Analytics Graph Query Solver (AGQS): Transforming Natural Language Queries into Actionable Insights

Debojyoti Saha[1,*], **Krishna Singh**[2,*], **Moushumi Mahato**[3], **Javaid Nabi**[4]
Language AI Appliances & Analytics
Samsung R&D Institute, Bengaluru, India
{[1]d.saha, [2]krish.singh, [3]moushumi.m, [4]javaid.nabi} @samsung.com

Abstract

In today's era, data analytics is crucial because it allows organizations to make informed decisions based on the analysis of large amounts of data. The evolving landscape of data analytics presents a growing challenge in effectively translating natural language queries into actionable insights. To address this challenge, we introduce a novel system that seamlessly integrates natural language processing (NLP), graph-based feature representation, and code generation. Our method, called Analytics Graph Query Solver (AGQS), utilizes large language models (LLMs) to construct a dynamic graph representing keywords and engineered features. AGQS transforms textual input queries into structured descriptions and generates corresponding plans. These plans are executed stepwise to create a unified code, which is subsequently applied to our in-house virtual assistant dataset to fulfill the user's query. Furthermore, a robust verification module ensures the reliability of the obtained results. Through experimentation, our system achieved an accuracy of 62.2%, outperforming models like GPT-4 (50.2%), Graph Reader (56.6%), Mistral3 7B (38.6%), and Llama 7B (37.6%). Overall, our approach highlights the importance of feature generation in textual query resolution and demonstrates notable improvements in accessibility and precision for data analytics. With this method, we aim to present a solution for converting natural language queries into actionable steps, ultimately generating code that provides data insights. This approach can be utilized across different datasets, empowering developers and researchers to gain valuable insights effortlessly.

1 Introduction

In the rapidly advancing field of data-driven analytics, the ability to extract meaningful insights from large datasets is crucial for decision-making. Researchers, developers, and non-technical users often require quick access to information or analytics results, but the formulation of technical code to get data insights can be a complex task. A better approach would be where users may express their needs using technical or diverse natural language forms to get data insights, some using technical terms like "churn prediction", while others may phrase their queries in more general, non-technical terms like "customers who are likely to leave next month". This makes it easier for users as they don't need to think about writing codes to extract data insights. This kind of system is more user-friendly and results in a significant number of incoming queries, for example, Figure 1 shows count of technical and non-technical queries for our smartphone based in-house virtual assistant data. The convenience of asking queries in a natural language allows users to easily gain insights from the data. However, to understand and address these queries is a different task altogether and a solution is required that is capable of bridging the gap between technical terminology or natural language and an automated plan for query execution, along with ensuring accuracy and relevance in the analysis process.

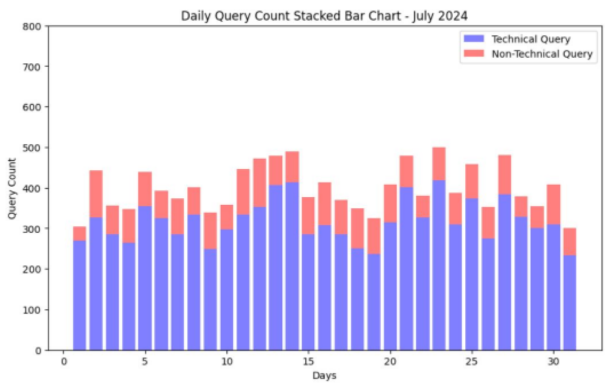

Figure 1: Technical and Non-technical Query Count in the month of July 2024

*Equal contribution

This research presents a novel system called Analytics Graph Query Solver (AGQS) designed to address these challenges by transforming natural language queries into executable, context-aware solutions. **The system takes input queries in any textual form and reformulates them into a structured and meaningful description, followed by the generation of a plan to execute the required analysis. The core of this system is a graph-based structure that represents keywords and engineered features, which are essential for extracting actionable insights from datasets. By leveraging this graph, the system can map user queries to relevant features and relationships, enabling the generation of step-wise plan to further produce a unified code that is applied to the dataset to fulfill the user's query.**

To construct the graph, we gather relevant research from the field of data analysis and also utilize large language models (LLMs) to expand this selection. We derive engineered features (functions or operations) from these studies, and extract detailed descriptions of the features using LLMs. From these descriptions, we extract relevant keywords. This collection of engineered features and keywords forms the nodes, while their connections form the edges of the graph. The resulting graph serves as a fundamental tool for exploring the relationships between keywords and features, playing a crucial role in identifying an effective solution for the user's query.

To ensure that the generated results are accurate, the system employs a verification module that operates at multiple stages of the query resolution process. This module checks the relevance of selected features, removes redundancies, and validates the final code. **By combining the capabilities of graph exploration, LLM-based feature generation, and a robust verification process, our system not only answers technical and non-technical queries but also ensures that the insights provided are both accurate and actionable.**

This approach allows users from diverse backgrounds to interact with complex datasets through natural language, while the system intelligently handles the underlying complexity. This empowers users to focus on their core goals without needing to translate their questions into technical specifications, providing a seamless and powerful analytics experience.

Our contributions include:

- We introduce a novel method for building graphs based on analytical use cases, allowing flexibility for complex datasets and enhancing data representation.
- Our approach provides automated code generation that delivers solutions to queries and insights, streamlining the transition from analysis to actionable results.
- We combine advanced graph construction, intelligent query reformulation, and automated code generation into a cohesive solution, improving both the depth and accuracy of data analysis.

2 Related Work

Natural Language Querying and Analytics Systems: Natural language interfaces (NLIs) for databases (Popescu et al., 2003) have been explored extensively. (Li and Jagadish, 2014) developed the NaLIR system, which enabled users to write natural language queries that are transformed into SQL. Similarly, DBPal (Basik et al., 2018) sought to improve natural language querying by leveraging deep learning models to parse and convert natural language into structured database queries. While these approaches mark significant advancements in allowing non-technical users to interact with databases, they often struggle with ambiguity in natural language and fail to extend beyond predefined technical vocabularies. Also, these systems primarily focus on database interactions and do not extend to tasks like feature engineering or code generation based on dynamic queries.

