevance and correctness of the context (Saad-Falcon et al., 2023; Es et al., 2023). 3) Protocols for Both: SelfCheck (Manakul et al., 2023) utilizes the stochastic nature of LLMs for self-validation. While these methods offer some insights, their reliability is often questioned, especially for closed-domain applications where specific, accurate answers are crucial.

**Reference-Based Evaluation** While reference-free evaluation can be applied for evaluating both open-domain and closed-domain queries, reference-based evaluation typically assesses open-domain knowledge. The reason is that open-domain queries are widely available with ground-truth answers from public resources, exams, and surveys (Santurkar et al., 2023; Wang et al., 2022; Zhong et al., 2022, 2023; Hendrycks et al., 2021; Choi et al., 2023). These evaluations compare generated answers to known correct answers, calculating metrics that reflect the model's accuracy. However, closed-domain scenarios lack reference-based evaluation due to the unavailability of ground-truth answers. GRAMMAR-Gen is proposed to solve this issue.

## 3 Is Reference-free Evaluation Reliable?

This section analyzes the reliability of reference-free evaluation methods using 198 examples from an engineering company. The RAG system for evaluation is implemented with dense retrieval and GPT-3.5 [2].

This preliminary study briefly highlights the potential benefits of reference-based evaluation, as our primary contribution focuses on proposing a reference-based evaluation framework. A detailed analysis of reference-free evaluation on state-of-the-art models is left for future work.

**Two Evaluation Perspectives: Optimism and Cynicism** To structure this analysis, reference-free evaluation is treated as a binary classification task. The ground truth reflects the actual correctness of RAG responses, while the evaluation models' judgments serve as the task's predictions. Thus, low precision indicates an optimistic bias (where incorrect RAG responses are often judged as correct), and low recall indicates a cynical bias (where correct RAG responses are often judged as incorrect).

**Two Evaluation Protocols** We assess two reference-free protocols.

- RAGAS-Fact (Es et al., 2023): This protocol utilizes the context-query-response triplets to assess the veracity of responses. It evaluates the faithfulness of a response by calculating the ratio of claims grounded on the context to the total claims made. This process involves identifying statements that hold atomic facts, following the methodology outlined by Chern et al. (2023).

- SelfCheck (Manakul et al., 2023): This protocol relies on the stochastic generation of responses, based on the premise that incorrect answers are unlikely to be consistently produced. This principle, initially applied to LLMs, is adapted in our analysis of RAG systems. In a departure from its original application, we enhance the evaluation prompt to include not only stochastic responses but

[2] An OpenAI embedding model for document embedding and *gpt-3.5-turbo-16k*

also the query itself. Refer to Appendix A for details. To generate four stochastic samples, we adjust the temperature setting to 1.0, contrasting with a temperature of 0.0 used to generate the primary response. A response is deemed correct if it aligns consistently across all stochastic samples.

**Results: Both are Extremely Optimistic on Wrong Predictions While SelfCheck Becomes Too Cynical on Correct Predictions** As shown in Table 1, both RAGAS and SelfCheck achieve low precision (19% and 15%, respectively), highlighting a deficiency in correctly identifying erroneous predictions. For RAGAS, this may be because the RAG system produces responses that, while contextually relevant, fail to directly address the intended query. However, RAGAS demonstrates high recall, indicating accurate assessment of correct RAG responses. Overall, the preliminary insight suggests that reference-free evaluation methods are not reliable for evaluation, underscoring the critical need for a robust, reference-based evaluation framework to ensure reliable model assessment.

| | Precision | Recall |
|---|---|---|
| RAGAS-Fact | 19% (11-27%) | 92% (86-97%) |
| SelfCheck | 15% (0.08-0.22%) | 50% (0.4-0.59%) |

Table 1: Reliability of reference-free evaluation protocols. Taking the size of the sample into account, the Z-test with 95% confidence intervals is utilized.

# 4 GRAMMAR

This section introduces GRAMMAR-Gen and GRAMMAR-Eval for grounded and modular evaluation: 1) GRAMMAR-Gen begins by generating templates based on a database schema, enabling scalable data creation [3]. By leveraging LLMs, GRAMMAR-Gen offers controllability in generating text variations, which not only supports scalable data generation but also facilitates modular robustness evaluation and hypothesis testing. The process of SQL and text template generation is detailed in § 4.1. SQL queries, serving as intermediate representations, are used to create ground-truth answers by querying relational databases, as explained in § 4.2. The overall process is illustrated in Algorithm 1 and Figure 2. 2) GRAMMAR-Eval, an evaluation protocol that effectively pinpoints defective modules by assessing the knowledge deficit of the retrieval database and detecting robustness issues originating from either the retrieval mechanisms or the language models. Further details can be found in § 4.3.

[3]Further details can be found in Appendices D.2 and E

**Algorithm 1** GRAMMAR-Gen

**Input**: SQL Template Generator $g_{\text{sql}}$, Text Template Generator $g_{\text{t}}$, Semantic criteria for SQL template generation $\mathcal{C}_{\text{sql}}$, Linguistic criteria for text template generation $\mathcal{C}_{\text{t}}$, Database $\mathcal{D}$, Database Schema $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$, where each $S_i$ is a schema for a table.
**Output**: Final evaluation data $Q$

$Q \leftarrow \emptyset$
$\mathcal{S}_{\text{target}} \in \mathcal{S}$
$\{tpl_{\text{sql}}\} \leftarrow g_{\text{sql}}(\mathcal{S}_{\text{target}}, \mathcal{C}_{\text{sql}})$
**for** $tpl_{\text{sql}} \in \{tpl_{\text{sql}}\}$ **do**
  $\{tpl_{\text{t}}\} \leftarrow g_{\text{t}}(tpl_{\text{sql}}, \mathcal{C}_{\text{t}})$
  $P \leftarrow$ extract placeholders from $tpl_{\text{sql}}$
  **for** each $p \in P$ **do**
    $C \leftarrow$ extract column from $p$
    $V_p \leftarrow$ query $\mathcal{D}$ for distinct values of $C$
  **end for**
  $Comb \leftarrow$ Cartesian product of $\{V_p : p \in P\}$
  **for** each $comb \in Comb$ **do**
    $q_{\text{sql}} \leftarrow$ substitute $comb$ into $tpl_{\text{sql}}$
    $a \leftarrow$ query $\mathcal{D}(q_{\text{sql}})$
    $\{q_t\} \leftarrow$ substitute $comb$ into $tpl_{\text{t}} \in \{tpl_{\text{t}}\}$
    $Q \leftarrow Q \cup \{(\{q_t\}, a)\}$
  **end for**
**end for**
**return** $Q$

## 4.1 Generating Query Templates

The GRAMMAR-Gen process first utilizes database schemas and LLMs to generate SQL query templates and their corresponding textual forms.

