CineSphere - AI Powered Entertainment Guide

CINESPHERE

ULTIMATE MOVIE COMPANION

CineSphere caters to your entertainment needs by offering recommendations and a Q&A platform to address all your inquiries about world cinema.

The app leverages the following tools

- Neo4j
- Python
- OpenAI
- Langchain
- TMDB API

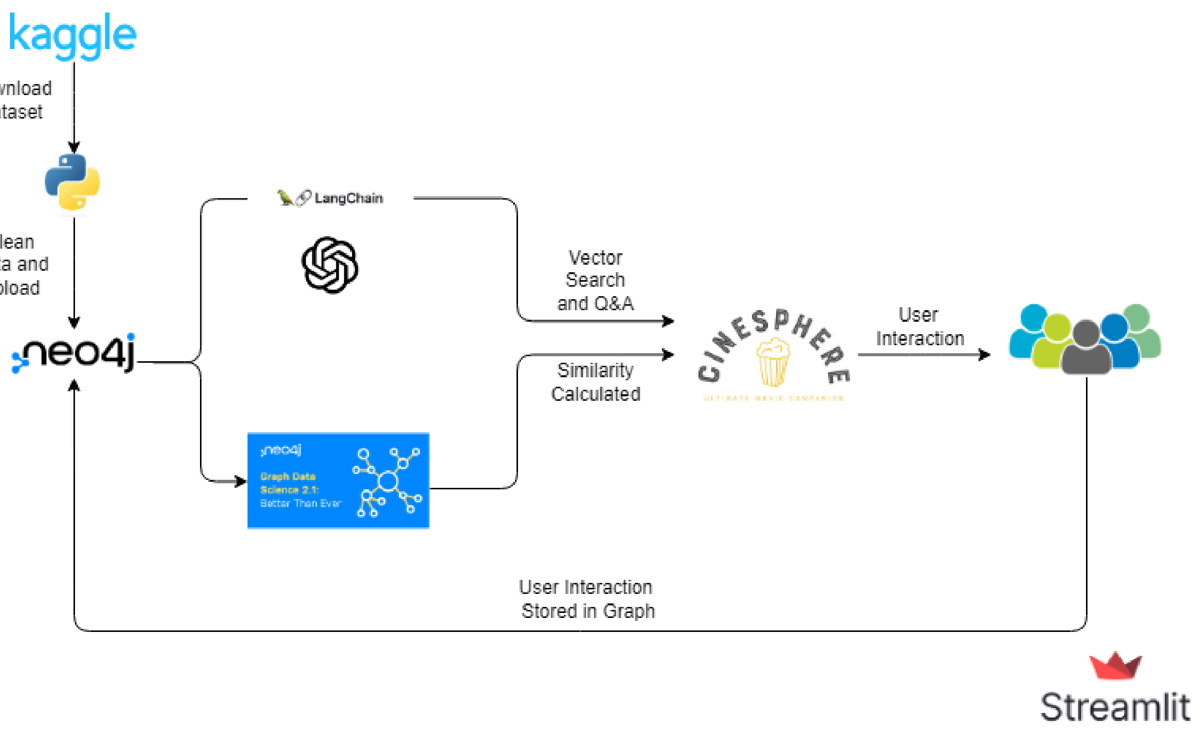
To reproduce - Create a conda/virtual environment with LangChain and Streamlit installed and do the following

1) Clone the repo

2) Create .env file with the following variables

```
NEO4J_URI = ''
NEO4J_USER = ''
NEO4J_PASSWORD = ''
OPENAI_API_KEY=''
OPENAI_BASE_URL=''
MOVIE_API_KEY=''
MOVIE_API_TOKEN=''
```

Flow of Application



About the Dataset

The dataset was obtained from Kaggle and consists of over 45000 movies. The dataset has been extracted from the Full MovieLens dataset and consists of movies released on or before July 2017. Data points include cast, crew, plot keywords, budget, revenue, posters, release dates, languages, production companies, countries, TMDB vote counts, and vote averages

Building The Graph

The Neo4j desktop version is publicly available on their website. Once the application has been installed ensure you have the following plugins installed

