Grounding Large Language Models with Knowledge Graphs for Superior Results

A DataWalk Proof of Concept

Executive Summary

Thanks to the versatile analytical capabilities of the hybrid graph-relational engine, the DataWalk platform proves to be an exceptional tool for supplementing LLMs. This allows the creation of LLM agents that can seamlessly solve complex analytical tasks by interacting with the system to dynamically traverse graphs, calculate pivot aggregations, perform searches, and much more. Moreover, DataWalk can also be used to feed the right data to the LLMs by using standard approaches like RAG and GraphRAG, enhancing the overall performance and reliability of the system. The system's capabilities in managing knowledge graphs make it an ideal platform for advanced AI applications, providing the necessary infrastructure to support complex workflows and data integrations.

Introduction

In today's rapidly evolving tech landscape, Large Language Models (LLMs) have emerged as powerful tools capable of delivering enormous benefits. However, they come with their own set of challenges and limitations. At DataWalk, our Innovation team has been exploring ways to combine LLMs with Knowledge Graph systems to address these challenges and enhance their capabilities. In this paper, we'll delve into DataWalk's recent research and development efforts, showcasing how this combination can be leveraged to create more effective and reliable LLM solutions. This paper will be interesting to those thinking about how knowledge graphs, genAI, and LLMs will support their future strategic initiatives, including chief data officers, data architects, and data scientists.

Understanding the Limitations of LLMs

LLMs are known for their general knowledge, generalizability, and in-context learning abilities. However, they also have notable drawbacks:

- **Hallucinations**: LLMs can generate inaccurate information.

- **Black Box Nature**: The decision-making process of LLMs is often opaque.
- **Lack of Domain Knowledge**: LLMs might not understand specific intricacies of internal documentation or domain-specific knowledge.
- **Inability to Incorporate New Knowledge**: LLMs are limited to the data they were trained on. Even with expensive and time-consuming fine-tuning, data always has a time lag.
- **Mathematical Reliability**: LLMs are not reliable for performing deterministic mathematical operations.

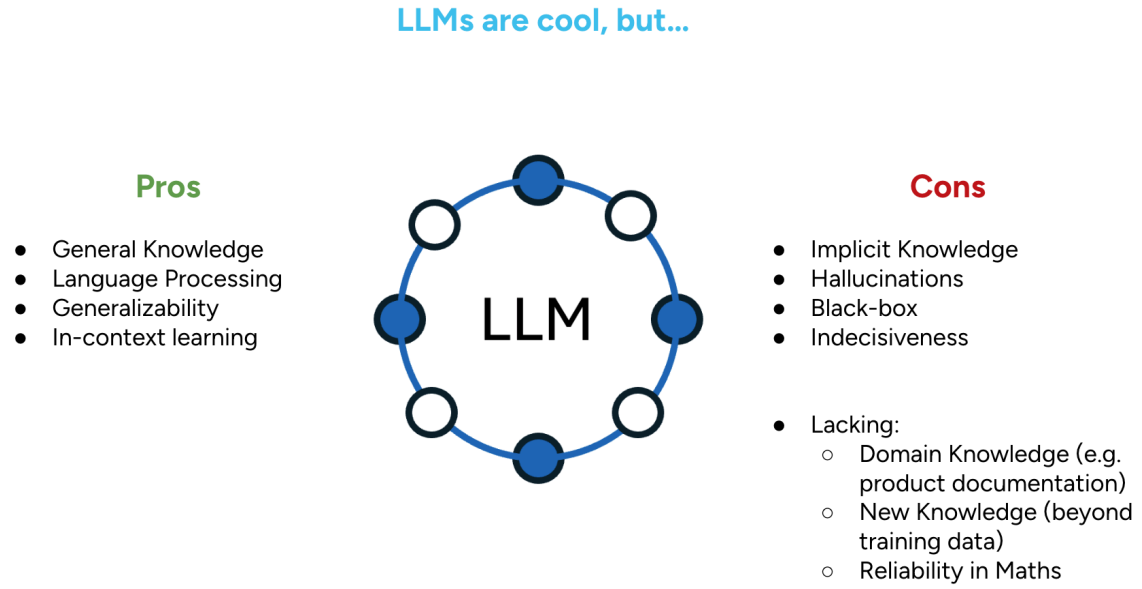
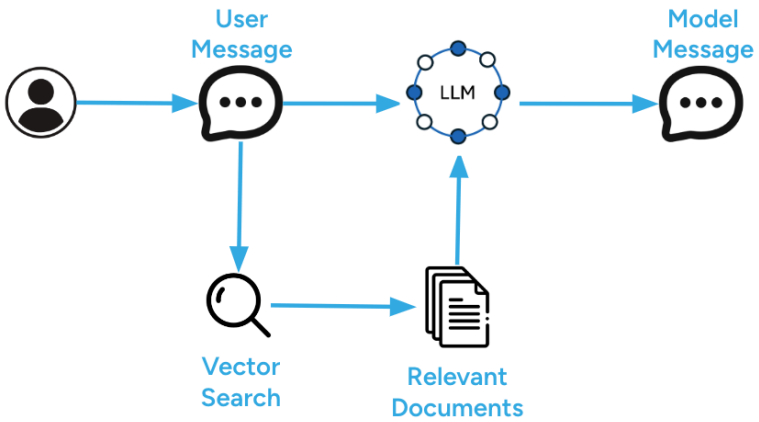


Figure 1: Pros and cons of LLMs

Retrieval Augmented Generation (RAG)



The popular Retrieval Augmented Generation (RAG) approach involves transforming documents into a vector database. When a user asks a question, it is converted into a vector and compared with the document vectors to find the most relevant ones. These documents are then used by the LLM to

generate a response. This method leverages LLM's summarization strengths and improves output quality by using private datasets while avoiding the high cost of fine-tuning.