**Database Schema** Database schema serves as the blueprint that defines how classes of entities are organized and the relations among them. Appendix B demonstrates the reasons why we use database schema for knowledge representation. Let a database schema with a collection of $N$ tables $S = \{S_1, S_2, \ldots, S_N\}$, where each $S_i$ is a schema for a database table. A table schema $S_i$ consists of the table name $T_i$, a set of attributes $\{A_1, A_2, \ldots, A_M\}$ and a set of constraints, i.e., $S_i = (T_i, \{A_1, A_2, \ldots, A_m\}, C)$, where the constraint $C$ demonstrates the primary key $PK$ and foreign keys $FK$. Specifically, $FK\,(T_i.A_k \rightarrow T_j.A_h)$ indicates a foreign key $A_k$ in table $T_i$ referencing the primary key $A_h$ in table $T_j$. The schema defines the knowledge structure of an entity, which can be enough to generate query semantics that is not tied to any specific linguistic

expressions and sensitive data.

**Generating SQL Templates** A SQL template $tpl_{\text{sql}}$ essentially represents the semantics of a query. Various query semantics are formulated through the application of relational algebra operators. In our study, we focus on three fundamental operators: Select ($\sigma$), Project ($\pi$), and Join ($\bowtie$). An automated SQL template generator $g_{sql}$ is powered by a generative LLM. When given a target schema $\mathcal{S}$target and specified SQL criteria $\mathcal{C}$sql, $g_{\text{sql}}$ generates a set of SQL query templates.

**Generating Text Templates** $tpl_{\text{sql}}$ can be instantiated into various text templates, each denoted as $tpl_t$. This process, driven by the text template generator $g_{\text{t}}$, leverages linguistic criteria $\mathcal{C}_{\text{t}}$ to transform $tpl_{\text{sql}}$ into natural language forms that align with specified linguistic characteristics, e.g., complexity, length and stylistic nuances [4]. The text template generator $g_{\text{t}}$ is also operationalized using a sophisticated language model, such as GPT-4, which is adept at producing diverse linguistic variations of the same semantic content.

$$\{tpl_{\text{t}}\} = g_{\text{t}}(tpl_{\text{sql}}, \mathcal{C}_{\text{t}}),$$

where $\{tpl_{\text{t}}\}$ signifies the resulting list of text templates.

### 4.2 From Templates To Evaluation Data

Utilizing a database $\mathcal{D}$ that aligns with the predefined schema, the framework samples a diverse range of queries along with their corresponding ground-truth answers. This dataset, derived from SQL and text templates filled with database content, provides a rich source for evaluation.

**Generating Queries Via Placeholder Fill-in** The placeholder fill-in step populates the templates with actual data from the rows of tables in $\mathcal{D}$. Specifically, the approach employs a SELECT query format: "SELECT DISTINCT {column_name} FROM {table_name};" to ensure variety in the data points. It handles SQL query templates with multiple placeholders by employing combinations of placeholders and using Cartesian products to generate multiple query permutations. This approach leads to a comprehensive set of SQL queries ($q_{\text{sql}}$) and their natural language equivalents, the text queries ($q_t$).

[4]Examples of $\mathcal{C}_{\text{sql}}$ and $\mathcal{C}_{\text{t}}$ are specified in Appendix C.

**Generating Answers** Answer generation is integral to completing our dataset. Each SQL query ($q_{\text{sql}}$) is executed against $\mathcal{D}$ to match with a factual answer $a$. Each answer is then paired with the relevant text queries $\{q_t\}$, forming the basis of our evaluation data.

### 4.3 Modular Evaluation

Evaluating the overall performance of an RAG model can obscure the specific weaknesses or robustness of its constituent modules. This section introduces a modular evaluation protocol to address this.

**Query Grouping and Tagging** The process involves grouping and tagging queries based on their semantics. Specifically, with GRAMMAR-Gen, textual queries $Q$ generated from a particular SQL query $q_{\text{sql}}$ are organized into a semantic group. Depending on the model $M$'s performance, $q_{\text{sql}}$ (and consequently, the group $Q$) are tagged into three principal categories, as shown in Figure 1.

- **Gap Groups** emerge when the model $M$ consistently provides incorrect answers for every text query within a subset $S \subseteq Q$ produced through the data generation process. This reflects a deficiency in $M$ 's knowledge or capability. Formally, this scenario is described as:

  $$\forall q_t \in S, \neg M_e(M\,(q_t)),$$

  where $M_e$ is the evaluation model, which returns "TRUE" if the model's response $M\,(q_t)$ is correct.

- **Robust Groups** are established when model $M$ accurately responds to all text queries within the subset $S$, evidencing a robust comprehension of the SQL logic. Formally, this is articulated as:

  $$\forall q_t \in S, M_e(M(q_t)).$$

- **Non-robust Groups** are characterized by model $M$'s ability to potentially answer the query semantics correctl, but with at least one query $q_t$ within $S$ that $M$ fails to predict correctly. This is formally denoted as:

  $$\exists q_t \in S, M_e(M(q_t)) \wedge \exists q_t \in S, \neg M_e(M(q_t)).$$

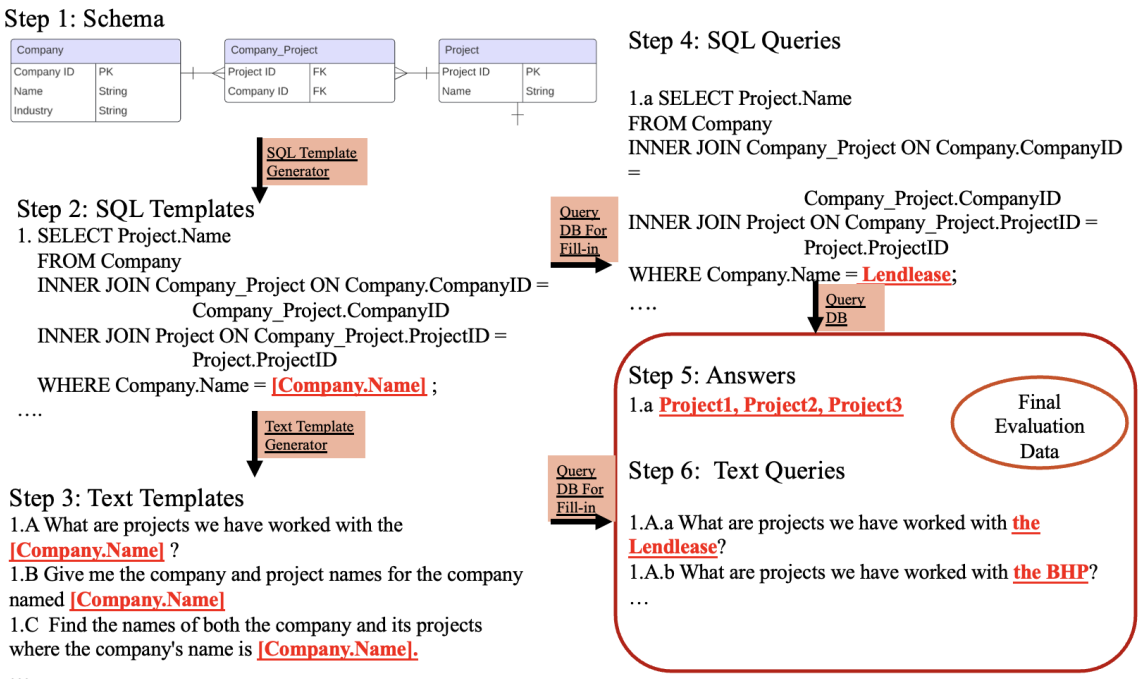



Figure 2: Scaled Generation of Query-answer Pairs. Step 3 and step 4 are independent of each other and depend only on the SQL templates.

**Assessing Retrieval Database via Gap Groups** Gap examples, which are incorrectly predicted instances within gap groups, highlight errors in the retrieval database that arise due to its limited coverage. The adequacy of the knowledge within the database can be quantified as follows:

$$\text{Acc}_{\text{retrieval\_db}} = 1 - \frac{\text{Number of Gap Groups}}{\text{Total Number of Semantic Groups}} \quad (1)$$

A low $\text{Acc}_{\text{retrieval\_db}}$ suggests the need for expanding the database.

**Isolating Retrieval Database Errors** To ensure accurate evaluation of the retrieval and language models without interference from retrieval database errors (i.e., distinguishing knowledge gaps from robustness performance), two strategies have been developed:

1. Removing Gap Examples: To assess the other two modules, we calculate refined accuracy, denoted as $R$, by excluding gap examples:

$$R = \frac{\text{\# with Correct Predictions}}{\text{Total \#} - \text{Total \# in Gap Groups}}, \quad (2)$$

where "#" represents "the number of instances".

2. Balancing Gap Groups: To understand the advantage of this refined accuracy over baseline accuracy Acc, consider the following relationship:

$$\begin{aligned} \text{Acc} &= R \times \frac{\text{Total \#} - \text{Total \# in Gap Groups}}{\text{Total \#}} \\ &= R \times (1 - \frac{\text{Total \# in Gap Groups}}{\text{Total \#}}) \end{aligned} \quad (3)$$

This equation shows that a smaller normalized term, $\lambda = \frac{\text{Total \# in Gap Groups}}{\text{Total \#}}$, enhances evaluation accuracy. Intuitively, a smaller $\lambda$ implies fewer gap examples (errors propagated from the initial module), leading to more accurate results. Maintaining a consistent $\lambda$ across evaluation models or datasets ensures accurate assessments, such as for the hypothetically robust dataset $\mathcal{D}_{\text{robust}}$ and the non-robust dataset $\mathcal{D}_{\text{non-robust}}$ discussed in §5. With GRAMMAR-Gen, an equal number of text queries can be generated within each semantic group by adjusting the text template generator.

**Identifying Non-Robust Retrieval or Non-Robust LM: A "Context Comparison" Approach** To determine whether robustness issues stem from the retrieval module or the language model, we focus on incorrectly predicted queries within the non-robust group, each referred to as

| | SQL Templates | SQL Queries | Text Templates | Text Queries Short | Text Queries Long | Text Queries (Balanced) Short | Text Queries (Balanced) Long |
|---|---|---|---|---|---|---|---|
| Aurp | 11 | 157 | 30 | 314 | 257 | 471 | 471 |
| Spider-Open | 5 | 57 | 10 | 397 | 436 | 570 | 570 |
| Spider-Closed | 5 | 57 | 10 | 426 | 430 | 570 | 570 |

Table 2: Statistics of Evaluation Datasets

| Isolating Errors from Retrieval Database | LLM | Aurp Robust | Aurp Non-Robust | Spider-Closed Robust | Spider-Closed Non-Robust | Spider-Open Robust | Spider-Open Non-Robust |
|---|---|---|---|---|---|---|---|
| Baseline: No Action | No Action | 0.27 | **0.51** | 0.26 | **0.28** | 0.67 | **0.87** |
| | Context Comparison | 0.29 | **0.51** | 0.31 | **0.34** | **0.37** | 0.35 |
| Remove Gap Groups | No Action | 0.95 | **0.96** | **0.58** | 0.44 | **0.98** | 0.96 |
| | Context Comparison | **1** | 0.98 | **0.62** | 0.49 | **0.51** | 0.39 |
| Balance Gap Examples | No Action | **0.27** | 0.27 | **0.28** | 0.21 | **0.69** | 0.68 |
| | Context Comparison | **0.29** | 0.28 | **0.32** | 0.28 | **0.36** | 0.29 |

Table 3: Validating GRAMMAR's tricks in in detecting non-robust retrieval module under a deliberately constructed RAG pipeline. Accuracy is calculated on selected examples across various types of semantic groups.

$q_{\text{target}}$. Specifically, we pose the following question:

*Does the retrieval module provide sufficient context for $q_{target}$?*

By leveraging other semantically identical but correctly answered queries within the group, we can assess whether the context provided is sufficient. Specifically, we compare the document indices idxtarget associated with $q$target to the indices of correctly answered queries, denoted as $I = \text{idx1}, \text{idx2}, \ldots$. If idxtarget matches any index in $I$, the context provided by idxtarget is considered sufficient.

## 5 A Toy Case for Validation: Detecting A Vulnerable Retrieval

This section demonstrates the effectiveness of GRAMMAR on question answering tasks.

### 5.1 Experiment Setup

**Evaluation Target** To validate our framework, we deliberately introduce a vulnerability in the retrieval component by using a basic keyword-matching retrieval approach. We use GPT3.5 for language generation. While this method is less common in practical applications, it is inherently weak in handling lengthy queries, making it an ideal candidate for testing our evaluation framework.