The challenge of handling diverse natural language inputs is further complicated when queries involve complex analytical tasks. Research (Tur and De Mori, 2011) in speech-to-text analytics systems and subsequent developments in dialogue systems have highlighted the difficulty of ensuring query intent is accurately captured and translated into actionable steps. Another work (Liu et al., 2020) has shown that while NLP techniques such as semantic parsing can improve the flexibility of these systems, there remains a gap in generating accurate and contextually relevant outputs in response to natural language queries that mix technical and non-technical terminology.

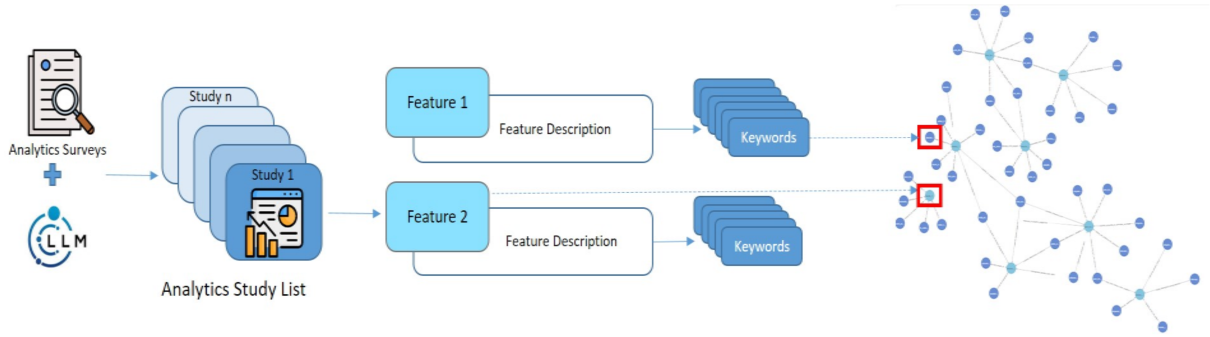

Figure 2: Graph Construction Mechanism

Graph-Based Representations and Feature Engineering: Graph-based approaches have emerged as powerful tools for capturing relationships between entities in various fields, including information retrieval and knowledge discovery. The use of graphs to represent structured information has been explored extensively in knowledge graphs (KGs) (Ji et al., 2021), where entities and their relationships are stored in a graph format to enhance data retrieval and exploration. KGs, such as those developed by Google and Microsoft, have demonstrated the effectiveness of this approach for large-scale applications. However, extending these principles to more focused, domain-specific analytics tasks presents unique challenges, especially when it comes to feature engineering. AutoKG (Chen and Bertozzi, 2023) automates the construction of domain-specific knowledge graphs, allowing dynamic query handling, but it does not extend to generating features or unified code execution for analytics as in our system.

Feature engineering, a critical component of data analytics, has traditionally relied on manual processes or predefined dataset-specific algorithms. Work by (Zhang et al., 2018) explored automated feature engineering through tools like FeatureTools (Alteryx, 2018), which generate features based on relational data. While promising, these tools still require significant manual setup and lack integration with natural language inputs. Other approaches, such as the use of automated machine learning (AutoML) (He et al., 2021), have sought to streamline feature engineering, but they often produce features in isolation without considering how features interact or relate in the context of a broader task. AutoAI (Tushir, 2019) automates feature engineering and model selection but is not designed for natural language inputs. They require structured data and predefined workflows, which can dynamically generate features from natural language queries. AutoFE (Li et al., 2023) automates feature engineering with machine learning techniques but lacks integration with natural language processing for dynamic query reformulation.

LLM-Based Code Generation: OpenAI Codex (Finnie-Ansley et al., 2022) and GitHub Copilot (Wermelinger, 2023) translate natural language instructions into code, simplifying technical tasks for developers. These tools generate code but are not designed to reformulate user queries, extract or generate features, and explore relationships like graph-based systems. AlphaCode (Li et al., 2022) generates competitive-level coding solutions based on user input but lacks the integration of feature engineering or analytics-related tasks.

Query Reformulation: It is a well-studied area in the context of information retrieval (Manning, 2009), where queries are rewritten to improve search results. However, applying these principles to analytical queries, especially in data science and machine learning contexts, is a more recent development. Systems like AutoAI (Tushir, 2019) and H2O.ai (H2O.ai, 2022) have introduced automated workflows that guide users through model selection and hyperparameter tuning, but they still require structured inputs. Another research (Cao et al., 2021) suggests that reformulating natural language queries into a more structured plan that outlines each step of the analysis process can significantly improve the effectiveness of these systems.

3 Proposed Methodology

Our methodology focuses on constructing a graph that represents keywords and engineered features and then using this graph to formulate plan to address user queries. The process consists of several steps, including graph construction, query reformulation, graph exploration and unified code genera-

Engineered Feature	Feature Description	Keywords
average_session _duration	This engineered feature calculates the average duration of a **voice assistant session**. The average **session duration** is calculated by taking the difference between the local timestamps of the first and last utterances within a session, and then averaging these durations across all sessions. This metric provides insights into the **user's engagement** with the voice assistant, helping to understand how long users typically spend interacting with the assistant in a single session.	1. session duration 2. user engagement 3. voice assistant session

Table 1: Example of an Engineered Feature

tion. Additionally, a verification module is implemented at multiple stages to ensure the accuracy of the generated results. The entire process of *graph construction* is illustrated in Figure 2. The overall mechanism of query reformulation, graph exploration and unified code generation are illustrated in Figure 3.

The graph, denoted as $G = (V, E)$, where each node $v_i \in V$ represents a keyword or an engineered feature, and each edge $e_i \in E$ represents the relationship between keyword and engineered feature. The graph's construction which is explained in detail in Section 3.1, follows a systematic approach. Subsequent sections 3.2 through 3.4 outline the execution flow of this approach, showing how we use the graph throughout the process.

3.1 System Architecture

3.1.1 Graph Construction

The construction of the graph begins with the collection of studies from analytics surveys. An example study is "Churn Analysis" where we find users who are going to leave the service in near future. These studies are filtered based on their applicability and usefulness in the context of the data we are working with. Such as, study like "average order value analysis" for data of e-commerce will not be suitable for our smartphone-based in-house virtual assistant data.

To expand the scope of the study selection, we utilize a large language model (LLM) to suggest additional studies that might not have been covered in our manual filtering process. The LLM is provided with a prompt to generate a list of studies that align with our research domain, ensuring that the results are relevant and diverse.

