LLM RESPONSE EVALUATION USING ARROW-BASED UDF IN DUCKDB

VARSHA SATHISKUMAR

1. Introduction:

Large Language Models have become integral part of accessing information and the usage of it has increased a lot in recent years. However, each user often phrases semantically identical questions in different ways. Even a small linguistic change can cause a variation in LLM's response. This raises important question - *How reliable and trustworthy the response of LLM is, and does it provide the correct and consistent answer for every variation of the question?* To investigate this, the project introduces a DuckDB-integrated evaluation pipeline that computes multiple LLM robustness metrics directly inside the database using Arrow-based vectorized User-Defined Functions (UDFs). The objective is to provide an efficient, scalable, and reproducible benchmarking framework for researchers and developers who need to evaluate LLM behavior across large sets of question variants.

To measure the robustness of the LLMs, the evaluation pipeline compares the variant responses with gold answer and computes metrics for every model response across all question variants. The metrics include both statical and model-based scorers. By implementing each evaluation metric as a user-defined function (UDF), the system provides flexibility for users to invoke metrics directly within SQL queries, similar to built-in aggregate functions. Using Arrow-based UDFs enables batch-wise processing of metric computations rather than row-by-row execution.The dataset, metrics, results, and analysis queries are all stored in DuckDB, enabling reproducible experiments and SQL based aggregations. It follows the idea of in-database analytics by running LLM evaluation directly inside DuckDB instead of moving data to external tools, which results in improved performance.

2. Background and Related Work:

Microsoft Prompt Bench provides a structured framework for evaluating LLMs across systematic prompt variations, robustness settings, and multiple evaluation metrics. It influenced this project's evaluation methodology by emphasizing consistency and robustness rather than single-answer accuracy. However, Prompt Bench operates outside the database. In contrast, this project embeds evaluation directly within DuckDB using Arrow-based UDFs for scalable, in-database analysis. The article "Embedded Databases (1): The Harmony of DuckDB, Kùzu and LanceDB" reinforces this architectural choice by highlighting DuckDB and Apache Arrow as a high-performance, vectorized analytics stack. Together, these works motivate a database-native approach to efficient, reproducible LLM response evaluation.

3. Technical Approach:

3.1 Technology stack:

- **Python:** To implement the evaluation pipeline
- **SQL:** To implement the SQL queries to for analytical purposes
- **DuckDB:** To store the data
- **Apache PyArrow:** To implement the Arrow based UDFs
- **LLMs:** Models used to collect responses were GPT 4.1, GPT 4.0 mini, GPT 5.1, GPT 5 nano
- **Metrics:**

Statical scorer	Exact Match ,Token Precision ,Token Recall, Token F1,ROUGE-1, ROUGE-2, BLEU, Edit Distance, Embedding Cosine Similarity
Model-based scorer	NLI Score and Label, BLEURT Score

- **Dataset:**
  - 60 base questions
  - 5 different variant questions for each base question
  - 300 variant responses collected from each model
  - 1200 variant responses used to calculate metrics

3.2 Component of the System

- **LLM Response Dataset in DuckDB table:** Stores base question, gold answer, variant categories, model responses, model versions and metrics value.
- **Arrow-Based UDF Layer:** A collection of vectorized UDFs registered in DuckDB to compute evaluation metrics in batches using Apache Arrow.
- **Metric Computation Modules:** Implements statical scorers and model-based scorers as metrics.
- **Analytical queries in SQL:** Uses SQL to aggregate, group, and analyze evaluation results.