1. APOC – Awesome Procedure on Cypher
2. GDS – Graph Data Science
3. GenAI - Generative AI (For vector search)

```
CREATE CONSTRAINT unique_movie_id IF NOT EXISTS FOR (p:Movie) REQUIRE
(p.id) IS NODE KEY;
CREATE CONSTRAINT unique_person_id IF NOT EXISTS FOR (p:Person)
REQUIRE (p.person_id) IS NODE KEY;
CREATE CONSTRAINT unique_prod_id IF NOT EXISTS FOR
(p:ProductionCompany) REQUIRE (p.prod_comp_id) IS NODE KEY;
CREATE CONSTRAINT unique_genre_id IF NOT EXISTS FOR (p:Genre) REQUIRE
(p.genre_id) IS NODE KEY;
CREATE CONSTRAINT unique_lang_id IF NOT EXISTS FOR (p:SpokenLanguage)
REQUIRE (p.lang_id) IS NODE KEY;
CREATE CONSTRAINT unique_countries_id IF NOT EXISTS FOR (p:Country)
REQUIRE (p.country_id) IS NODE KEY;
CREATE CONSTRAINT unique_user_id IF NOT EXISTS FOR (p:User) REQUIRE
(p.userId) IS NODE KEY;
```

We want to create nodes for Movies, Persons (consisting of Crew and Cast), Users, Genres, Languages, and Countries. The initial step in building the graph is to start with creating constraints for each node that is present, define an exact schema for each node, and then start to populate data. Since we have over 100,000 data points and millions of relationships created, having constraints in place helps speed up the process.

```
LOAD CSV WITH HEADERS FROM $csvFile AS row
WITH row WHERE row.id IS NOT NULL LIMIT 100
MERGE (m:Movie {id: toInteger(row.id)})
ON CREATE SET m.name = row.original_title,
m.imdb_id = row.imdb_id,
m.popularity = toFloat(coalesce(row.popularity, 0.0)),
m.revenue = toInteger(coalesce(row.revenue, 0)),
m.spoken_languages = row.spoken_languages,
m.adult = toFloat(coalesce(row.adult, 0.0)),
m.budget = toFloat(coalesce(row.budget,0.0)),
m.overview = row.overview,
m.vote_average = toInteger(coalesce(row.vote_average, 0)),
m.vote_count = toInteger(coalesce(row.vote_count, 0));
```

Now another requirement is the APOC library. This library ensures that we can read each CSV file from the import folder and process them as batches

```
CALL apoc.periodic.iterate(
    'LOAD CSV WITH HEADERS FROM "file:///clean_ratings.csv" AS row
RETURN row',
    'MERGE (pc:User {userId: TOINTEGER(row.userId)})',
    { batchSize: 100}
) YIELD batches, total, errorMessages;
CALL apoc.periodic.iterate(
    'LOAD CSV WITH HEADERS FROM "file:///clean_ratings.csv" AS row
RETURN row',
    'MATCH (m:Movie {id: TOINTEGER(row.movieId)}) ' +
    'MATCH (pc:User {userId: TOINTEGER(row.userId)}) ' +
    'MERGE (pc)-[r:RATING { rating: TOFLOAT(row.rating) }]->(m) ',
    { batchSize: 100}
) YIELD batches, total, errorMessages;
```

Graph Data Science

GDS is the graph data science algorithm that facilitates the usage of data science algorithms like KNN, and GNN on top of the graph. We create projections from the main graph – sort of like a subset of the main graph, we can take the required nodes and relationships to analyze specific use cases.

```
CALL gds.graph.project('movies2',
              ['Movie','Genre','User','Person'],
              {RATING:{properties:'rating'},
              GENRE:{orientation:'REVERSE'},
              ACTED_IN:{orientation:'REVERSE'},
              CREWED_IN:{orientation:'REVERSE'}}
            )
```