While effective at incorporating recent data, RAG has limitations:

- Expertise Required: More expertise and management are needed compared to simple prompting.
- Semantic Understanding: Struggles with holistic semantic concepts and questions requiring traversal of specific structures like knowledge graphs.
- Multi-Hop Problem: Struggles with answering questions that require connecting the dots.
- Hallucinations: This does not solve the hallucination issue.

Enhancing LLMs with Knowledge Graphs

Another method of addressing LLMs' limitations is to combine them with Knowledge Graphs, building on the foundation set by RAG.

Graph Retrieval Augmented Generation (GraphRAG)

GraphRAG aims to address some of the limitations of RAG by incorporating graph-based methods. We explored two variations:

1. **GraphRAG with Unstructured Data**: This approach involves entity and relationship extraction to build a knowledge graph from text chunks. This graph allows for running graph algorithms like topic detection and community detection to add semantic meaning. This method improves over baseline RAG by providing a holistic view of the dataset and generating answers based on relationships within the data.

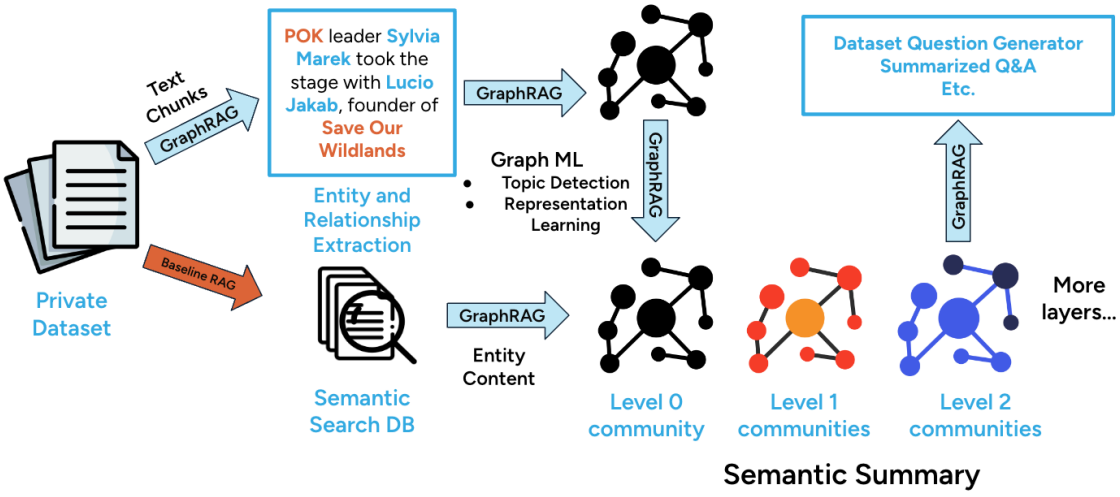


Figure 3: Workflow for creating a semantic summary from unstructured data

2. **GraphRAG with Structured Data**: Here, we leverage the existence of a knowledge graph system. This method is particularly useful for multi-hop problems, where questions can be represented as deterministic queries on the knowledge graph. By using predefined graph paths, this approach reduces computational complexity and lowers the chances of hallucination.

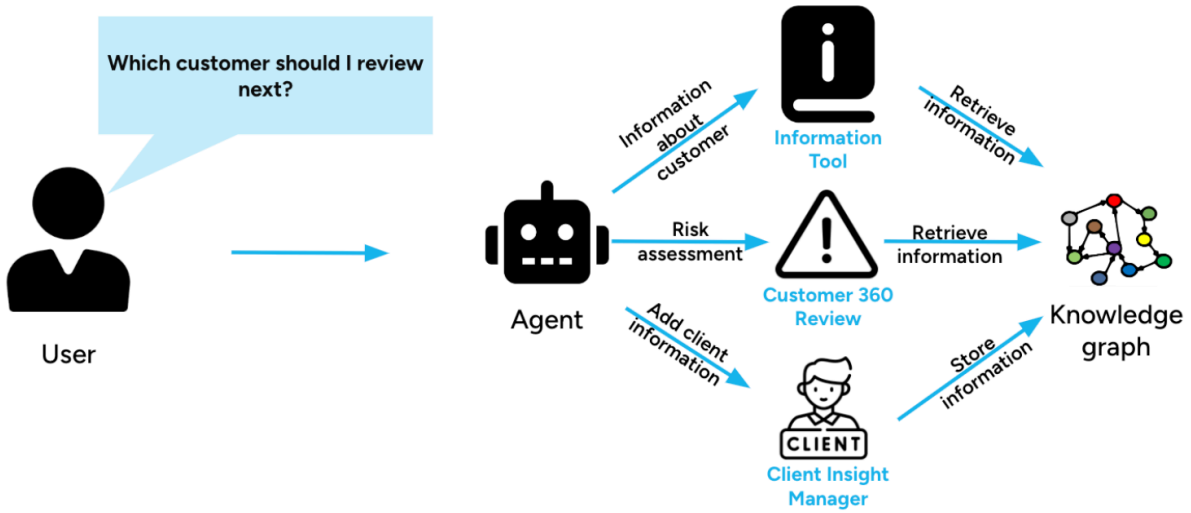


Figure 4: Workflow for creating a semantic summary from structured (knowledge graph) data