Once the set of studies is finalized, we prompt the LLM to generate a set of engineered features for each study. An example of feature is shown in Table 1. These features are essentially python code which define specific operations or transformations that can be applied to the data to extract meaningful insights. Example codes are given in B.1 and B.2 in Appendix. The features would strictly use the data attributes available in our dataset. The descriptions of data attributes are provided to the LLM as context to create results tightly aligned with the dataset's structure and content.

The set of analytics studies, respective engineered features and their corresponding data attributes are stored in a DB as shown in Listing 1.

```
{
  "Study Name": "Feature Usage Analysis",

  "Engineered Features": {
    "user_device_preference": ["device_type", "
         hands_free_mode"],

    "time_of_day_usage": ["localHour", "
         app_launch_time"]
  }
}
```

Listing 1: An example of Analytics Study DB

Next, detailed descriptions of the engineered features are generated. These descriptions provide a deeper understanding of what each feature means and how it can be applied to the data. From these descriptions we extract salient keywords (Li et al., 2024) which gives an overview of the functionality of features as shown in Table 1 and Figure 2. We also generate relationship which gives rationale of how a particular keyword relates to the feature. These keywords serve as the building blocks of the graph.

Since many studies might have similar features and may end up having similar keywords, we normalise the set of keywords to get a unique set K, shown in eq. (1).

$$K = \{K_1, K_2, \cdots, K_n\}. \qquad (1)$$

This normalization process involve grouping synonymous or similar terms under a unified keyword to avoid redundancy. For example, ***'hands_free_mode'****:['hands-free', 'hands-free mode usage', 'hands-free mode',*

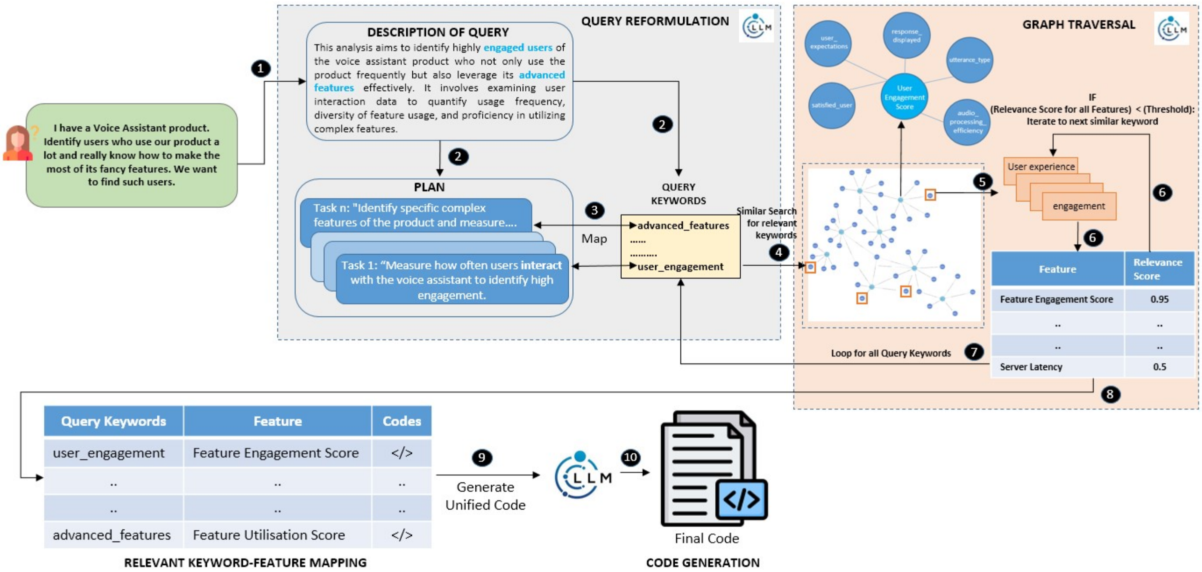

Figure 3: Mechanism showing Query Reformulation, Graph Exploration and Unified Code Generation

'hands_free_mode']

The graph G is then constructed, where nodes represent either features or normalized keywords. The final graph is expressed as shown in eq. (2).

$$G = \{F, K, R\} \tag{2}$$

where F represents the set of engineered features, K represents the set of normalized keywords, and R represents the relationships between them.

3.1.2 Query Reformulation and Plan Generation

Given a user query, our objective is to leverage the graph created to identify relevant features that can help provide a solution to the current user query. This is achieved through a single chain-of-thought prompt. The process begins by determining a suitable analysis name, then generating an elaborate description of the user query, capturing its nuances and context comprehensively. Moreover, essential query keywords(Li et al., 2024) are extracted from the user query. These are the key terms that summarize the main components of the query. We denote these query keywords as shown in eq. (3)

$$Qk = \{qk_1, qk_2, ..., qk_n\} \tag{3}$$

The output of above process is shown in Step 1 of Table 6.

The next step is to generate a plan that consists of subtasks to be followed to arrive at the query's solution. The plan is generated by LLM using the prompt available in Appendix A. Each query keyword is mapped to a specific step in the plan as shown in Step 2 of Table 6, ensuring that there is a clear connection between the keywords and the planned analysis.

To improve the accuracy of this step, we employ the "Verification Module" (detailed in Section 3.1.5). This eliminates redundant keywords and steps from the generated plan and also removes keywords which do not correspond to an actionable step. As shown in Step 3 of Table 6, the action plan of "voice assistant" is similar to "usage frequency", hence it is removed by the verification module.

3.1.3 Graph Exploration

- **Traversal**: For each query-keyword qk_i, we search for relevant keywords in our analytics-study based graph G as in eq. (2), denoted as $rk = \{rk_1, rk_2, rk_3..\}$, where $rk_i \in K$. The relevant keywords are selected based on a *relevance score*, which is calculated using a semantic similarity measure between the query keyword qk_i and each potential matching keyword rk_j as shown in Step 4 of Table 6. For each relevant keyword rk_j, we examine the neighbouring feature nodes in the graph as shown in Step 5 of Table 6. These feature nodes are evaluated for their relevance based on the relationship R defined in the graph. If the features meet a predefined relevance threshold, they are retained for further analysis. If no features cross the threshold for a particular relevant keyword, the system moves on to the next keyword in the set rk_{j+1} . This process is repeated for all query keywords in eq. (3).