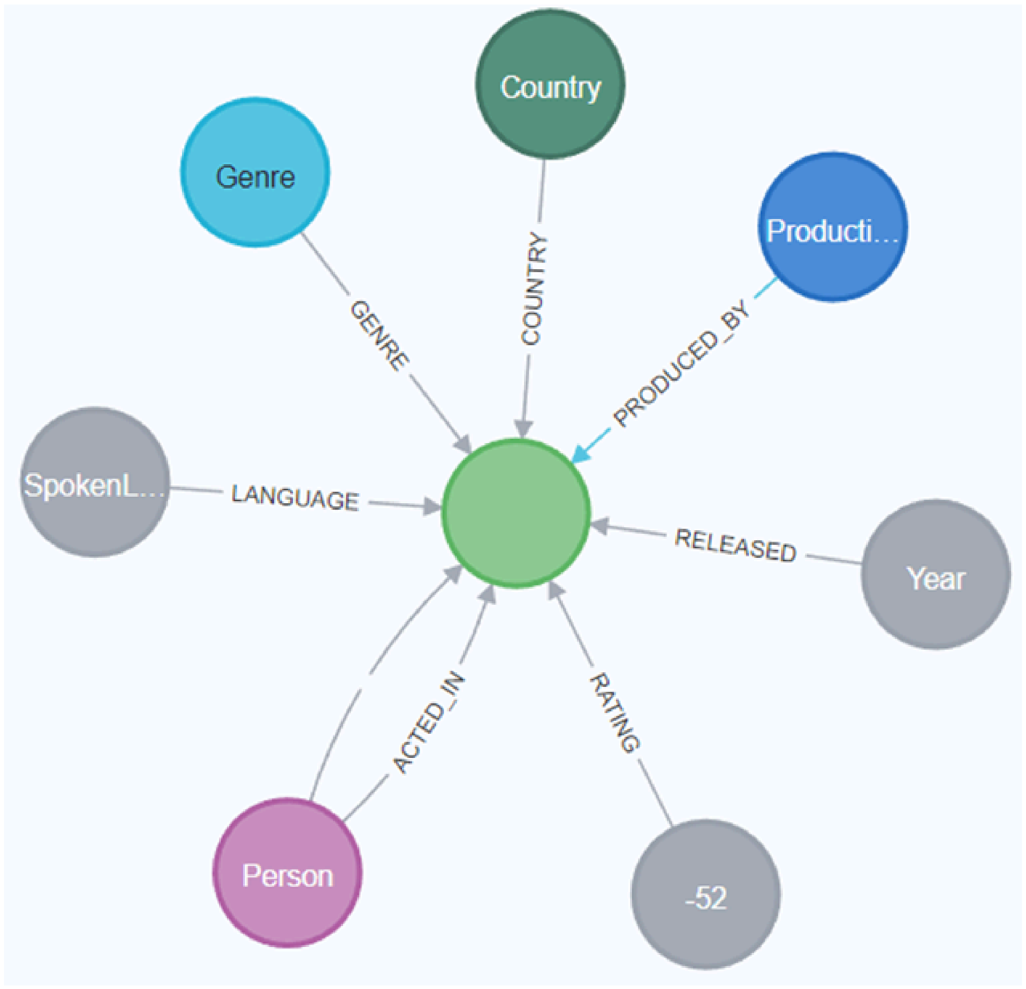
The query above creates a projection of the main graph by taking just the Movie, Genre, User, and Person Node with the required properties. Along with the nodes we can take the Relationships associated with the nodes along with the properties. The directions can be manipulated as well depending on use cases.

Finding Similarity

The two most common similarity algorithms are.

- KNN – K Nearest Neighbors
- Node Similarity based on Jaccard/Cosine/Overlap distance between Nodes

We must know that Neo4j is a graph database that leverages Node Similarity's importance based on relationships and shortest paths. The foundation of similarity between entities is the very application of Neo4j or other graph databases. Hence before going to more **complicated techniques to evaluate or extract insights from the data go through the first principles which are basic and standard.**



In this use case, basic Node Similarity can be used to find similarities between various nodes. We know from the above diagram that a Movie node has multiple other nodes surrounding it, these nodes strongly influence each other based on the movie node.

- A genre node surrounding the Movies will be influenced by the Movie nodes – Hence we can find similar genres based on the interconnection of Movies
- A Person, say an Actor is linked to a movie so similar actors can be found based on each person's interactions with Movies.
- The Node Similarity algorithm compares a set of nodes based on the nodes they are connected to. Two nodes are considered similar if they share many of the same neighbors.

```
CALL gds.nodeSimilarity.filtered.stream('movies2',{
    nodeLabels:['Movie','Genre','User','Person'],
    relationshipTypes:['GENRE','RATING','ACTED_IN'],
    sourceNodeFilter:'Movie',
    targetNodeFilter:'Movie'
})
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS Person1,
gds.util.asNode(node2).name AS Person2, similarity
ORDER BY similarity DESCENDING, Person1, Person2
```

We have a pattern in which we can now check Node similarity – We can use the filtered node similarity to filter out unwanted nodes and analyze the nodes we want. Using the Stream API of GDS enables to query the database faster and more efficiently while allowing the user to see the data. While using the Mutate or Write API is used for production use cases where the graph is usually changed.