Introducing Reasoning Agents

Building on the GraphRAG approaches, we developed a reasoning agent to automate workflows using knowledge graphs. This agent represents a significant advancement in our research, as it provides a dynamic and intelligent way to solve and manage complex and spontaneous tasks that were previously difficult to manage with LLMs alone.

The Role of the Reasoning Agent

The reasoning agent is designed to generate high-level plans for complex and spontaneous tasks, iterate with users, and generate the necessary steps to execute these tasks. Using repeatable code greatly reduces hallucination and combined with the user feedback iterations, can nearly eliminate hallucinations. This process involves several key steps:

1. **High-Level Plan Generation**: Based on the user's query, the agent generates a high-level plan for the workflow based on the knowledge graph structure that outlines the steps needed to achieve the desired outcome.
2. **User Iteration and Feedback**: The generated plan is presented to the user for review. The user can provide feedback based on their domain knowledge and make corrections to ensure the plan is accurate and comprehensive.
3. **Code Generation**: Once the plan is approved, the agent generates the Python code required to execute the workflow. This code leverages the Graph Explorer and other APIs we have developed to interact with the knowledge graph.
4. **Execution and Review**: The generated code is executed, and the results are presented to the user. The user can review the results and verify their accuracy.

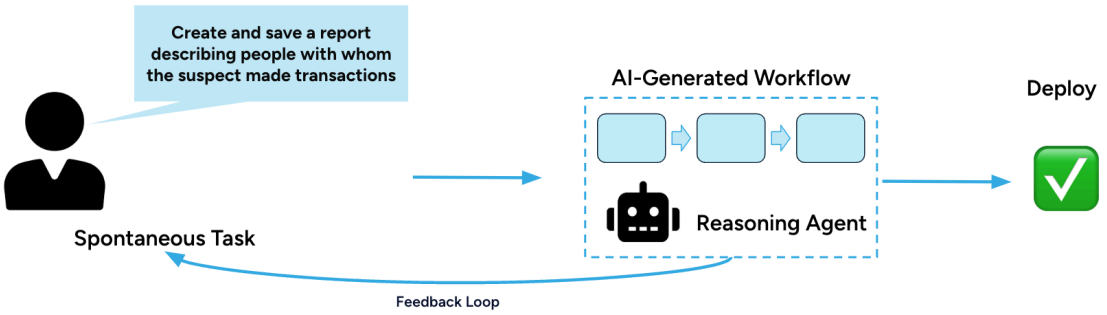


Figure 5: Overview of agent model with feedback

Reasoning Agents - Details of the process

Our approach is inspired by the FlowMind framework, which functions in two primary stages: an initial lecture to the LLM on the task context and available APIs, followed by workflow generation using the APIs and workflow execution to deliver the result to the user.

Stage 1: Lecture to LLM

The first stage involves a lecture[1] on the context, available APIs, and underlying knowledge graph ontology. We adhere to a generic lecture recipe to ensure the LLM has a clear understanding of the overall goal, the scope, inputs, and outputs of the functions in the APIs. The lecture recipe includes three key components:

1. **Context**: Introduce the domain of the expected tasks or queries. For example, in our experiments, we set up the context as receiving user information queries and creating high-level workflow steps and code.
2. **APIs**: Provide structured descriptions of the available APIs, including function names, input arguments, and output variables. These descriptions are semantically

[1] In this case, a lecture means to share appropriate background and context with the LLM to improve its ability to respond to a specific topic.

meaningful and relevant to the context to help the LLM make good use of the functions.

3. **Ontology**: Provide the knowledge graph's available classes, attributes, and relationships. The LLM must understand the ontology to create the appropriate queries.

Stage 2: Workflow Generation and Execution

In the second stage, the LLM generates the workflow using the provided APIs and executes it to deliver the results to the user. An optional feedback loop allows users to review and provide feedback, enabling the system to adjust the generated workflow accordingly.