- **Termination**: In order to terminate the graph exploration, we employ a Large Language Model (LLM) to verify if the initial keyword-plan mapping can be accomplished using the selected features alone. If the LLM confirms that the analysis can be completed with the chosen features, we proceed to assemble all the features to complete the analysis. If, however, the LLM determines that additional features are needed, the system prompts the LLM to generate new features for any remaining steps. This ensures that the query is comprehensively addressed, even if the initial graph exploration did not yield all the necessary features. This feature is added in the graph as well so that it can be utilized if required to solve a future user query.

- **Reusability**: If new feature is suggested by LLM to solve a query, the feature is added in the graph for reusability. To do so, we create description and keywords for this feature in the same way we did for all other features. In this way the graph is also dynamically updated with new features and related keywords.

3.1.4 Unified Code Generation and Execution

Using the enhanced context which constitutes the given problem statement, initial plan generated for the analysis and selected engineered features along with custom generated codes (if required), we generate a unified code (shown in Step 6 in Table 6) to complete the task. Example codes of features are given in B.1 and B.2 in Appendix along with unified code in B.3.

Before executing the code, we run it through a verification process to check for syntactical correctness and ensure that it aligns with the plan generated earlier. This is crucial for maintaining the coherence of the solution. The verification process is described further in the next section.

3.1.5 Verification Module

As it is widely recognised, LLMs are prone to hallucinations(Banerjee et al., 2024); the verification module plays a pivotal role in maintaining the accuracy and reliability of the results. It is designed to perform checks at various points in the flow to ensure the output is both efficient and actionable.

- **Code-Based Verification**: In the graph construction phase (Section 3.1), two critical steps are the *normalization of keywords* and the *creation of keyword-feature relationships*. However, we observe the LLM sometimes deviates from the given instructions and output format requirement. To tackle these issues, the verification module actively identifies and corrects deviations in both keyword normalization and keyword-feature relationship creation. It ensures that keywords are standardized correctly, eliminates redundancies, and establishes relationships for all keywords, thereby preserving the sanctity of the data. To address this issue, it ensures removal of extra relationships generated for keywords that are not relevant to the resolution of the query.

- **LLM-Based Verification**: In the *query reformulation step*, it is noticed that the LLM sometimes produces redundant or non-actionable keyword-plan mappings. This can result in unnecessary exploration of the graph without discovering the required feature and in turn increasing latency. To address this issue, the LLM-based verification module eliminates such redundant or non-actionable keyword plan mappings, as shown in Table 2 In addition, it is essential to review the generated code for syntactical errors, so the LLM-based verification module is employed to check for such errors in all the codes generated.

- **SME Verification**: After the query is transformed into a step-by-step logical plan, a Subject Matter Expert (SME) reviews the plan to ensure it aligns with the intended purpose of the user query. If any errors are identified in the sequence of actions, the SME provides feedback on the mistake. This feedback, along with the original user query and the generated plan, is then presented to the LLM as part of a self-reflection process. The LLM uses this feedback to revise the plan, which is subsequently applied to complete the user's task.

Engineered Feature	LLM generated Keyword-Feature-Relation tuple	Verification module revised tuple
name: 'user_engagement', 'key-words': ['user_interaction', 'audio_length', 'preferences', 'user_expectations']	('preferences', 'user_engagement','indicates'), ('audio_length','user_engagement','evaluates'), ('user_interaction','user_engagement','measures'), ('active_user','user_engagement','may indicate'), ('satisfied_user','user_engagement','may indicate'), ('interests','user_engagement','can help in understanding'), ('user_expectations','user_engagement','can help in understanding')	('preferences', 'user_engagement', 'indicates'), ('audio_length', 'user_engagement', 'evaluates'), ('user_interaction', 'user_engagement', 'measures'), ('user_expectations', 'user_engagement', 'can help in understanding')

Table 2: Keyword-Feature-Relation Verification

4 Experiments and Results

4.1 Dataset

- **Data Attribute Preparation**: In the data pre-processing phase, initially we select the relevant data attributes from the raw dataset of smartphone-based in-house virtual assistant. Next, attributes containing null values are excluded as these will not impact meaningfully to any analysis. Furthermore, attributes with only a single unique value are removed, as they lack variability and will not contribute much to our features.

  Additionally, to enhance the interpretability and utility of all the data attributes, we employ Mistral3 (AI, 2023) to generate detailed descriptions of each attribute. These descriptions not only provide a clear understanding of the attributes but also facilitate in the accurate selection and coding of the features which use these attributes. This comprehensive approach to data pre-processing ensures that the generated code is based on high-quality, well-defined data, thereby enhancing the reliability and validity of the overall solution.

- **Experimental Setup**

Data	Overall	Technical	Non-Technical
Test	2000	1380	620
Dev	500	370	130

Table 3: Data sizes for experimental setup

Our approach is a zero-shot method, hence it does not require any training data. For validation, we begin with a dataset comprising 500 queries (technical and non-technical) from developers and researchers, which upon closer analysis, distilled into 50 unique analytical studies. Each of these studies represents a distinct analytical task requested by the developers. The test and validation data distribution is shown in Table 3. Examples of technical and non-technical queries is illustrated in Table 6. The classification of technical and non-technical query is beyond the scope of this paper.

To evaluate the effectiveness of the method, we compare the data summary identified by our approach against a set of true labels that were established for these 50 analytical studies.

By applying our proposed method, we generate predicted results for each query, which we then compare with the true output. This comparison enable us to calculate the accuracy of our approach across the 500 validation queries. We apply this method across all queries, providing a comprehensive measure of performance in terms of how accurately they identified the correct data insights requested by the user.