**Evaluation Benchmarks and Datasets** Closed-domain RAG systems lack established benchmarks, especially because questions must be answered using documents containing knowledge that is not publicly available or known to public LLMs. To address this gap, we modify the Spider benchmark (Yu et al., 2018) and create two synthetic benchmarks with fictitious data. **1) Context-Augmented Spider (Open-Domain)**: We use the "company_employee" relational database from Spider to simulate a general real-world scenario. Documents for retrieval are synthesized using GPT-4 based on the database. **2) Context-Augmented Spider (Closed-Domain)**: This is similar to the first dataset, but with company names modified to fictitious entities that are either unused by specific companies or generic according to the latest GPT-4 models. **3) Aurp**: To create a strictly closed-domain evaluation, we synthesize fictitious facts about a fictional company called Aurp. A relational database and corresponding retrieval documents are generated so that questions derived from the database can only be answered by the provided documents. Details on the data synthesis process are provided in Appendix G, and the database schemas are outlined in Appendix F. Using these three settings, datasets are generated by GRAMMAR-Gen to validate the design of GRAMMAR-Eval. The statistics of the generated datasets are summarized in Table 2.

### 5.2 Results

This section validates the effectiveness of modular evaluation within the proposed framework in

identifying defective modules.

**GRAMMAR-Eval's Effectiveness on Modular Evaluation** The results in Table 3 highlight the value of modular evaluation in identifying the defective retrieval. Regardless of the methods used to isolate errors from the retrieval database and LLMs, the accuracy consistently identifies the vulnerable retrieval module. The next three paragraphs will provide a more detailed analysis.

**Preventing Forward Error Propagation** Preventing the propagation of knowledge deficits from retrieval databases is crucial. The two proposed strategies effectively address this issue. In all benchmarks, after either removing gap groups or balancing gap examples, accuracy on the robust text form consistently outperforms the non-robust text form (with the exception of Row 3 in Table 3, where the results are very close).

**Impact of Backward Errors from LLMs on Retrieval Assessment** In some cases, such as Row 3 in Table 3, language model inaccuracies reduce the accuracy, masking the true performance of retrieval. While these instances are not prevalent in our dataset, they still distort the results. This distortion may be due to the stochastic nature of the LM's performance, or, in a less favorable scenario, because the non-robust text forms used in retrieval inadvertently benefit the LM. Based on the results from the other two datasets, it is more likely the former.

**Applicability of GRAMMAR to Open-Domain RAG** Table 3 also demonstrates the effectiveness of GRAMMAR for performing modular evaluation on the open-domain Spider dataset. However, in open-domain settings, the GPT model may generate accurate answers from its internal knowledge. This can lead to inaccuracies in assessing both the retrieval system's knowledge gaps and the robustness of the retrieval mechanism. Specifically, the potential limitations in applying GRAMMAR-Eval to open-domain settings include: **1) Misjudgment of the Retrieval Database**: Some gap examples in the retrieval database may go undetected, leading to an inflated Accretrieval_db. This issue is empirically demonstrated in Table 4, which compares Accretrieval_db before and after filtering out open-domain queries for Spider-Open. Closed-domain queries are identified by applying GRAMMAR to detect gap groups using answers from the LLM-only system. **2) Limitations of Context Comparison**: Context comparison may fail when correctly answered examples do not actually receive sufficient context. This issue is evident in Table 3, where the accuracy decreased under "Context Comparison" compared to "No Action." The decrease in accuracy is likely due to the introduction of incorrect context, which adds noise to the language model's generation, thereby hindering its performance. In contrast, accuracy consistently improves for the two closed-domain benchmarks when context comparison is applied.

| | $\text{Acc}_{\text{retrieval\_db}}$ |
|---|---|
| With Open-Domain Queries | 0.53 |
| Without Open-Domain Queries | 0 |

Table 4: Wrong judgement on retrieval database with open-domain queries

## 6 Conclusions

In this work, we have introduced the GRAMMAR framework specifically designed for Retrieval-Augmented Generation (RAG) systems in closed-domain settings. The data generation module (GRAMMAR-Gen) improves the reliability of evaluation, while the evaluation protocol (GRAMMAR-Eval) enables detailed modular analysis.

**Reproducibility** Our open-source Python package enables easy reproduction of all results, including the regeneration of experimental data and outcomes from our validation study. Detailed instructions on using the package are provided in Appendix H. Due to data privacy policies, the 198 examples and the RAG system from the sponsoring company are not included.

## 7 Limitations

The data generation process has certain limitations, such as its inability to handle scenarios with multiple correct answers (refer to Appendix D.1). Additionally, the expressiveness of the generated queries is constrained by the database schema and SQL (refer to Appendix D.3), limiting the ability to generate queries that require multi-step reasoning or free-form responses. However, as demonstrated in the case study, simple queries are sufficient for identifying and addressing intrinsic robustness issues within the models.

Besides, while the simple context comparison method performs well for closed-domain evalua-

tion, it may need refinement for more complex retrieval tasks. This method was primarily used to validate the importance of modular evaluation and to establish a foundation, along with a reusable code base, for future development.

## References


Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani Srivastava, and Kai-Wei Chang. 2018. Generating natural language adversarial examples. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2890–2896, Brussels, Belgium. Association for Computational Linguistics.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. 2022. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pages 2206–2240. PMLR.

Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. 2017. Reading Wikipedia to answer open-domain questions. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1870–1879, Vancouver, Canada. Association for Computational Linguistics.

I Chern, Steffi Chern, Shiqi Chen, Weizhe Yuan, Kehua Feng, Chunting Zhou, Junxian He, Graham Neubig, Pengfei Liu, et al. 2023. Factool: Factuality detection in generative ai–a tool augmented framework for multi-task and multi-domain scenarios. *arXiv preprint arXiv:2307.13528*.

Jonathan H Choi, Kristin E Hickman, Amy Monahan, and Daniel Schwarcz. 2023. Chatgpt goes to law school. *Available at SSRN*.

Shahul Es, Jithin James, Luis Espinosa-Anke, and Steven Schockaert. 2023. Ragas: Automated evaluation of retrieval augmented generation.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding. In *International Conference on Learning Representations*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc.

Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. 2019. Textbugger: Generating adversarial text against real-world applications. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.

Linyang Li, Ruotian Ma, Qipeng Guo, Xiangyang Xue, and Xipeng Qiu. 2020. BERT-ATTACK: Adversarial attack against BERT using BERT. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6193–6202, Online. Association for Computational Linguistics.

Potsawee Manakul, Adian Liusie, and Mark J. F. Gales. 2023. Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models.