This comprehensive approach is illustrated in the following diagram:

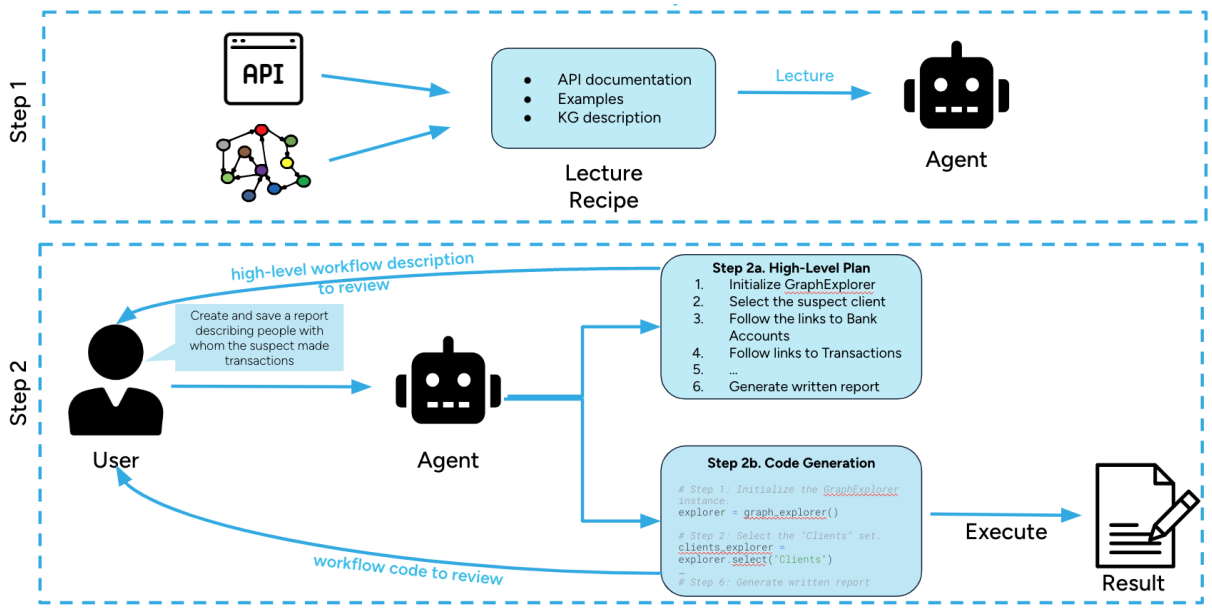


Figure 6: Agent model steps include lecture, recipe, workflow description, code execution, and result.

Implementation Details of the Reasoning Agent in DataWalk

The reasoning agent in DataWalk is built with several key components that enhance its usability and efficiency:

1. **Chaining API**: We implemented an intuitive chaining API called GraphExplorer that the LLM can learn how to use effectively. This API simplifies the process of chaining various tasks and operations, making it easier for the LLM to execute complex workflows seamlessly.
2. **Knowledge Graph Ontology Structure**: Based on the user's query, we provide the LLM with the structure of the knowledge graph's ontology, including sets, links, and

attributes that can be utilized. This detailed structure helps the LLM understand the relationships and context within the data, enabling it to generate more accurate and relevant results.

3. **Workflow Management and Reuse**: We developed a system to manage and reuse workflows generated in the past. This allows for the efficient execution of repetitive tasks and ensures consistency in the results. By storing and managing these workflows, users can quickly re-execute tasks without redefining the entire process, saving time and resources.

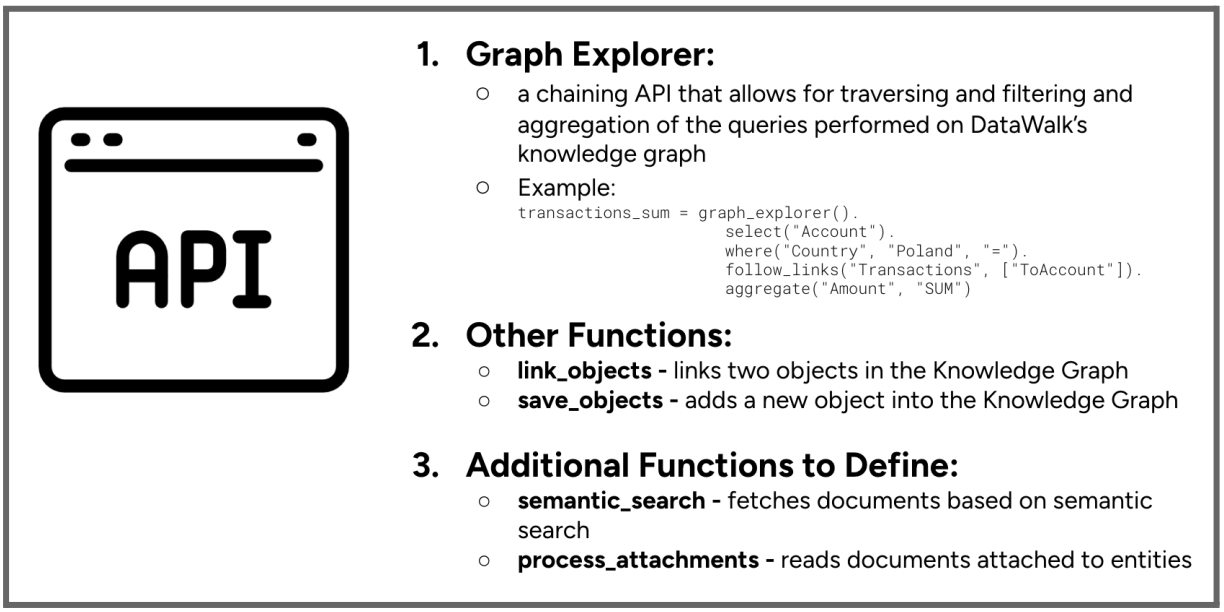


Figure 7: Key APIs from DataWalk's knowledge graph include Graph Explorer and others.