4.2 Results

Model	Overall	Technical	Non-technical
GPT 4 (Achiam et al., 2023)	50.2	52.43	48.85
Graph Reader (Li et al., 2024)	56.6	65.14	32.31
Mistral3 7B (AI, 2023)	38.6	41.62	30.0
Llama 7B (Touvron et al., 2023)	37.6	39.73	31.54
AGQS (Ours)	**62.2**	**69.19**	**42.31**

Table 4: Comparison of model accuracy

The results are derived by employing a zero-shot approach with various large language models

Type of Query	User Query	Expected Output	Actual Output	Output Overlap	Result
Non technical	I have a voice assistant product for smartphone. Identify users who use our product a lot and really know how to make the most of its fancy features. We want to find such users.	user_id list: [u1, u2, u4, u10,. . . ..u3476]	user_id list: [**u2**, **u4**, **u3476**, u5076]	81.3%	Pass
Technical	I have a voice assistant product for smart-phone. I want to find the potential churn users over a month.	user_id list: [u85, u1599, u5493, u7444,. . . ..,u10587]	user_id list: [u1489, u1521, **u1599**, u8565,. . . ..,**u10587**]	71.8%	Fail

Table 5: Example output of Technical and Non-technical queries

(LLMs) as shown in Table 4. These models are given the attributes from our in-house dataset, accompanied by their non-enhanced descriptions. A set of user query is provided to each model, requesting the generation of code for solving the given query. The resultant code is executed on the same dataset, and the results are compared with those obtained from our proposed method. The performance of the validation set is assessed, and the accuracy findings are tabulated in Table 4. An example of output on resolving technical and non-technical queries is shown in Table 5. As per the example queries, the output is a list of user ids. We find the maximum overlap of the actual output from expected output where if the overlap is beyond 75%, we mark the query resolution as'Pass', otherwise'Fail'.

GPT-4 (Achiam et al., 2023) outperforms other models in addressing non-technical queries due to its extensive training on large amount of dataset. GPT-4 (Achiam et al., 2023) achieves an accuracy level of 43.85% for non-technical queries and 52.43% for technical queries, resulting in an overall accuracy of 50.2% on the validation set. In contrast, the Graph Reader (Li et al., 2024), which utilizes a graphical representation of fields, performs better on technical queries, with an accuracy of 65.14%, but only achieves 32.31% accuracy for non-technical queries, resulting in an overall accuracy of 56.6% on the development set. Despite AGQS's reformulation of queries not being as effective as GPT-4 (Achiam et al., 2023), it is able to identify relevant features from the graph, which enhances the context and increases the accuracy of the results. The proposed method achieves an overall accuracy of 62.2%, with a 42.31% accuracy for non-technical queries.

Discussion: GPT-4 (Achiam et al., 2023), with its significantly larger number of parameters compared to other models, is capable of extracting more information from the large number attribute descriptions, enabling it to generate superior code. On the other hand, Graph Reader (Li et al., 2024) approaches the problem by first constructing a logical plan and then systematically linking the steps in this plan to the corresponding attributes in the graph. This method allows it to identify relevant attributes more effectively than other LLMs, which explains its better performance compared to the other techniques evaluated. However, our method outperforms others as it incorporates the relationship between engineered features and keywords, which provides a more specific and precise contribution to the plan, ultimately assisting in executing actions to fulfill the user's query.

5 Conclusion

Our work introduces a novel system that bridges the gap between natural language queries and complex data analytics through a unified framework comprising techniques of NLP, knowledge graph, and automated code generation. By constructing a dynamic graph of keywords and engineered features and leveraging large language models for query reformulation, plan generation and code creation, our system achieves a notable accuracy of 62.2%, surpassing existing models. AGQS demonstrates significant advancements in translating diverse user queries into actionable data insights, emphasizing the critical role of feature generation and keyword-feature relationships in effective step-wise planning and execution.

In future work, we plan to expand the system's scalability to handle larger datasets and more complex queries, improving natural language understanding to better manage ambiguous inputs, and developing more sophisticated feature generation methods. Additionally, practical improvements will be guided by user feedback, along with integrating emerging technologies that holds the potential to further enhance the system's capabilities.

Type of Query	User Query	Step 1 : Query Description	Step 2: Plan Generation and Mapping	Step 3: Plan Verification	Step 4: Initial Node Selection	Step 5: Feature Exploration	Step 6: Unify Code

Non technical	**Query 1**: I have a voice assistant product for smart-phone. Identify users who use our product a lot and really know how to make the most of its fancy features. We want to find such users.	"Analysis":
		{ **"analysis_name"**: "User Engagement and Feature Utilization Analysis",
		
		"description": "This analysis aims to identify highly engaged users of the voice assistant product who not only use the product frequently but also leverage its advanced features effectively. It involves examining user interaction data to quantify usage frequency, diversity of feature usage, and proficiency in utilizing complex features.",
		
		"keywords": ["user engagement", "feature utilization", "voice assistant", "advanced features", "user interaction data", "usage frequency"] }
			"keyword_plan_mapping":
			{
			"user engagement": "Measure how often users interact with ...",
			
			"feature utilization": "Analyze the variety of features used by individuals to gauge ...",
			
			"voice assistant": "Focus of the analysis; the product being examined ..",
			
			"advanced features": "Identify specific complex features of the product and measure ..",
			
			"user interaction data": "Collect and analyze data on how users interact with the voice assistant, ..",
			
			"usage frequency": "Quantify how often users engage with the voice assistant to identify ..." }
				"verified_ keyword_ plan_ mapping":
				{
				"user engagement": "Measure how often users interact with the voice assistant to identify high engagement.",
				
				"feature utilization": "Analyze the variety of features used by individuals to gauge ...",
				
				"advanced features": "Identify specific complex features of the product and measure ...",
				
				"user interaction data": "Collect and analyze data on how users interact with the voice assistant, ..."}
				
				"Rationale": "'voice assistant' and 'usage frequency' removed from plan. 'Voice assistant' is not an actionable step, making it redundant in the context of a plan. 'Usage frequency' is covered under 'user engagement' and 'user interaction data'."
					{
					"user engagement":
					
					{"keyword_name": "user_engagement", "relevance_score":"0.95"},
					
					{"keyword_name": "maintaining_user_interest", "relevance_score":"0.75"},
					
					{"keyword_name": "user_interaction", "relevance_score":"0.85",
					
					{"keyword_name": "user_experience", "relevance_score":"0.80",
					
					{"keyword_name": "user_behavior", "relevance_score":"0.70"},
					
					"feature utilization"
					[..
					]
					}
						"user engagement":
						{**"Rationale"**: " The "user_engagement_score" quantifies interaction level. The "feature_utilization_score" serves as a baseline for engagement, indicating how features contribute to overall engagement. The "session_engagement_score" measures session engagement. It is reasonable to explore the corresponding code chunks for more detailed insights.",
						
						"Score_list":
						[{"feature_name": "user_engagement_score", "relevance_score": "0.95"},
						
						{"feature_name": "feature_utilization_score", "relevance_score": "0.85"}],
						
						{"feature_name": "session_engagement_score", "relevance_score": "0.9"}],
						
						"selected_features":
						["user_engagement_score", "feature_utilization_score", session_engagement_score]
						}
							< feature 1 code>
							+
							..
							+
							<feature n code>
							=
							<unified code>