We have made the projections similar to the Syntax provided in the GDS section. We can call the GDS API to obtain the filtered node similarity, following a syntax similar to the example provided above.

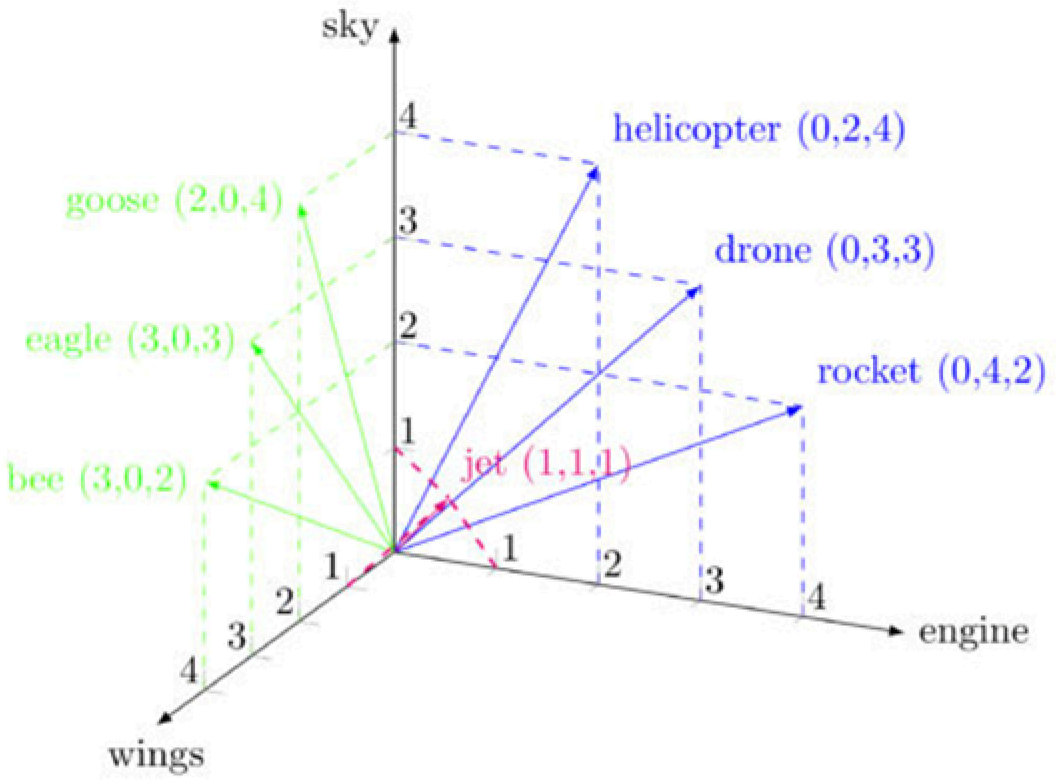
- nodeLabels – This is specified to give the algorithm the nodes to take into consideration while calculating similarity.
- relationshipTypes – The relationship around the nodes to calculate the shortest distance.
- sourceNodeFilter – The filter used to define the starting Node.
- targerNodeFilter – The filter used to define the node to compare the sourceNodeFilter with.

```
MATCH (p:Person)
WHERE tolower(p.name) = 'kate winslet'
WITH id(p) as actorNodeId
CALL gds.nodeSimilarity.filtered.stream('movies3',{
      nodeLabels:['Movie','Person'],
      relationshipTypes:['ACTED_IN'],
      sourceNodeFilter: actorNodeId,
      targetNodeFilter: 'Person',
      topk:20
})
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS Person1,
gds.util.asNode(node2).name AS Person2, similarity
ORDER BY similarity DESCENDING, Person1, Person2
```

Another example of Node similarity is shown above where we use the Node Similarity to find similar actors to a particular actor. We compute this by taking into consideration the Person node and ACTED_IN nodes. Writing a sub-query to extract the node ID of the actor in the beginning, we can see in the documentation we can pass the ID of the node to the sourceNodeFilter parameter of the API. This means that we start at a particular node and branch out to the surrounding nodes till we reach the target nodes all while computing the distance and in the end retrieving our similarity.