Christopher D Manning. 2009. *An introduction to information retrieval*. Cambridge university press.

Grégoire Mialon, Roberto Dessi, Maria Lomeli, Christoforos Nalmpantis, Ramakanth Pasunuru, Roberta Raileanu, Baptiste Roziere, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. 2023. Augmented language models: a survey. *Transactions on Machine Learning Research*. Survey Certification.

Sewon Min, Kalpesh Krishna, Xinxi Lyu, Mike Lewis, Wen-tau Yih, Pang Wei Koh, Mohit Iyyer, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2023. Factscore: Fine-grained atomic evaluation of factual precision in long form text generation. *arXiv preprint arXiv:2305.14251*.

Jon Saad-Falcon, Omar Khattab, Christopher Potts, and Matei Zaharia. 2023. Ares: An automated evaluation framework for retrieval-augmented generation systems.

Shibani Santurkar, Esin Durmus, Faisal Ladhak, Cinoo Lee, Percy Liang, and Tatsunori Hashimoto. 2023. Whose opinions do language models reflect? *arXiv preprint arXiv:2303.17548*.

Xinyue Shen, Zeyuan Chen, Michael Backes, and Yang Zhang. 2023. In chatgpt we trust? measuring and characterizing the reliability of chatgpt. *arXiv preprint arXiv:2304.08979*.

Shangqing Tu, Yuanchun Wang, Jifan Yu, Yuyang Xie, Yaran Shi, Xiaozhi Wang, Jing Zhang, Lei Hou, and Juanzi Li. 2024. R-eval: A unified toolkit for evaluating domain knowledge of retrieval augmented large language models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD24-ADS)*.

Siyuan Wang, Zhongkun Liu, Wanjun Zhong, Ming Zhou, Zhongyu Wei, Zhumin Chen, and Nan Duan. 2022. From lsat: The progress and challenges of complex reasoning. *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, 30:2201–2216.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Wanjun Zhong, Ruixiang Cui, Yiduo Guo, Yaobo Liang, Shuai Lu, Yanlin Wang, Amin Saied, Weizhu Chen, and Nan Duan. 2023. Agieval: A human-centric benchmark for evaluating foundation models. *arXiv preprint arXiv:2304.06364*.

Wanjun Zhong, Siyuan Wang, Duyu Tang, Zenan Xu, Daya Guo, Yining Chen, Jiahai Wang, Jian Yin, Ming Zhou, and Nan Duan. 2022. Analytical reasoning of text. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 2306–2319, Seattle, United States. Association for Computational Linguistics.

# A Prompt Templates for LLM-Based Evaluation

For SelfCheck, given that our responses are typically concise, we opt for assessing the overall response rather than rather than performing a sentence-by-sentence validation as in the original implementation, as shown in Table 5.

# B Knowledge Structure and Representation

Commonly, the knowledge is formalized by entities along with their relations, e.g., a knowledge graph, ontology and a database schema. Instead of using knowledge graphs containing concrete entities and their relations, we use the schema-based definition, where the schema is characterized by entity types (classes of entities). An entity type is defined by its name, its attributes and relations with other entity types that will be concretized as components in a database, i.e., the table name, columns and foreign keys.

The definition of entity types can provide combinatorial expansion of data generation, protecting the leak of private information (normally stored as rows of tables) during the data generation process and the ability of retrieving ground-truths via Structured Query Language (SQL). The three advantages are the reasons why we represent knowledge in database schema rather than knowledge graphs. It disentangles the private data stored in the database with universal metadata expressed in the database schema. Since schemas and templates contain only meta-data for structuring the real data, there is no private issue to use commercial large language models for this process.

# C Generator Criteria

Each criterion for the SQL generator and text generator ($\mathcal{C}_{\text{sql}}$ and $\mathcal{C}_{\text{t}}$, respectively) consists of brief instructions provided to the LLM, specifying the type of SQL or text that needs to be generated.

Table 6 provides examples:

# D Details of SQL Template Generation

## D.1 Constraints/Guidelines for Creating SQL Templates

The constraints limit the types of templates that can be generated. This section demonstrates two important constraints, which are verbalized as the prompt for LLMs.

**Selecting Attributes That Are Understandable to Humans** The SQL template generator is required to follow the constraint: “The selected and condition columns in the query MUST BE MEANINGFUL and DESCRIPTIVE to ensure the queries are easily understood by non-technical users.”.

**Avoiding Answer Multiplicity** In the evaluation of question-answering (QA) models, a unique challenge arises from the existence of multiple valid answers to a single query, which necessitates a nuanced approach to assessing model performance. Consider the question: “Get the name of the client associated with the project named Innovation Precinct” For such a question, a set of correct responses could include any combination of names from a predefined list, such as {Apple, Amazon, Meta, Facebook}. This multiplicity of correct answers underscores the complexity of evaluating QA models, as it requires the assessment mechanism to recognize and validate the full spectrum of possible correct answers rather than comparing the model’s output against a single ’gold standard’ answer. This scenario demands a more flexible method for answer validation that can accommodate the variability in correct responses. Also, evaluation metrics may be required to effectively measure the performance of QA models in handling diverse and equally valid answers. Such metrics must account for the exhaustive set of correct answers and evaluate the model’s ability to retrieve any or all valid responses within the context of the query, thereby ensuring a comprehensive assessment of the model’s

***SelfCheck*** from (Manakul et al., 2023)
Context: {context}
Sentence: {sentence}
Is the sentence supported by the context above?
Answer Yes or No: ***SelfCheck-QA***
Query: {query}
Answer A: {answer}
Answer B: {stochastic_answer}
Do both answers address the query with equivalent meaning?
Use only "Yes" or "No" for your evaluation:

***Ragas*** from (Es et al., 2023)
Natural language inference. Use only 'Yes' (1), 'No' (0) and 'Null' (-1) as verdict.
context: {context}
statement: {statement}
verdict:

***Ours***
Evaluate the accuracy of the given response in relation to the true answer for the specified query. After evaluating, provide a judgement as either "Correct" or "Incorrect" based on whether the ##Given Response## accurately matches the ##True Answer##.
##Query##: {query}
##True Answer##: {true_answer}
##Given Response##: {given_response}
##Judgement##:

Table 5: Prompt templates for LLM-based Evaluation.