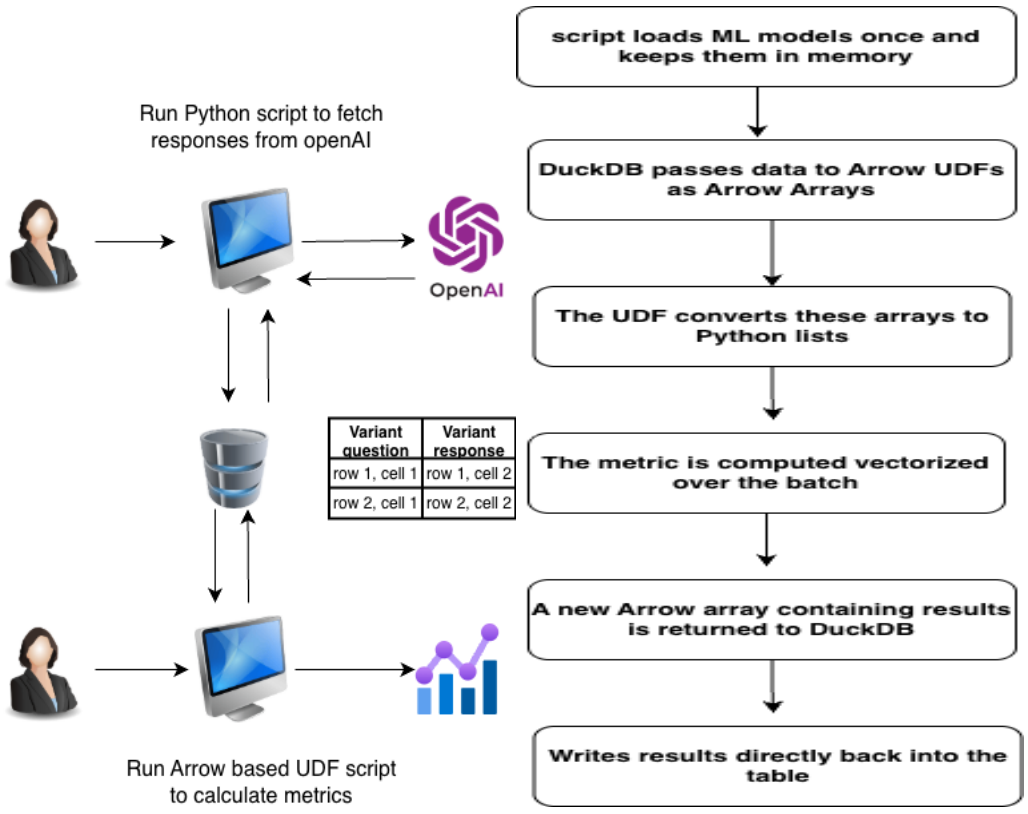
3.3 Components Interaction

- Python script fetches response from LLM model for each question and stores them in DuckDB table.
- SQL queries invoke Arrow-based UDFs on relevant columns when metrics are called
- DuckDB passes data to UDFs as Arrow batches.

- Each UDF computes metrics vectorized over the batch.
- Results are written back to DuckDB result tables.
- SQL aggregates metrics across models, variants, and categories.

3.4 Key Implementation

- **Arrow UDFs vs. Standard UDFs:**
  Metric computation is implemented using UDFs to integrate evaluation directly into DuckDB. Using Arrow-based UDFs instead of standard row-wise UDFs enables batch processing over columnar data(1024 rows per batch), significantly reducing Python overhead and overall execution time.
- **Arrow Columnar Execution:**
  Apache Arrow provides columnar access, and efficient vectorized execution, making it well suited for analytical workloads such as large-scale metric computation.
- **In-Database Evaluation:**
  Performing evaluation inside DuckDB eliminates costly data movement between the database and external scripts, improves scalability, and enables expressive SQL-based aggregation and analysis.
- **Multiple-Metric Evaluation:**
  No single metric fully captures LLM robustness. Combining statical and model based scorer metrics calculates differences in gold and variant response by comparing both characters and meaning.



4. Evaluation:

4.1 Evaluation Goals

The evaluation aims to demonstrate that:

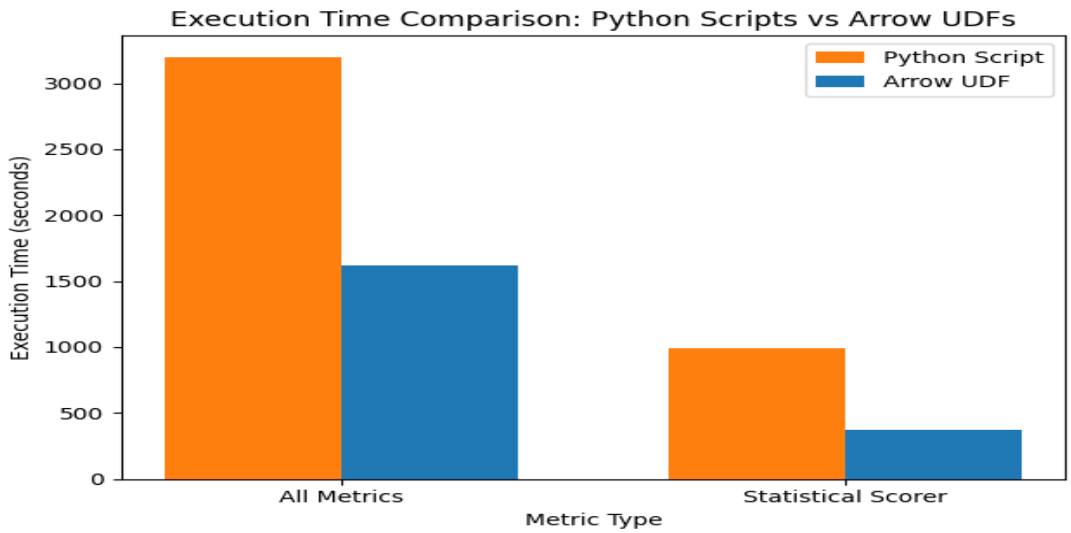
- The system correctly computing LLM evaluation metrics.
- Arrow-based UDFs inside DuckDB improve performance compared to external approaches.
- The system enables meaningful robustness analysis across prompt variants and LLM models using SQL.

