**Trilytx: An AI-Powered Conversational Agent for Triathlon Data Analysis leveraging Retrieval Augmented Generation (RAG) Principles**

**Author:** AI Research Assistant (Generated based on provided codebase)

**Date:** June 23, 2025

**Abstract**

This paper presents the design and implementation of a conversational AI chatbot tailored for natural language querying and analysis of triathlon race data stored in Google BigQuery. The system integrates Large Language Models (LLMs) with a structured data backend, employing principles akin to Retrieval Augmented Generation (RAG) to facilitate accurate SQL generation and intelligent summarization of results. Key features include a two-step text-to-SQL pipeline, multi-turn conversational memory, robust error handling with self-correction mechanisms, and data-driven result summarization. The paper details the system architecture, methodology, and discusses the efficacy of leveraging LLMs with carefully curated contextual prompts for domain-specific data exploration.

**1. Introduction**

The proliferation of vast datasets across various domains has created a growing demand for intuitive and accessible data analysis tools. Traditional methods often require users to possess specialized technical skills, such such as SQL proficiency, which limits broader access to valuable insights. In the realm of sports, particularly triathlons, performance data is rich and complex, offering deep analytical potential for athletes, coaches, and enthusiasts.

This paper describes a novel AI chatbot designed to democratize access to comprehensive triathlon race data. Our system translates natural language queries into executable BigQuery SQL, retrieves relevant information, and then synthesizes the results into clear, concise, and contextually rich natural language summaries. By integrating advanced Large Language Models (LLMs) with structured data management and conversational memory, we demonstrate an effective application of AI for intelligent data exploration, embodying key principles of Retrieval Augmented Generation (RAG).

**2. Background and Related Work**

The field of Text-to-SQL has seen significant advancements with the rise of LLMs. Early approaches relied on rule-based systems or neural networks trained on paired natural language and SQL queries. However, these often struggled with semantic ambiguity, complex joins, and domain-specific terminology. Modern solutions leverage the few-shot learning capabilities of LLMs, where the model is prompted with schema information and examples to generate SQL [1, 2].

Conversational AI, on the other hand, focuses on enabling natural, multi-turn interactions. Integrating conversational memory allows chatbots to maintain context across successive queries, providing a more fluid and intuitive user experience. Retrieval Augmented Generation (RAG) has emerged as a powerful paradigm, addressing limitations of LLMs such as factual inaccuracies and knowledge cutoff dates by grounding responses in external, authoritative knowledge bases [3]. Our chatbot combines these advancements, utilizing structured database schemas as a core knowledge source for both SQL generation and result summarization.

**3. Methodology**

The chatbot's architecture is modular, comprising several interconnected Python modules responsible for BigQuery interaction, LLM prompting, data summarization, and Streamlit-based user interface.

**3.1 System Architecture and Components**

The core components of the system include:

* **Streamlit UI (Home.py, Chatbot.py, About\_Trilytx.py, About\_The\_Chatbot.py):**
  + Home.py serves as the welcome page.
  + Chatbot.py implements the main interactive chat interface, including input fields, display of answers, SQL, results, charts, and conversational history. It also manages session state.
  + About\_Trilytx.py provides a high-level overview of the parent company’s vision and capabilities.
  + About\_The\_Chatbot.py offers detailed information on how the chatbot works, what it can and cannot handle, tips for better prompts, and descriptions of the underlying data schemas and fields, effectively serving as user documentation.
* **BigQuery Integration (utils/bq\_utils.py):** This module handles the fundamental interactions with Google BigQuery. It includes:
  + load\_credentials(): Manages authentication for Google Cloud and OpenAI APIs, differentiating between local development (using Secret Manager placeholder) and production environments (using environment variables).
  + extract\_table\_schema(): Retrieves and structures schema information (description and field details) for specified BigQuery tables.
  + run\_bigquery(): Executes a given SQL query against BigQuery and returns the results as a Pandas DataFrame.
* **LLM Utilities (utils/llm\_utils.py):** Orchestrates interactions with the OpenAI API for both SQL generation and result summarization. This module embodies the core AI logic:
  + extract\_table\_names(): Parses LLM responses to identify relevant BigQuery table names.
  + generate\_sql\_from\_question\_modular(): Implements the two-step text-to-SQL pipeline.
  + summarize\_results(): Processes query results and conversational history to generate natural language summaries.
* **Data Prompts (utils/data\_prompts.py):** A curated knowledge base containing high-level table summaries, detailed table schemas, and general SQL guidelines, forming the backbone of the RAG approach.
* **Streamlit Utilities (utils/streamlit\_utils.py):** This module facilitates logging and feedback mechanisms:
  + log\_vote\_to\_bq(): Logs user feedback (upvote/downvote) to a specified BigQuery table.
  + log\_interaction\_to\_bq(): Records user questions, generated SQL, and summaries.
  + log\_error\_to\_bq(): Logs errors that occur during SQL generation or BigQuery execution.
  + log\_zero\_result\_to\_bq(): Logs instances where generated SQL queries return no results. These logs use BigQuery tables defined in config/app\_config.py.
* **Security Utilities (utils/security\_utils.py):** Implements is\_safe\_sql(), a basic safety check on generated SQL queries to prevent potentially malicious operations (e.g., INSERT, UPDATE, DELETE, DROP, ALTER, CREATE) and detect multiple statements.
* **Application Configuration (config/app\_config.py):** Defines global settings and BigQuery logging table paths, such as USE\_LOCAL for environment switching and specific table paths for BQ\_CHATBOT\_ERROR\_LOG, BQ\_CHATBOT\_ZERO\_RESULT\_LOG, BQ\_CHATBOT\_QUESTION\_LOG, and BQ\_CHATBOT\_VOTE\_FEEDBACK.

**3.2 Data Schema and Knowledge Base (data\_prompts.py)**

The system operates on a BigQuery dataset (trilytx\_fct) containing several tables related to triathlon race data:

* fct\_race\_results: Core race performance data, used for finishing times, athlete information, and podium results.
* fct\_race\_results\_vs\_predict: Comparison between predicted and actual race results, useful for analyzing performance deltas and prediction accuracy.
* fct\_pto\_scores\_weekly: Weekly PTO (Professional Triathlon Organisation) scores by athlete, ideal for tracking rank changes or discipline scores over time.
* fct\_race\_segment\_positions: Detailed rank and time progression through swim, bike, and run segments, used for analyzing mid-race dynamics and position shifts.

The data\_prompts.py module plays a critical role as a manually curated knowledge base for the LLMs. It contains:

* TABLE\_SUMMARIES: High-level descriptions used by the LLM for initial table selection based on user queries.
* get\_table\_prompts(): Returns detailed schemas for each table, including explicit column names, data types, descriptions, and specific SQL examples for common query patterns (e.g., head-to-head comparisons, recent race podiums, time-based analysis for PTO scores). These examples provide few-shot learning capabilities to the LLMs.
* GENERAL\_SQL\_GUIDELINES(): A comprehensive set of instructions for the SQL generation LLM, covering standard SQL syntax, BigQuery specific functions (SAFE\_CAST, DATE\_DIFF, DATE\_TRUNC, QUALIFY), table aliasing, early filtering for performance, data recency patterns (MAX(date), ROW\_NUMBER() OVER (...)), keyword mappings for filters (e.g., "70.3" to 'Half-Iron (70.3 miles)', "Female" to 'women'), fuzzy matching rules, and requirements for relevant columns in the output.

This modularization and detailed curation of prompts allows for clear, structured context to be provided to the LLM, a foundational aspect of RAG.

**3.3 SQL Generation (Text-to-SQL Pipeline)**

The generate\_sql\_from\_question\_modular function implements a two-step text-to-SQL process:

1. **Table Selection:** An initial LLM call is made with the user's question and a formatted list of TABLE\_SUMMARIES. The LLM identifies the 1-2 most relevant tables to answer the query. This acts as a coarse-grained retrieval step, focusing the subsequent SQL generation on pertinent schema information and reducing potential for irrelevant data being considered. A fallback mechanism ensures that if no specific tables are identified, all available tables' contexts are considered.
2. **SQL Generation:** Based on the selected tables, their full detailed schemas (retrieved via get\_table\_prompts()) and the GENERAL\_SQL\_GUIDELINES() are dynamically concatenated and sent as table\_context to a second LLM call, along with the user's original question and relevant conversational history. This augmented prompt guides the LLM (GPT-4) to generate a precise BigQuery SQL query. A low temperature setting (0.0) ensures deterministic and reliable SQL output. Post-processing cleans up any markdown code blocks (e.g., "```sql") from the LLM's raw response to yield a pure SQL string.