Practical Example

Consider a scenario where a banking investigator needs to prepare a report on suspicious transactions involving a client named Thomas Miller. The traditional approach would require the investigator to manually analyze transactions flagged as potentially fraudulent. With the reasoning agent, the process is streamlined:

1. **User Query**: The investigator asks the agent to create a report describing people with whom Thomas Miller made transactions, including the sum of inbound and outbound transactions. For our scenario, we asked the following question:

   Create a report describing the people with whom the suspect Thomas Miller made transactions. Calculate the sum of inbound and outbound transactions

made with the suspect for each associated person. Then create a written report and save it to the Reports set using the Content attribute.

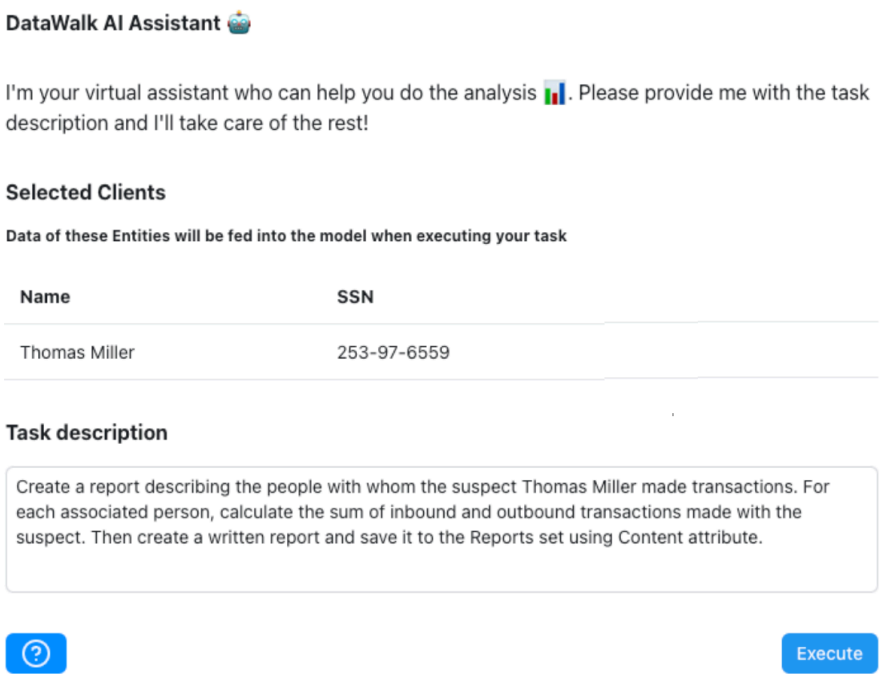
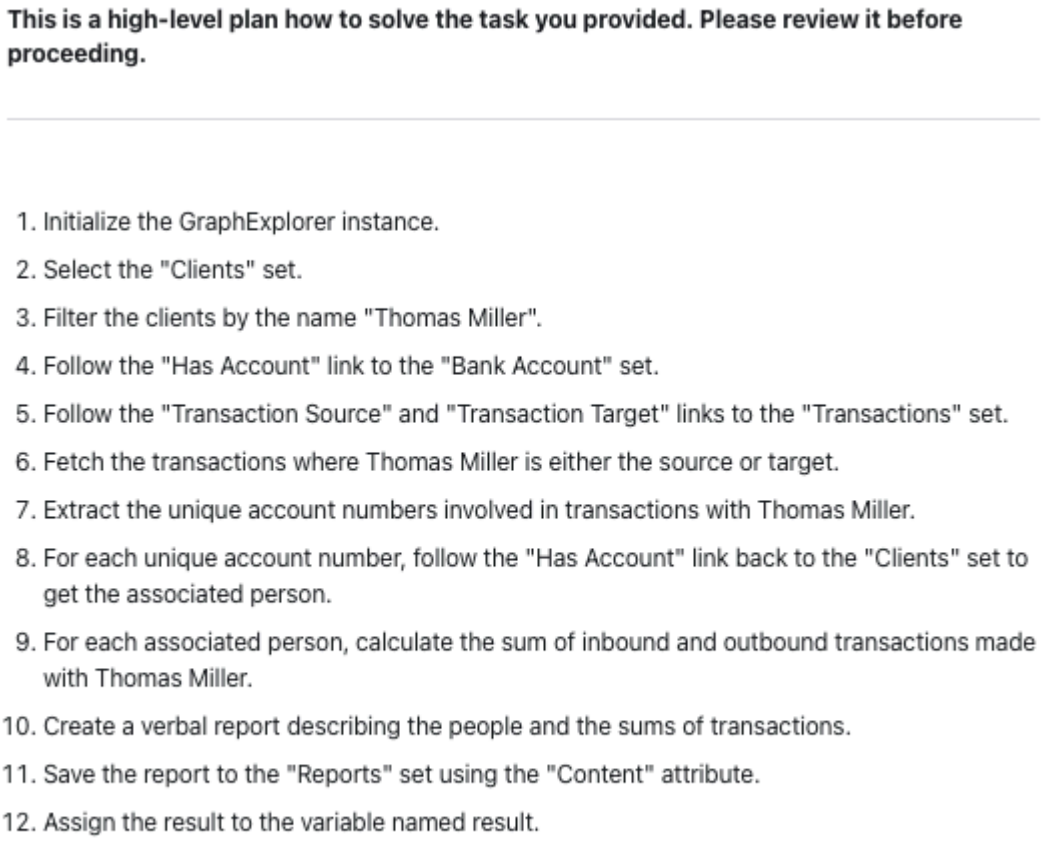