Technical	**Query 2**: I have a voice assistant product for smart-phone. I want to find the potential churn users over a month.	"Analysis": {**"analysis_name"**: "Churn Prediction Analysis", **"description"**: "This analysis aims to identify the users who are likely to discontinue using the voice assistant product over a month by analyzing their usage pattern.", **"keywords"**: ["churn", "users", "usage pattern", "smart-phone"] }	"keyword_plan_mapping": {**"churn"**: "Identify the users who are likely to discontinue using the voice assistant product", **"users"**: "Identify the users who are likely to discontinue using the voice assistant product", **"usage pattern"**: "Analyse the usage pattern data of the users to predict churn", **"smart-phone"**: "Voice assistant product for smart-phone"	"verified_keyword_plan _mapping": {**"churn"**: "Identify the users who are likely to discontinue using the voice assistant product over a month", **"usage pattern"**: "Analyse the usage pattern data of the users to predict churn"}	{**"churn"**: [{"keyword_name": "domain_abort_rate", "relevance_score": "0.8"}, {"keyword_name": "user_satisfaction", "relevance_score": "0.7"}, {"keyword_name": "user_engagement_level", "relevance_score": "0.6"}], **"usage_pattern"**: [{"keyword_name": "user_satisfaction", "relevance_score": "0.9"}, {"keyword_name": "user_preferences", "relevance_score": "0.8"}, {"keyword_name": "user_information_needs", "relevance_score": "0.7"}]	{**"churn"**: {**"Rationale"**: "The 'domain_execution_abort_time _by_reason' feature indicates the time taken for the domain execution to be aborted due to various reasons, which could be a potential indicator of churn as users may abandon the domain due to poor performance or other issues.", **"Score_list"**: [{"feature_name": "domain_execution_abort_time _by_reason", "relevance_score":"0.8"}]	< feature 1 code> + .. + <feature n code> = <unified code>

Table 6: An example table for both types of queries and their processing through various modules

Limitations

While our approach presents significant advancements, it does have a few limitations. The system may encounter challenges with highly ambiguous or context-dependent queries, potentially affecting the accuracy of reformulations and feature selections. Additionally, scalability issues could arise as datasets and graphs grow larger and more complex, potentially impacting performance. To address these limitations, continuous enhancements of the system is required to ensure its robustness and effectiveness across diverse applications.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Mistral AI. 2023. Mistral-7B-Instruct-v0.3. https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.3. [Online; accessed 01-Jan-2023].

Alteryx. 2018. Featuretools is an open source framework for automated feature engineering. https://www.featuretools.com/. [Online; accessed 26-Jan-2018].

Sourav Banerjee, Ayushi Agarwal, and Saloni Singla. 2024. Llms will always hallucinate, and we need to live with this. *arXiv preprint arXiv:2409.05746*.

Fuat Basik, Benjamin Hättasch, Amir Ilkhechi, Arif Usta, Shekar Ramaswamy, Prasetya Utama, Nathaniel Weir, Carsten Binnig, and Ugur Cetintemel. 2018. Dbpal: A learned nl-interface for databases. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1765–1768.

Kaibo Cao, Chunyang Chen, Sebastian Baltes, Christoph Treude, and Xiang Chen. 2021. Automated query reformulation for efficient search based on query logs from stack overflow. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1273–1285. IEEE.

Bohan Chen and Andrea L Bertozzi. 2023. Autokg: Efficient automated knowledge graph generation for language models. In *2023 IEEE International Conference on Big Data (BigData)*, pages 3117–3126. IEEE.

James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*, pages 10–19.

H2O.ai. 2022. operate and innovate with AI . https://h2o.ai/. [Online; accessed 01-Jan-2022].

Xin He, Kaiyong Zhao, and Xiaowen Chu. 2021. Automl: A survey of the state-of-the-art. *Knowledge-based systems*, 212:106622.

Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and S Yu Philip. 2021. A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE transactions on neural networks and learning systems*, 33(2):494–514.

Fei Li and Hosagrahar V Jagadish. 2014. Nalir: an interactive natural language interface for querying relational databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 709–712.

Liyao Li, Haobo Wang, Liangyu Zha, Qingyi Huang, Sai Wu, Gang Chen, and Junbo Zhao. 2023. Learning a data-driven policy network for pre-training automated feature engineering. In *The Eleventh International Conference on Learning Representations*.

Shilong Li, Yancheng He, Hangyu Guo, Xingyuan Bu, Ge Bai, Jie Liu, Jiaheng Liu, Xingwei Qu, Yangguang Li, Wanli Ouyang, et al. 2024. Graphreader: Building graph-based agent to enhance long-context abilities of large language models. *arXiv preprint arXiv:2406.14550*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Qian Liu, Bei Chen, Jiaqi Guo, Jian-Guang Lou, Bin Zhou, and Dongmei Zhang. 2020. How far are we from effective context modeling? an exploratory study on semantic parsing in context. *arXiv preprint arXiv:2002.00652*.

Christopher D Manning. 2009. *An introduction to information retrieval*.

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th international conference on Intelligent user interfaces*, pages 149–157.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Gokhan Tur and Renato De Mori. 2011. *Spoken language understanding: Systems for extracting semantic information from speech*. John Wiley & Sons.

Kartikay Tushir. 2019. Autoai using ibm watson.

Michel Wermelinger. 2023. Using github copilot to solve simple programming problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 172–178.

Zhenli Zhang, Xiangyu Zhang, Chao Peng, Xiangyang Xue, and Jian Sun. 2018. Exfuse: Enhancing feature fusion for semantic segmentation. In *Proceedings of the European conference on computer vision (ECCV)*, pages 269–284.

A Prompts and their output

Query Description and Analysis determination prompt (Step 1)

As an Intelligent assistant, your objective is to, given a problem statement for a voice assistant product, You have to understand the type of analysis required to address the issue. Describe the issue in technical term and identify the important keywords.
***Output format:

```
{
    "analysis_name": <analysis_name>,
    "description": <description>,
    "keywords": [list],
}
```

Now, generate the analysis details for the given problem statement:
user_requirement: I have a voice assistant product for smartphone. Identify users who use our product a lot and really know how to make the most of its fancy features. We want to find such users.
The output should be strictly in JSON format.