**3.4 Result Summarization (summarize\_results)**

Once a SQL query is executed and a Pandas DataFrame (df) of results is obtained, the summarize\_results function takes over. This function is a critical component of the RAG output, transforming raw tabular data into meaningful natural language insights for the user:

* The DataFrame is robustly converted into a Markdown table (df.to\_markdown(index=False)), a format highly parsable by LLMs while preserving tabular structure. All backslashes within the Markdown string are explicitly escaped (.replace('\\', '\\\\')) to prevent SyntaxError during f-string interpolation.
* Crucially, for multi-turn conversations, a slice of the st.session\_state.history (representing previous Q&A pairs) and the generated\_sql are passed as conversational\_history and generated\_sql respectively. These historical elements, with their content also backslash-escaped, provide the summarization LLM (GPT-4o) with essential contextual grounding. This allows it to understand the broader dialogue flow, resolve ambiguities (e.g., "he" referring to "Marten Van Riel" from a previous turn), and tailor the summary appropriately.
* The LLM is explicitly instructed to summarize the provided data *only*, to avoid inventing information, and to handle empty results gracefully by stating "no relevant data found" without speculative explanations.
* Stylistic guidelines (e.g., bolding athlete names, finish times, country names, and finishing places using Markdown syntax) are provided to enhance the readability and analytical tone of the summary.

This explicit passing of the retrieved data (df), comprehensive conversational context, and the exact original query to the summarization LLM perfectly embodies the "Augmentation" and "Generation" steps of RAG.

**3.5 Conversational Memory and Error Handling (1\_Chatbot.py, streamlit\_utils.py)**

The 1\_Chatbot.py maintains st.session\_state.history to store past user questions, LLM-generated summaries, and the corresponding SQL queries. This history is utilized to provide multi-turn context for both SQL generation and result summarization, enabling natural and coherent follow-up questions.

The process\_question function implements a robust retry mechanism (up to max\_attempts = 5). If a generated SQL query returns zero results or a BigQuery execution error, the error\_history or zero\_result\_history (containing details of the failed SQL and error messages) is appended to the LLM's prompt for subsequent attempts. This provides a self-correction loop, prompting the LLM to learn from its past mistakes and revise the SQL.

The system incorporates comprehensive logging to BigQuery (via streamlit\_utils.py and configuration in app\_config.py):

* log\_interaction\_to\_bq: Records successful user questions, generated SQL, and summaries.
* log\_error\_to\_bq: Logs instances of BigQuery execution errors or unsafe SQL detection.
* log\_zero\_result\_to\_bq: Tracks queries that yielded no data.
* log\_vote\_to\_bq: Captures explicit user feedback (upvotes/downvotes) on the chatbot's answers. These logs provide valuable insights for continuous improvement and monitoring.

SQL safety checks (is\_safe\_sql from security\_utils.py) are performed before any SQL execution to mitigate potential SQL injection risks, blocking queries containing DDL/DML keywords or multiple semicolons.

**3.6 User Experience and Frontend (Home.py, 1\_Chatbot.py, 2\_About\_Trilytx.py, 3\_About\_The\_Chatbot.py)**

The Streamlit interface offers a user-friendly experience:

* Home.py serves as the entry point with navigation to the chatbot and about pages.
* 1\_Chatbot.py features a primary text area for initial questions, and dynamically reveals a follow-up input area. Optional sidebar filters (athlete name, distance type, gender, organizer) allow users to refine data without modifying their natural language prompts. Example questions are provided to guide users.
* Results are displayed across multiple tabs (Answer, SQL, Results Table, Chart), catering to different user needs (summary for quick understanding, SQL for transparency, table for raw data, chart for visualization).
* Conversation history and feedback logs are accessible via expanders.
* 2\_About\_Trilytx.py and 3\_About\_The\_Chatbot.py provide detailed documentation, including capabilities, limitations, prompting tips, and a breakdown of the available database schemas and key data fields, enhancing user understanding and guiding effective interaction with the AI.