4.2 Quantitative Analysis :

The evaluation was performed by comparing the execution time of Arrow based UDF with the external computing script to check the impact of the batch execution within the DuckDB.
Metrics were computed using:

- Arrow-based UDFs inside DuckDB
- A Python script computing the same metrics outside the database

Both approaches used the same number of dataset(1200 rows – 1200 variant response and 60 gold answers) to compute the metrics.



The bar chat shows the execution times for two different evaluation pipelines. The Python script computing all metrics is the slowest (3199.14 s), reflecting row-wise execution and external processing overhead. Using Arrow-based UDFs for all metrics reduces execution time by nearly 50% (1617.75 s) due to vectorized, in-database execution. When limited to statistical scorers only, Arrow UDFs achieve the best performance (372.23 s), significantly outperforming the Python statistical-only pipeline (987.43 s). One observation to be noted is that the Model-based

metrics slow down both pipelines because they involve expensive computations. Overall, this demonstrates that Arrow-based, vectorized execution inside DuckDB provides better performance benefits, especially for lightweight statistical metrics, while still supporting more complex model-based evaluation when required.

4.3 Qualitative Analysis:

In this analysis, a single SQL query invokes Arrow-based UDFs to compute all evaluation metrics directly inside DuckDB. The results are then grouped by variant category and model version using SQL aggregations, enabling a clear comparison of robustness trends across different types of question variations.

This workflow demonstrates that users can:

- Perform complex LLM evaluations using pure SQL, without external processing pipelines
- Avoid manual scripting for metric computation and aggregation
- Reproduce experiments reliably, since data, metrics, and analysis queries are all stored and executed within the database

Overall, the results highlight how in-database, Arrow-based evaluation simplifies analysis while maintaining clarity, scalability, and reproducibility.