Figure 8: Example of prompt for reasoning agent model in DataWalk

2. **Plan Generation**: The agent generates a high-level plan consisting of 10 steps to gather the necessary data and perform the calculations.



This is a high-level plan how to solve the task you provided. Please review it before proceeding.

1. Initialize the GraphExplorer instance.
2. Select the "Clients" set.
3. Filter the clients by the name "Thomas Miller".
4. Follow the "Has Account" link to the "Bank Account" set.
5. Follow the "Transaction Source" and "Transaction Target" links to the "Transactions" set.
6. Fetch the transactions where Thomas Miller is either the source or target.
7. Extract the unique account numbers involved in transactions with Thomas Miller.
8. For each unique account number, follow the "Has Account" link back to the "Clients" set to get the associated person.
9. For each associated person, calculate the sum of inbound and outbound transactions made with Thomas Miller.
10. Create a verbal report describing the people and the sums of transactions.
11. Save the report to the "Reports" set using the "Content" attribute.
12. Assign the result to the variable named result.

Figure 9: Example of a high-level plan created by the LLM in the reasoning agent model

3. **Code Generation and Execution**: The agent generates and executes the corresponding Python code, automatically retrieving the relevant data from the knowledge graph.

```python
import json
# Step 1: Initialize the GraphExplorer instance.
explorer = graph_explorer()

# Step 2: Select the "Clients" set.
clients_explorer = explorer.select("Clients")

# Step 3: Filter the clients by the name "Thomas Miller".
thomas_miller_explorer = clients_explorer.where("Name", "Thomas Miller", "=")

# Step 4: Follow the "Has Account" link to the "Bank Account" set.
thomas_accounts_explorer = thomas_miller_explorer.follow_links("Bank Account", ["Has Account"])

# Step 5: Follow the "Transaction Source" and "Transaction Target" links to the "Transactions" set.
transactions_explorer = thomas_accounts_explorer.follow_links("Transactions", ["Transaction Source", "Transaction Target"])

# Step 6: Fetch the transactions where Thomas Miller is either the source or target.
transactions = transactions_explorer.fetch()

# Step 7: Extract the unique account numbers involved in transactions with Thomas Miller.
unique_account_numbers = set()
for transaction in transactions:
    unique_account_numbers.add(transaction["Source Account Number"])
    unique_account_numbers.add(transaction["Target Account Number"])

# Step 8: For each unique account number, follow the "Has Account" link back to the "Clients" set to get the associated person.
associated_people = []
for account_number in unique_account_numbers:
    if account_number != thomas_accounts_explorer.fetch()[0]["Account Number"]:
        person_explorer = explorer.select("Bank Account").where("Account Number", account_number, "=").follow_links("Clients", ["Has Account"])
        person = person_explorer.fetch()
        if person:
            associated_people.append(person[0])

# Step 9: For each associated person, calculate the sum of inbound and outbound transactions made with Thomas Miller.
report_data = []
for person in associated_people:
    person_account_explorer = explorer.select("Bank Account").where("Account Number", person["Account Number"], "=")
    inbound_sum = person_account_explorer.
        follow_links("Transactions", ["Transaction Target"]).
        where("Source Account Number", thomas_accounts_explorer.fetch()[0]["Account Number"], "=").
        aggregate("Amount", "SUM")
    outbound_sum = person_account_explorer.
        follow_links("Transactions", ["Transaction Source"]).
        where("Target Account Number", thomas_accounts_explorer.fetch()[0]["Account Number"], "=").
        aggregate("Amount", "SUM")
    report_data.append({
        "Name": person["Name"],
        "SSN": person["SSN"],
        "Inbound Transactions Sum": inbound_sum,
        "Outbound Transactions Sum": outbound_sum
    })

# Step 10: Create a verbal report describing the people and the sums of transactions.
report_content = "Report on transactions with Thomas Miller:\n\n"
for data in report_data:
    report_content += f"Person: {data['Name']} (SSN: {data['SSN']})\n"
    report_content += f"  Inbound Transactions Sum: {data['Inbound Transactions Sum']}\n"
    report_content += f"  Outbound Transactions Sum: {data['Outbound Transactions Sum']}\n\n"

# Step 11: Save the report to the "Reports" set using the "Content" attribute.
report_id = save_object("Reports", {"Content": report_content, "Name": "Transactions Report for Thomas Miller"})

# Step 12: Assign the result to the variable named result.
result = json.dumps({"report_id": report_id, "report_content": report_content})
```

Figure 10: Example of a code created by the LLM in the reasoning agent model

4. **Result Review**: The investigator reviews the output and summary of the results, which shows the transaction sums for each associated person.