$\mathcal{C}_{\text{sql}}$: *Generating SQL queries with one placeholder*

```
Each query must contain at least one parameter placeholder in the WHERE clause.
```

$\mathcal{C}_{\text{t}}$: *Generating short queries*

```
Short and Clear: Keep your queries short and straightforward.
Cut down on words and skip parts of speech, such as conjunctions and articles.
It's okay to use fragmented phrases as long as they still convey the full meaning.
Valid examples: "client of '[Project.Name]'" or "client for '[Project.Name]'";
Invalid Examples: "Find the client of a project named '[Project.Name]'.
```

$\mathcal{C}_{\text{t}}$: *Generating long queries*

```
Complex Sentence Structure: Ensure your queries are always in complete sentences.
Opt for longer, more complex sentence structures,
incorporating elements of speech like conjunctions and articles for fuller expression.
Each query should be at least 30 words long.
You can add context and background information to the query.
```

Table 6: Examples of criteria for the SQL generator and text generator.

understanding and response generation capabilities. Generally, encompassing all valid responses within the database is not feasible. Hence, rather than let the non-robust results affect the accuracy of evaluation, we look for an approach to avoid it, i.e., only generating queries ensuring one ground-truth answer. Specifically, the evaluation of certain SQL queries, such as "SELECT Name, Department, BusinessAddress FROM Employee WHERE JobTitle = 'designer'", requires a complete and thorough listing of multiple answers that can be dynamically changed. To solve the issue, the solution involves adding a specific criterion to the prompt for generating SQL queries. This criterion, "Ensure the query yields a specific and singular answer", aims to produce queries that result in a single, clear answer, thereby avoiding the complexities of multiple possible correct answers. For example, with this criterion, the query "SELECT Industry FROM Company WHERE Name = '[Company.Name]';"

```
You are a SQL query Template Generator: Generate ACCEPTABLE SQL query
templates with placeholders according to the give data schema and
requirements. A simple example of an acceptable SQL query template is:
SELECT Industry FROM Company WHERE Name = '[Company.Name]';

You must follow the basic criteria below except for other requirements:
##CRITERIA##
- The placeholder format should be a combination of a table name and
a column name, enclosed within square brackets, e.g., '[User.Name]'.
- Use only 'SELECT' queries.
- Select specific column(s) instead of using '*'. Avoid projecting
attributes that appear in the predicate.
- The selected and condition columns in the query MUST BE MEANINGFUL and
DESCRIPTIVE to ensure the queries are easily understood by non-technical
users.
- Avoid using technical column names that don't clearly signify the nature
of the entities or objects involved, e.g., column for semantically void
record identifiers.
- Do not create redundant or semantically duplicated queries when translated
into natural language.
- Each query must contain at least one parameter placeholder in the WHERE
clause.
- Ensure the query yields a specific and singular answer to avoid multiplicity
issues, thus facilitating accurate chatbot evaluation.
 {SPECIFIC_REQUIREMENTS}
- If no acceptable SQL template can be generated with the given table and
column information, do not generate any text.

##RESPONSE FORMAT##
- Output each SQL template as a single line, without any prefix or suffix.
- Do not include any other text in your response, even something like
##RESPONSE_END##.

##DATA SCHEMA##
{GIVEN_SCHEMA}

##RESPONSE_START##
```

Table 7: A prompt template designed to guide the LLM in functioning as a controllable SQL template generator

is preferred as it's likely to yield a singular answer about a company's industry based on a specific company name. In contrast, without this criterion, queries like "SELECT Name FROM Company WHERE Industry = '[Company.Industry]';" are acceptable but may result in multiple names, leading to evaluation difficulties due to data completeness and query multiplicity issues.

### D.2 Possibilities Of SQL Templates

**One Table/Entity** Given a schema with only one table, the possibilities for SQL templates can be analyzed by considering the basic SQL operators like SELECT, WHERE. Here's a breakdown:

- Variation in Selected Attributes (SELECT): The number of SQL templates varies based on the combination of columns selected. If the "Company" table has $n$ columns, then theoretically, there are $2^n - 1$ possible combinations of columns for selection (excluding the case where no column is selected).
- Conditions in Queries (WHERE): Each SQL query can include zero or more conditions in the WHERE clause. The number of possible conditions is determined by the number of columns, the type of each column (text, numeric, date, etc.), and the range of operators applicable to these types (like =, <, >, LIKE, IN for text columns; =, !=, <, >, BETWEEN for numeric columns). The complexity increases combinatorially with multiple conditions combined using AND/OR.

**Two Entities** The relations between tables further amplifies the number of possible templates. For example, with two entities, "Company" and "Project," and their associative table "Company_Project", the possibilities for SQL templates

expand significantly due to the introduction of joins and more complex WHERE clauses. Let's break down the possibilities.

- Selection Variations (SELECT): The number of SQL templates grows with the combination of columns selected across the three tables: "Company", "Project" and "Company_Project". If "Company" has n columns, "Project" has m columns, and "Company_Project" has p columns, the possible combinations for selection are $(2^n - 1) \times (2^m - 1) \times (2^p - 1)$.

- Join Conditions (JOIN): The introduction of the associative table "Company_Project" allows for meaningful JOIN operations between "Company" and "Project." Templates can include joins like Company JOIN Company_Project ON condition and Project JOIN Company_Project ON condition, or a multi-table join linking all three. The variety of JOIN conditions adds another layer of complexity to the possible templates.

- WHERE Clause Complexity: With more tables, the WHERE clause can include a wider range of conditions, potentially involving attributes from any of the three tables. The complexity increases with the number of columns and their types across all tables, and combinations of these conditions.

### D.3 Limitations

**Scope of Using Automated SQL Generators** The effectiveness of queries generated by LLMs hinges on the meaningfulness of table and column names to accurately reflect the essence of the entities or concepts they represent. If these names lack contextual clarity, the resulting queries may be impractical. To mitigate this, practitioners can conduct an informal assessment or consult established benchmarks that evaluate LLMs' proficiency in domain knowledge and commonsense reasoning, for instance, (Zhong et al., 2023; Hendrycks et al., 2021).

**Queries Beyond SQL Expressiveness** While SQL and relational algebra offer a wide range of operations enabling the formulation of numerous user queries, certain semantic nuances exceed the expressive capabilities of SQL. The following examples illustrate such limitations:

- Real-world queries often contain ambiguity and subjective interpretations that SQL struggles to accommodate. For instance, the term "major" in the query "What are some major rail projects we've been involved with?" does not directly translate into SQL criteria without additional interpretative steps.

- SQL queries are typically structured to elicit specific, predefined responses, contrasting with the open-ended nature of many real-world inquiries that seek exploratory or comprehensive answers.