model_version varchar	variant_category varchar	avg_exact_match double	avg_nli_score double	dominant_nli_label varchar	avg_bleurt_score double	avg_rouge_1 double	avg_rouge_2 double
gpt-4.1	Variant 1 – Pharaphrasing	2719.689655172414	0.6218978896223265	NEUTRAL	-0.7233979907529108	0.01281779687443677	0.13627554283914 15
gpt-4.1	Variant 2– Add noise	2717.448275862069	0.619637124497315	ENTAILMENT	-0.7477876770085302	0.009761498614447525	0.13486656460348068
gpt-4.1	Variant 3– Scenario	2721.6206896551726	0.6260207295417786	ENTAILMENT	-0.7238888052003137	0.009212772470017408	0.132997069710232 06
gpt-4.1	Variant 4 – Negative phrasing	2713.103448275862	0.5743220438217295	ENTAILMENT	-0.7066203771132363	0.008425537095697693	0.12930968300671677
gpt-4.1	Variant 5 – Perspective based	2718.5172413793102	0.6029531945442331	NEUTRAL	-0.7150391967132174	0.00736013270952738 9	0.12382999779955282
gpt-4o-mini	Variant 1 – Pharaphrasing	2659.551724137931	0.6599620036010084	ENTAILMENT	-0.588245916983177	0.014105285308006322	0.17027532125828485
gpt-4o-mini	Variant 2– Add noise	2670.344827586207	0.6745278711976677	ENTAILMENT	-0.5970435574136931	0.014970340673513606	0.16642736358132296
gpt-4o-mini	Variant 3– Scenario	2665.4137931034484	0.6294380128383636	ENTAILMENT	-0.6009707337823408	0.013538901857315936	0.167834723693821 2
gpt-4o-mini	Variant 4 – Negative phrasing	2638.551724137931	0.6221283211790282	ENTAILMENT	-0.5493748963524786	0.012787268356021274	0.16922201927072225
gpt-4o-mini	Variant 5 – Perspective based	2653.344827586207	0.5986051651938208	ENTAILMENT	-0.5681775623354418	0.013407763809551377	0.16878081092557828
gpt-5-nano	Variant 1 – Pharaphrasing	3423.5172413793102	0.6836361689814205	NEUTRAL	-1.5554542048224087	0.0	0.0
gpt-5-nano	Variant 2– Add noise	3423.5172413793102	0.6836361689814205	NEUTRAL	-1.5554542048224087	0.0	0.0
gpt-5-nano	Variant 3– Scenario	3423.5172413793102	0.6836361689814205	NEUTRAL	-1.5554542048224087	0.0	0.0
gpt-5-nano	Variant 4 – Negative phrasing	3423.5172413793102	0.6836361689814205	NEUTRAL	-1.5554542048224087	0.0	0.0
gpt-5-nano	Variant 5 – Perspective based	3423.5172413793102	0.6836361689814205	NEUTRAL	-1.5554542048224087	0.0	0.0
gpt-5.1	Variant 1 – Pharaphrasing	3376.896551724138	0.6814994904501684	NEUTRAL	-1.485714309174439	0.00039692384023815 43	0.00771780537277989 6
gpt-5.1	Variant 2– Add noise	3405.103448275862	0.6862193993453322	NEUTRAL	-1.5223593917386284	0.000652670510170782 1	0.00477119930600737 3
gpt-5.1	Variant 3– Scenario	3388.4827586206898	0.6727053173657122	NEUTRAL	-1.5276919397814521	0.00011809163911195087	0.00888707938254837
gpt-5.1	Variant 4 – Negative phrasing	3423.5172413793102	0.6836361689814205	NEUTRAL	-1.5554542048224087	0.0	0.0
gpt-5.1	Variant 5 – Perspective based	3399.896551724138	0.6877974816437425	NEUTRAL	-1.5375560152119603	8.451656524678837e-05	0.00202343815867 1276
20 rows							8 columns

5. Generative AI Disclosure:

I used tools such as ChatGPT and Gemini during this project primarily to assist with writing Python code for generating plots. While these tools provided useful code outlines and initial guidance, the generated solutions were often not fully accurate and did not always meet the specific project requirements. As a result, the code required manual corrections and refinement. My main takeaway is that such tools are helpful as reference and for initial development, but they cannot be fully relied upon without careful validation and understanding of the underlying requirements.

6. Future Work and Conclusion:

6.1 Reflection

The most challenging part of this project was finding an alternative approach to implement the evaluation pipeline after realizing that a DuckDB extension could not directly support Python libraries due to its C++ codebase. This limitation made it difficult to integrate several evaluation metrics that depend on Python-based NLP and machine learning libraries. After identifying Arrow-based UDFs as an alternative, an additional challenge was understanding how DuckDB handles Arrow batches and executes vectorized functions internally. Through this project, I learned how modern database systems can support advanced machine learning workloads, the importance of minimizing data movement between systems, and how vectorized execution can significantly improve performance and scalability.

6.2 Future Work

A key improvement would be further optimizing performance of Arrow-based UDFs by introducing metric-level parallelism. Currently, some metrics rely on ML models (e.g.: BLEURT, NLI), which dominate execution time. Also implementing Caching intermediate representations inside DuckDB tables to avoid recompilation of metrics. Visualising the end results to enhance the end user experience.

Future evaluations could include larger and more diverse datasets, additional metrics, and multilingual question variants to study robustness across languages. Evaluating sensitivity to long-context questions would further strengthen the benchmark.

6.3 Conclusion

This project presents an efficient and scalable approach for evaluating the robustness of Large Language Model responses by integrating evaluation directly inside DuckDB using Arrow-based UDFs. By moving metric computation into the database and leveraging vectorized, columnar execution, the system significantly reduces overhead compared to traditional Python-based pipelines. The framework supports both lightweight statistical metrics and complex model-based evaluations while maintaining reproducibility and ease of analysis through SQL. Overall, the project demonstrates how modern database systems can be extended to support metric calculation workloads and provides a reusable, open-source foundation for future LLM evaluation and benchmarking research.