Summary of the Result

The task execution resulted in the creation of a report that details the transactions made by Thomas Miller with other individuals. The report includes the names and Social Security Numbers (SSNs) of the associated people, along with the sums of inbound and outbound transactions they had with Thomas Miller.

Detailed Report

Person: Michael Johnson

- **SSN:** 520-42-3736
- **Inbound Transactions Sum:** $814.62
- **Outbound Transactions Sum:** $372.54

Person: Sarah Williams

- **SSN:** 434-11-5693
- **Inbound Transactions Sum:** $2845.32
- **Outbound Transactions Sum:** $0

Person: Noah Patel

- **SSN:** 009-09-9561
- **Inbound Transactions Sum:** $0
- **Outbound Transactions Sum:** $1452.98

Explanation

The report identifies three individuals who had financial transactions with Thomas Miller. For each person, the report provides:

1. **Name and SSN:** To uniquely identify the individual.
2. **Inbound Transactions Sum:** The total amount of money received from Thomas Miller.
3. **Outbound Transactions Sum:** The total amount of money sent to Thomas Miller.

This information can be useful for understanding the financial interactions and relationships Thomas Miller has with these individuals. The report has been saved in the system with the ID `13750008` for future reference.

Figure 11: Example of the results created by the LLM in the reasoning agent model

Benefits of the Reasoning Agent

- **Auditability**: The high-level plan and generated code provide transparency and explainability, allowing users to audit the process and verify the accuracy of the results.
- **Reusability**: The generated code can be reused for similar tasks in the future, enhancing efficiency and consistency.
- **Reduced Hallucinations**: The reasoning agent reduces the likelihood of hallucinations in the LLM's outputs by generating reusable deterministic plans and code. While this approach doesn't completely eliminate issues like hallucinations, it introduces valuable features that make LLMs more practical and reliable for real-world applications.