Query Description and Analysis determination Output (Step 1)


```
{
    "Problem_statement": "I have a voice assistant product for smartphone.
        Identify users who use our product a lot and really know how to make
        the most of its fancy features. We want to find such users.",
    "Analysis": {
        "analysis_name": "User Engagement and Feature Utilization Analysis",
        "description": "This analysis aims to identify highly engaged users of
             the voice assistant product who not only use the product
            frequently but also leverage its advanced features effectively. It
             involves examining user interaction data to quantify usage
            frequency, diversity of feature usage, and proficiency in
            utilizing complex features.",
        "keywords": ["user engagement", "feature utilization", "voice
            assistant", "advanced features", "user interaction data", "usage
            frequency"]
    },
}
```

Plan Generation and Mapping prompt (Step 2)

As an Intelligent assistant, given the problem statement and analysis details, your objective is to define a plan. You should associate every given keyword to a part of the plan to solve the problem statement.
Strategy: ****
1. You can choose to modify the keywords if you feel it fits the plan more appropriately. ****

***Output format:

```
{
    "Problem_statement": <problem statement>
    "Analysis": <analysis details>,
    "keyword_plan_mapping": <mapping>
}
```

Now map every keyword to a part of the plan to solve the given problem statement:
Problem_statement: I have a voice assistant product for smartphone. Identify users who use our product a lot and really know how to make the most of its fancy features. We want to find such users.

```
"Analysis": {
        "analysis_name": "User Engagement and Feature Utilization Analysis",
        "description": "This analysis aims to identify highly engaged users of
             the voice assistant product who not only use the product
            frequently but also leverage its advanced features effectively. It
             involves examining user interaction data to quantify usage
            frequency, diversity of feature usage, and proficiency in
            utilizing complex features.",
        "keywords": ["user engagement", "feature utilization", "voice
            assistant", "advanced features", "user interaction data", "usage
            frequency"]
    }
```

The output should be strictly in JSON format.

Plan Generation and Mapping Output (Step 2)

```
{
    "Problem_statement": "I have a voice assistant product for smartphone.
        Identify users who use our product a lot and really know how to make
        the most of its fancy features. We want to find such users.",
    "Analysis": <same as previous step>,
    "keyword_plan_mapping": {
        "user engagement": "Measure how often users interact with the voice
            assistant to identify high engagement.",
        "feature utilization": "Analyze the variety of features used by
            individuals to gauge their ability to leverage the product's
            capabilities.",
        "voice assistant": "Focus of the analysis; the product being examined
            for user engagement and feature utilization.",
        "advanced features": "Identify specific complex features of the
            product and measure how proficiently these are used by individuals
            .",
        "user interaction data": "Collect and analyze data on how users
            interact with the voice assistant, including command types,
            frequency, and feature usage.",
        "usage frequency": "Quantify how often users engage with the voice
            assistant to identify the most active users."
    }
}
```

Plan Verification prompt (Step 3)

As an Intelligent assistant, your objective is to verify a plan generated for a problem statement and remove any redundant step if any. Given the "Problem_statement" and "keyword_plan_mapping", evaluate the plan and give the verified plan “verified_keyword_plan_mapping".
Strategy:
****
1. Sub-task mapped to a keyword should be actionable in codified format. 2. Remove redundant keyword if other keywords already cover the subtask.
****

*** Output format:

```
{
    "Problem_statement": <problem statement>
    "Analysis": <analysis details>,
    "keyword_plan_mapping": <given plan>,
    "verified_keyword_plan_mapping": <modified plan>,
    "Rationale": <answer>
}
```

Now verify the below plan and give the output in specified format:
<Plan Generation and Mapping Output>
The output should be strictly in JSON format.

Plan Verification Output (Step 3)

```
{
    <same as previous step>,
    "verified_keyword_plan_mapping": {
        "user engagement": "Measure how often users interact with the voice
            assistant to identify high engagement.",
        "feature utilization": "Analyze the variety of features used by
            individuals to gauge their ability to leverage the product's
            capabilities.",
        "advanced features": "Identify specific complex features of the
            product and measure how proficiently these are used by individuals
            .",
        "user interaction data": "Collect and analyze data on how users
            interact with the voice assistant, including command types,
            frequency, and feature usage."
    },
    "Rationale": "The steps 'voice assistant' and 'usage frequency' were
        removed from the plan. 'Voice assistant' is not an actionable step but
         rather the focus of the analysis, making it redundant in the context
        of a plan. 'Usage frequency' is effectively covered under both 'user
        engagement' and 'user interaction data', making it redundant to list
        separately. The modified plan streamlines the approach by focusing on
        actionable steps without repetition."
}
```

Initial Node Selection prompt (Step 4)

As an Intelligent assistant given a problem statement for a voice assistant product, your objective is to provide solution by selecting analytics features from a graph. To achieve that, your current task is to identify relevant keywords from keyword nodes of graph with a given query keyword.
Strategy:
****
1) Results should be ordered by relevance-score. Relevance-Score should be strictly between 0 and 1. 2) Results should have at max 5 relevant keywords. 3) The output should be strictly in JSON format. 4) STRICTLY use the keywords from the given reference-keyword-list. Do not create new keywords apart from reference-keyword-list.
****

*** Output format:

```
{
    "query keyword": [{"keyword_name": "keyword1", "relevance_score": float(
        score1)},      {"keyword_name": "keyword2", "relevance_score": float(
        score1)},
          ...]
}
```

Your current task is to identify relevant keywords for below query-keyword:
"user engagement"

The **reference-keyword-list** to match from is given below:
<Graph_keyword_node_list>

The output should be strictly in JSON format.