## E Breakdown of Scalable Data Generation

The scalability of the proposed data generation process is achieved through two key factors: 1) the creation of templates using various SQL formulations from a modest, manually crafted database with a limited number of attributes and rows, and 2) the generation of text queries with LLMs. The potential number of query permutations can be expressed succinctly by the formula (see Figure 2 for a visual representation):

$$\text{Total Query Variations} = M \times N \times Q, \quad (4)$$

where "M" denotes the array of SQL templates that can be extracted from a given schema, "N" refers to the vast number of text templates that can be produced from a single SQL template, showcasing the flexibility of natural language, and "Q" accounts for the broad spectrum of text queries that can be generated from a single text template, with this diversity arising from different combinations of unique values filling the placeholders.

Let's break down each step of the proposed data generation process:

- 1 schema $\Rightarrow$ M SQL Templates: The number of possible SQL templates (M) that can be generated from a defined schema is influenced by the SQL operators used, the number of tables, and the columns in the database schema. Each combination of tables and columns, along with different SQL operators (like SELECT, WHERE, JOIN), can lead to a unique SQL query template, e.g., the exponential complexity of the SELECT operator, and the combinatorial increase of possible predicates (See Appendix D.2 for details). Note that while the theoretical maximum is

high, practical and meaningful queries will be a smaller subset, as specified in the case study and Appendix D.

- 1 SQL Template ⇒ N Text Templates: For each SQL template, there can be an arbitrary number (N) of textual expressions. This variation arises from the different ways to linguistically express the same SQL query due to the flexibility and richness of natural language, e.g., the variation of synonyms and sentence structures.

- 1 Text Template ⇒ Q Text Queries: Each text template can lead to a number of text queries (Q), depending on the unique values available for each placeholder. With more than one placeholder, the potential for growth in the number of text queries is combinatorial, barring semantic conflicts (context conflict), where certain combinations of column values may not be semantically valid. Overall, this framework demonstrates a combinatorial expansion at each transition stage, especially notable in steps involving natural language due to its inherent variability. However, this growth is tempered by practical constraints such as the meaningfulness of queries (semantic validity) and the actual data distribution in the database. The exponential increase is most pronounced in the transition from text templates to text queries, where the permutations of placeholders can lead to a vast array of unique query possibilities.

# F Database Schema Overview

Figures 3a and 3b illustrate the database schemas used for an actual industrial context and a fabricated scenario, respectively, within our research.

# G Synthetic Processes for Data Generation

## G.1 Aurp Setup

The generation starts from a company profile, i.e., a fictitious company named Aurp, and then goes to structural knowledge to create a relational database, including organizational structures, employees, clients and projects. Finally, all the information above is used to generate synthetic project documents. Below is a detailed process.

1. **Generating Company Profile**: Initially, a company profile for Aurp is created, detailing its foundation year, headquarters location, CEO, number of employees, and the services it offers, such as bespoke architectural solutions, sustainable urban planning, and structural health monitoring.

2. **Generating Organizational Structure**: Next, a project-oriented organizational structure is established, naming key positions and employees within the company, similar to real-world firms. This includes a wide range of roles from executive positions to specialized engineers and support staff.

3. **Generating Employee Information**: For each employee listed in the organizational structure, detailed job titles, departments, and direct supervisors or managers are fabricated, creating a network of relationships and reporting lines within the company.

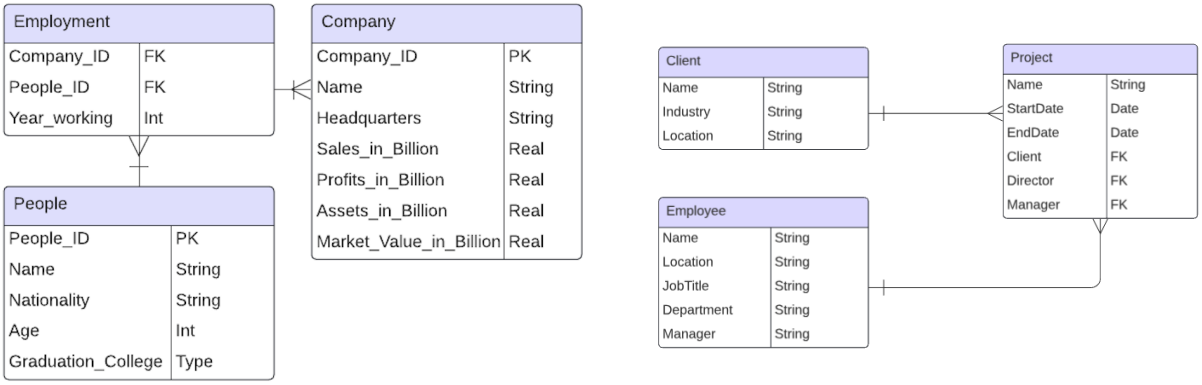
4. **Generating Client Information**: The process creates a list of ten clients across various industries—ranging from technology and real estate to hospitality and healthcare—each with specified locations, thereby illustrating the company's diverse portfolio.

5. **Generating Projects**: Specific projects are then devised, including names, locations, start and end dates, clients, project directors, and project managers. This step integrates previously generated data (client and employee information) to create realistic project scenarios.

6. **Generating Project Reports**: This step involves synthesizing data from the previous steps to produce detailed analyses, updates, and outcomes of the various projects, which may or may not contain detailed information for clients and employees.

## G.2 Context-Augmented Spider

Since relational databases have been constructed in Spider, simple company and employee profiles with factual knowledge are manually crafted using the stringified templates in Table 8. The simple profiles are then used to prompt GPT-4 to generate Wikipedia-style pages containing 200-400 words. Knowledge required for queries is sparse in all documents.




(a) Spider. (b) Aurp.

Figure 3: Entity-Relationship Diagrams for Data Generation

```
Company Profile
**Name**: {company_name}
**Headquarters**: {headquarter}
**Industry**: {industry}
**Sales in Billion**: {sales}
**Profits in Billion**: {profits}
**Assets in Billion**: {assets}
**Market Value in Billion**: {market_value}

Employee Profile
**Name**: {name}
**Age**: {age}
**Nationality**: {nationality}
**Graduation College**: {graduation_college}
```

Table 8: Templates for company and employee profiles. Placeholders in curly brackets will be replaced by factual information in Spider databases.

**Evaluation of Factuality in Generated Documents** To evaluate the factuality of generated documents, a Machine Reading Comprehension (MRC) task is setup, where the generated document is given as context to answer the questions encoding the facts it generate on. Note that the correct answer can only source from the generated document.