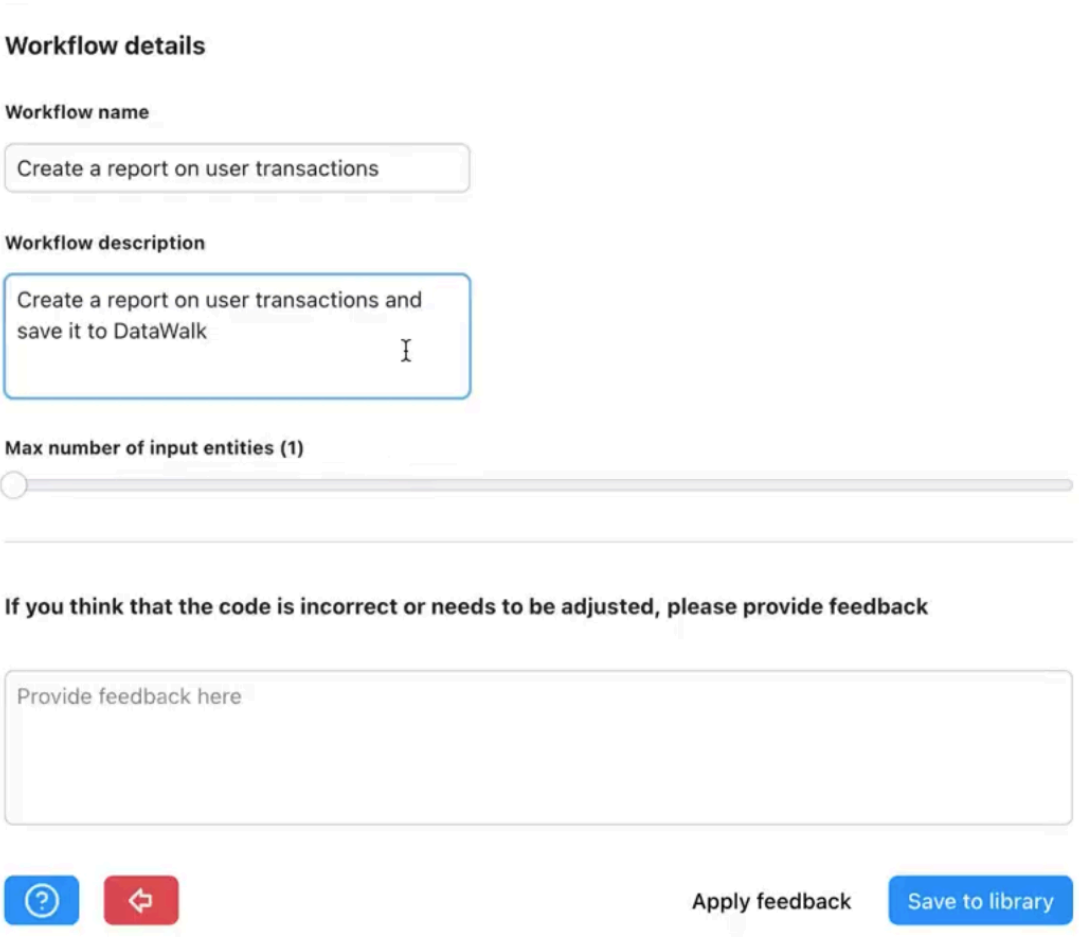


Figure 12: Example of saving a workflow created during the reasoning agent model

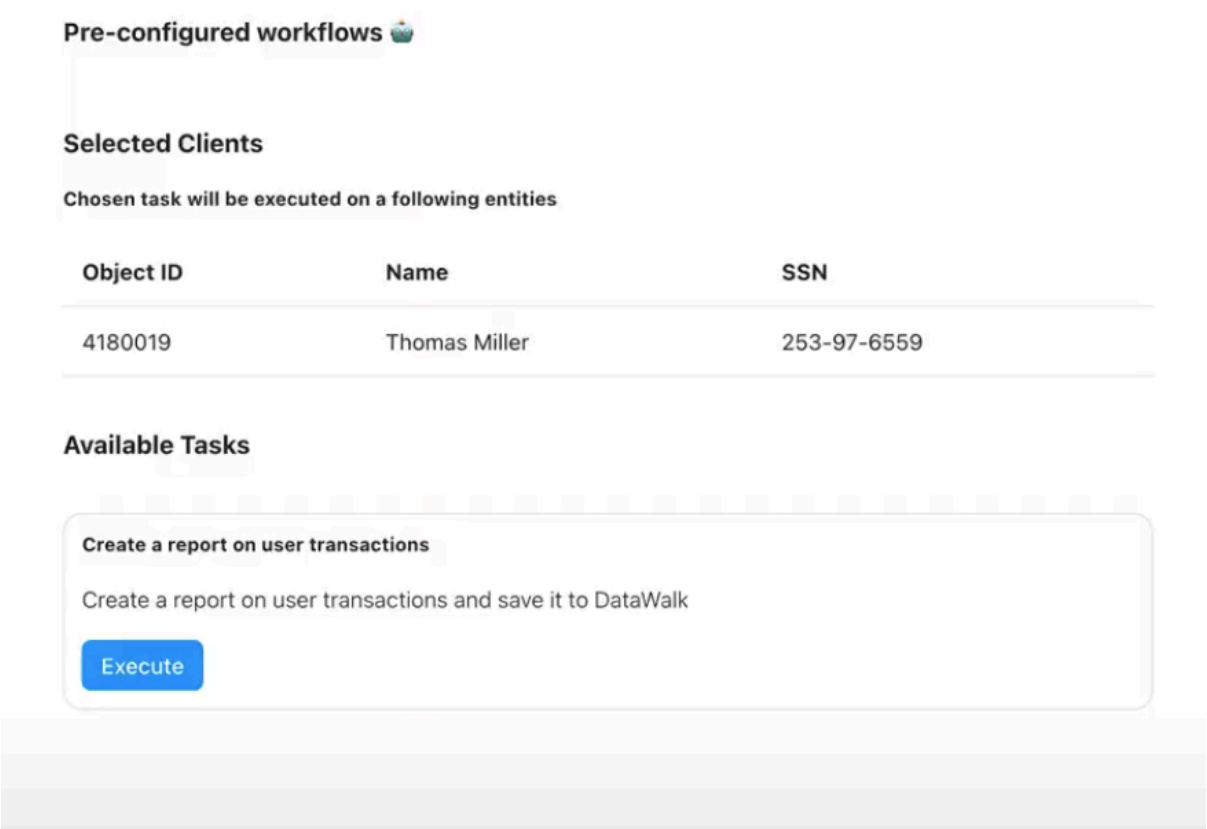


Figure 13: Example of reusing previously saved workflow via Pre-configured workflows. Any of the saved workflows can be executed under the Available Tasks tab.

Conclusion

Knowledge graphs offer a powerful solution to some of the inherent limitations of Large Language Models (LLMs) by providing rich relational and domain-specific context, thereby reducing hallucinations and improving accuracy. DataWalk enhances the reliability and broadens the applicability of LLMs through its ability to perform complex graph analytics on extensive data sets. The introduction of the reasoning agent marks a significant advancement, offering auditability, reusability, and the capability to dynamically handle complex queries. The synergy between DataWalk and LLMs promises to drive the development of increasingly sophisticated analytical solutions. We are enthusiastic about the potential of this technology and anticipate further innovations in this field.

About DataWalk

DataWalk is an enterprise-class software application and platform, for quickly integrating and analyzing vast amounts of data spread across your various data

sources. DataWalk enables both business and IT organizations to improve their efficiency, productivity, and competitiveness, while also meeting key technical and security requirements.

DataWalk meets a variety of needs across your organization by enabling you to fuse, clean, normalize, and connect your vast amounts of siloed data into a single source that can be used to feed AI/ML applications, Business Intelligence tools, and various other applications. DataWalk also enables you to perform sophisticated analyses of your data via complex no-code queries, graph algorithms, link analysis, machine learning, and more.

Additional Information

DataWalk's No Compromise Knowledge Graph
DataWalk Product Overview
DataWalk Technology

Copyright DataWalk 2024. DataWalk is a registered trademark of DataWalk S.A. Revision 0617. No statement in this document is intended to indicate future product development plans. Please contact your DataWalk representative for further information.