Initial Node Selection Output (Step 4)


```
{
  "user engagement": [
    {"keyword_name": "user_engagement", "relevance_score": 0.95},
    {"keyword_name": "maintaining_user_interest", "relevance_score": 0.75},
    {"keyword_name": "user_interaction", "relevance_score": 0.85},
    {"keyword_name": "user_experience", "relevance_score": 0.80},
    {"keyword_name": "user_behavior", "relevance_score": 0.70}
  ]
}
```

Feature Exploration prompt (Step 5)

As an Intelligent assistant, your objective is to identify relevant features useful for a sub-task of the plan to achieve the Plan generated to address the Problem Statement. You should score the feature relevance using the relation and the sub-task. The relation offers, what insight can be gained by the feature towards the sub-task.
Your current task is to check a lists of features, with the objective of determining whether to proceed with reviewing the code chunk corresponding to these features.
Given the task you have the following Action Options:
****
1. Selected_feature(List[feature_name chosen]): Choose this action if you believe that the code chunk corresponding to the features may hold the necessary information to gain insight related to the sub-task. This will allow you to access more complete information and detailed code. 2. return_to_relevance_list: Choose this action if you ascertain that relevance_score for all features is less than 0.75. ****

Strategy:
****
1. relevance_score should be strictly between 0 and 1. 2. You can choose to select multiple features at the same time. 3. Relation only reflect part of the insight that can be gain by the code, so even if you feel that the features are slightly relevant to the keyword, please try to score more leniently and get more complete information. 4. Choose return_to_relevance_list if you ascertain that relevance_score for none of the features is more than 0.75. ****

*** Output format:

```
{
    'Rationale': <answer>,
    'Score_list' [{'feature_name': 'feature1', 'relevance_score': float(score1)},
                  {'feature_name': 'feature2', 'relevance_score': float(score2)},
                   ...],
    'Chosen Action': <action option chosen>
}
```

Now identify relevant features for below sub-task:
"Measure how often users interact with the voice assistant to identify high engagement."

The list of tuple (feature, relation) with keyword to match from is given below:
<List_of_tuples>

The output should be strictly in JSON format.

Feature Exploration Output (Step 5)


```
{
  "user engagement": {
    "Rationale": "All provided features are directly related to the keyword 'user engagement'. The '
        user_engagement_score' quantifies interaction level, which is a direct measure of user
        engagement. The 'feature_utilization_score' serves as a baseline for engagement, indicating
        how features contribute to overall engagement. The 'session_engagement_score' measures
        session engagement, providing insight into user engagement within specific sessions. Given
        the direct relevance of these features to the keyword, it is reasonable to explore the
        corresponding code chunks for more detailed insights.",
    "Score_list": [
      {
        "feature_name": "user_engagement_score",
        "relevance_score": 0.95
      },
      {
        "feature_name": "feature_utilization_score",
        "relevance_score": 0.85
      },
      {
        "feature_name": "session_engagement_score",
        "relevance_score": 0.9
      }
    ],
    "Chosen Action": "Selected_feature['user_engagement_score', 'feature_utilization_score', '
        session_engagement_score']"
  }
}
```

B Example of Feature Code and Unified Code

B.1 Example Feature Code 1 - User Engagement

```python
def user_engagement(total_utterances_in_conversation, utterance_type, responseDisplayed, responseSpoken):
    # Initialize the base score with the total number of utterances
    base_score = total_utterances_in_conversation

    # Define score multipliers for different utterance types
    utterance_type_multiplier = {
        'none': 0.5,  # Assuming 'none' has the least impact
        'root': 1.5,  # Root utterances have more impact
        'request_input': 1.0,  # Request input is standard engagement
        'next_step': 1.2,  # Next steps indicate continued engagement
    }

    # Check if utterance type is valid and apply multiplier
    if utterance_type in utterance_type_multiplier:
        base_score *= utterance_type_multiplier[utterance_type]
    else:
        raise ValueError("Invalid utterance type provided.")

    # Add bonus points for response types
    response_bonus = 0
    if responseSpoken:
        response_bonus += 2  # Assuming spoken responses add more value
    if responseDisplayed:
        response_bonus += 1  # Displayed responses also contribute but less than spoken

    # Calculate the final engagement score
    user_engagement_score = base_score + response_bonus

    return user_engagement_score
```

B.2 Example Feature Code 2 - Feature Utilization

```python
def feature_utilization(total_utterances_in_conversation, action_ids, domain_ids):
    """
    Calculate the feature utilization score based on the total number of utterances,
    the variety of action_ids, and the diversity of domain_ids used in a conversation.

    Parameters:
    - total_utterances_in_conversation: int, total number of utterances in the conversation.
    - action_ids: list, a list of action IDs used in the conversation.
    - domain_ids: list, a list of domain IDs interacted with in the conversation.

    Returns:
    - score: float, the calculated feature utilization score.
    """
    # Ensure action_ids and domain_ids are unique to count the diversity
    unique_action_ids = set(action_ids)
    unique_domain_ids = set(domain_ids)

    # Calculate the diversity scores
    action_diversity_score = len(unique_action_ids) / len(action_ids) if action_ids else 0
    domain_diversity_score = len(unique_domain_ids) / len(domain_ids) if domain_ids else 0
    # Combine the scores with total utterances to get the final score with the weights based on importance
    score = (0.4 * total_utterances_in_conversation) + (0.3 * action_diversity_score * 100) + (0.3 * domain_diversity_score * 100)

    return score
```

B.3 Unified Code - Python Code for User Scores Calculation

```python
# The CSV file is named 'user_interactions.csv' and is located in the same directory as this script
file_path = 'user_interactions.csv'

# Read the CSV file into a DataFrame
import pandas as pd
df = pd.read_csv(file_path)

# Placeholder for user_engagement function definition
def user_engagement(total_number_of_utterances_in_conversation, utterance_type, responseDisplayed, responseSpoken):
    # Function body as provided
    pass

# Placeholder for feature_utilization function definition
def feature_utilization(total_utterances, action_ids, domain_ids):
    # Function body as provided
    pass

def aggregate_user_scores(df):
    user_scores = {}
    for userid, group in df.groupby('userid'):
        total_utterances = len(group)
        engagement_score = user_engagement(total_utterances, group['utterance_type'].iloc[0], responseDisplayed, responseSpoken)
        feature_score = feature_utilization(total_utterances, action_ids, domain_ids)
        user_scores[userid] = {
            'user_engagement_score': engagement_score,
            'feature_utilization_score': feature_score
        }

    return user_scores

# Calculate scores
user_scores = aggregate_user_scores(df)

# Identify highly engaged and proficient users
thresholds = {
    'user_engagement_score': pd.Series([score['user_engagement_score'] for score in user_scores.values()]).quantile(0.95),
    'feature_utilization_score': pd.Series([score['feature_utilization_score'] for score in user_scores.values()]).quantile(0.95)
}
highly_engaged_users = [
    userid for userid, scores in user_scores.items()
    if scores['user_engagement_score'] >= thresholds['user_engagement_score']
    and scores['feature_utilization_score'] >= thresholds['feature_utilization_score']
]
```