## H Reproducing Data and Results

### H.1 Reproducing Data

The following steps illustrate how to use our open-source tool to reproduce the data. The datasets are also available along with the tool.

```
from grammar.db_tool import DBTool
from grammar.llm import AnyOpenAILLM
from grammar.sql_template_generator
    import SQLTemplateGenerator
from grammar.text_template_generator
    import TextTemplateGenerator
from grammar.qa_generator import
    QADataGenerator
```

```
llm = AnyOpenAILLM(model_name = "gpt4-
    short")

setup_env = "aurp"
if setup_env == "spider" or setup_env ==
     "spider_closed":
    database_name = 'spider'
    connection_string = f'sqlite:///{
    database_name}/rel_database/
    company_employee.sqlite'
    schemas = [('company',), ('people',)
    , ('company', 'people')]
elif setup_env == "aurp":
    database_name = 'Aurp'
    connection_string = "mysql+pymysql
    ://root:password@localhost:3306/Aurp
    "
    schemas = [('client',), ('employee'
    ,), ('project', )]
db_tool = DBTool(connection_string)
```

```
# Step 1: Generate SQL Query Templates
file_path = f"{setup_env}/
    SQLTemplateGenerator/sql_templates.
    json"
sql_template_generator =
    SQLTemplateGenerator.from_file(
    file_path, sql_connection=
    connection_string, llm=llm)
entities_to_sql_templates =
    sql_template_generator.
    generate_batch(schemas, override=
    False, verbose=True)
sql_templates = [tpl for entity, tpls in
     entities_to_sql_templates.items()
    for tpl in tpls]
```

Below are examples of generated SQL templates in JSON format, where the templates are keyed by table names.

```
{
    "('client',)": [
        "SELECT Location FROM
    Client WHERE Name = '[Client.
    Name]';",
```

```
        "SELECT Industry FROM
    Client WHERE Name = '[Client.
    Name]';"
     ],
     "('employee',)": [
         "SELECT JobTitle FROM
    Employee WHERE Name = '[
    Employee.Name]';",
         "SELECT Department FROM
    Employee WHERE Name = '[
    Employee.Name]';",
         "SELECT
    SupervisorOrManager FROM
    Employee WHERE Name = '[
    Employee.Name]';"
     ],
     ...
}
```

```
# Step 2: Generate Text Query Templates
linguistic_attr = "long"
file_path = f'{setup_env}/
    TextTemplateGenerator/{
    linguistic_attr}.json'
text_template_generator =
    TextTemplateGenerator.from_file(
    file_path=file_path, verbalize_attrs
    =linguistic_attr, llm=llm) # Load
    existing generations to avoid re-
    generation
sql_to_text_templates =
    text_template_generator.
    generate_batch(sql_templates,
    verbose=True, num_generations=3,
    override=False)
text_template_generator.save(file_path=
    file_path, override=True)
```

Below are examples of generated text templates saved in JSON format.

```
{
     "SELECT StartDate FROM
    Project WHERE Name = '[Project
    .Name]';": [
         "Start date for project '
    [Project.Name]'",
         "Look up start date of '[
    Project.Name]'",
         "Get '[Project.Name]'
    start date"
     ],
     ...
}
```

```
# Step 3: Generate Evaluation Data (Text
    Queries and Answers)
save_file = f"{linguistic_attr}.json"
qa_generator = QADataGenerator(db_tool)
all_answers_to_text_queries =
    qa_generator.generate(
    sql_to_text_templates)
qa_generator.save(
    all_answers_to_text_queries,
    database_name, save_file, overwrite=
    True)
```

Below are examples of generated query-answer pairs saved in JSON format.

```
[
    [
        "[('Maldives',)]",
        [
            "Get 'Blue Horizon
    Hotels' location details",
            "Find which location
    'Blue Horizon Hotels' is
    located in",
            "Determine the
    location of 'Blue Horizon
    Hotels'"
        ]
    ],
    ...
]
```

### H.2 Reproducing Experiment Results

The tool aslo provides two high-level Python objects to reproduce results from our experiments: `TaggedGroup` to perform tagging on each semantic groups and generate metrics for modular evaluation. The code below demonstrates an example to reproduce the results in Table 3. The input to `get_eval_results` and `TaggedGroup` is a list of `RAGResult` objects, which is a data class including a query, the ground-truth answer and the RAG response.

```
import json
from grammar.eval.result import
    RAGResult
from grammar.eval.tag_group import
    TaggedGroup
from grammar.eval.match import
    SemanticsMatch

def get_eval_results(eval_results,
    linguistic_attr, root_dir, file_path
    ):
    tagged_group = TaggedGroup(
    eval_results)
    semnatics_match = SemanticsMatch.
    from_file(root_dir=root_dir,
    verbalize_attrs=linguistic_attr)

    for eval_result in eval_results:
```

```
        # sleep for 20 seconds after 9
    examples
        # if results.index(result) % 9
    == 0 and results.index(result) != 0:
        #     print("Sleeping for 20
    seconds")
        #     time.sleep(20)
        #     print("Waking up")
        eval_result.
    judge_retrieval_response(
    tagged_group=tagged_group, method='
    use_exist')
        eval_result.judge_rag_response(
    semnatics_match)

    num_retrieval_failure = sum([result.
    retrieval_judgement==0 for result in
     eval_results])
    print(f"Retrieval failed in {
    num_retrieval_failure} out of {len(
    eval_results)} examples")
    num_rag_failure = sum([result.
    judgement=="Incorrect" for result in
     eval_results])
    print(f"RAG failed in {
    num_rag_failure} out of {len(
    eval_results)} examples")
    semnatics_match.save(root_dir=f'{
    root_dir}', override=True)
    # semnatics_match.llm.
    gpt_usage_record.write_usage(
    model_name='chatgptk' )

    # save results
    results = [result.asdict() for
    result in eval_results]
    # ensure json serializable
    for result in results:
        result['true_document_ids'] =
    list(result['true_document_ids'])
        result['retrieved_document_ids']
     = list(result['
    retrieved_document_ids'])
    with open(file_path, 'w') as f:
        json.dump(results, f, indent=4)

    return eval_results, tagged_group

root_dir = 'aurp'
closed_domain = True
results, metric = get_eval_results( '
    short', root_dir, file_path=f'{
    root_dir}/eval_results/
    results_short_balanced.json')

# re-produce metrics in Table 3
print('Baseline Accuracy: ', metric.
    get_accuracy())
print('Accuracy (Remove LLM Errors): ',
    metric.get_accuracy(for_retrieval=
    True))
print('Removing Gap Examples: ', metric.
    get_robustness())
print('Removing LLM Errors & Gap
    Examples: ', metric.get_robustness(
    for_retrieval=True))
```