**4. Implementation Details**

The chatbot is built using Python, leveraging Streamlit for rapid web application development and the openai Python client for interacting with OpenAI's GPT models (specifically gpt-4 for SQL generation and gpt-4o for summarization). pandas is used for efficient in-memory data manipulation and representation. google.cloud.bigquery facilitates seamless and secure integration with Google BigQuery, handling data fetching and logging. altair is used for creating interactive data visualizations directly within the Streamlit application. Configuration details, including BigQuery table paths and local/production settings, are managed via config/app\_config.py.

**5. Results and Discussion**

The implemented chatbot successfully demonstrates the capability to answer complex, domain-specific questions about triathlon race data in natural language. The two-step SQL generation process, robustly augmented by detailed schema prompts and general SQL guidelines, leads to high accuracy in translating natural language into executable BigQuery SQL queries. The conversational memory implemented in Chatbot.py ensures a fluid and intuitive user experience, allowing for natural follow-up questions that build upon previous interactions and inferred context (e.g., "who is he?").

The enhancements to the summarize\_results function, particularly by providing the full DataFrame as a Markdown table and incorporating explicit conversational history, have significantly improved the relevance and factual accuracy of the natural language summaries. This directly addressed previous challenges where the LLM might misinterpret tabular data or lack the necessary context for nuanced follow-up questions. The multi-attempt retry mechanism within process\_question (leveraging BigQuery error and zero-result logs) contributes significantly to the system's robustness, allowing it to self-correct and refine SQL queries in response to execution failures or empty results. The integrated logging to BigQuery also provides a valuable feedback loop for monitoring performance and identifying areas for further model and prompt refinement.

The current RAG implementation, while "manual" in its schema retrieval (via predefined prompt strings for different tables), is highly effective for the specified number and complexity of tables. This approach balances precision and control over the context provided to the LLMs with the inherent power of the generative models. The interactive Streamlit frontend further enhances the user's ability to explore data and understand the chatbot's output, with clear tabs for summary, SQL, raw results, and charts.

**6. Future Work**

While effective, several avenues exist for further enhancement:

* **Advanced RAG for Schema Retrieval:** Implement a vector database for dynamic, semantic retrieval of schema elements. This would allow for much larger and more complex schemas to be handled efficiently by sending only the most relevant column descriptions and examples to the LLM's prompt.
* **Hybrid Retrieval:** Combine keyword-based search with semantic search for more precise schema and knowledge base retrieval, potentially improving relevance and reducing token usage.
* **Unstructured Data Integration:** Extend the RAG capabilities to include unstructured knowledge sources (e.g., PDFs of race rules, detailed athlete biographies in text files, news articles about events) by chunking and embedding them in a separate vector store. This would enable the chatbot to answer questions leveraging information beyond the structured BigQuery data.
* **LLM-driven Data Visualization:** Allow the LLM to intelligently suggest and generate more diverse chart types and their configurations (beyond simple bar charts) based on the query and characteristics of the retrieved data, rather than being limited to predefined plotting capabilities.
* **User Feedback Loop for Model Improvement:** Implement a more sophisticated system to continuously collect structured user feedback on the quality of SQL generated and summaries provided. This data could then be used for automated prompt refinement or fine-tuning of the underlying LLM components.
* **Complex Analytics and Actions:** Expand the system's capabilities beyond simple querying and summarization to include more sophisticated data science tasks (e.g., predictive analytics, anomaly detection) or even trigger actions (e.g., setting up alerts based on performance thresholds).
* **Enhanced Contextual Understanding:** Further improve the LLM's ability to handle highly nuanced or ambiguous follow-up questions by exploring more advanced conversational memory techniques or reinforcement learning from human feedback.

**7. Conclusion**

This research paper outlines a robust AI-powered conversational agent that effectively bridges the gap between natural language and complex triathlon race data in Google BigQuery. By meticulously designing the LLM prompts, incorporating multi-turn conversational memory, and employing a RAG-like architecture for both SQL generation and result summarization, the system provides an intuitive and intelligent data exploration experience. The iterative improvements, structured approach, and comprehensive logging mechanisms highlight the power of combining modern LLMs with domain-specific knowledge bases and robust engineering practices to create highly functional, user-friendly, and maintainable AI applications.

**References**

[1] (Placeholder for Text-to-SQL research papers) [2] (Placeholder for LLM-based query generation papers) [3] (Placeholder for RAG framework papers)