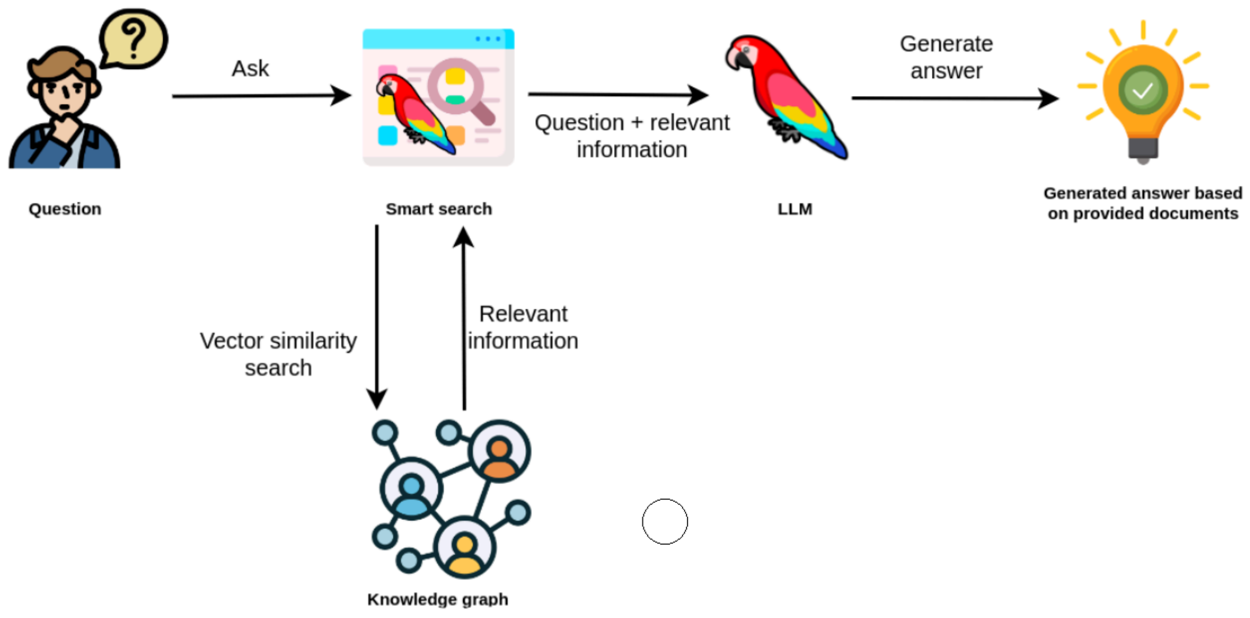
Vector Search in Neo4J

To extract very powerful insights from the graph we can enable vector searches on top of the database. Vector databases help leverage searches when asked natural language questions by comparing the numerical embedding of the question to the numerical embeddings of the data present in the database in the same space. While looking at the below example we can see that the words which have similar meanings are clustered near each other. Vector searches leverage this feature to help query information from texts faster.



Neo4j facilitates the storage of embeddings in the node properties. Considering we have the property – overview, we can convert this text to embeddings and store it in the Movie nodes. So to leverage Vector searches we need to have the following pre-requisites

- Large Language Model API Key and Endpoint – preferably OpenAI
- Genai Plugin from Neo4j
- All text related to Movies uploaded to the database
- Create a new environment and install Langchain



Steps to leverage vector search

1. Create an embedding property to the Movie node the values should be null. Assuming you have an empty property called embedding associated with your node we create a VECTOR INDEX on top of the property. This works in such a way that any value that gets populated there will be indexed as well.

```
CREATE VECTOR INDEX overview_embeddings2
FOR (m: Movie) ON (m.embedding)
OPTIONS {indexConfig: {
    `vector.dimensions`: 1536,
    `vector.similarity_function`: 'cosine'
}};
```

2. The index has been created and we use Langchain to bridge the gap between OpenAI and Neo4j using the Neo4jVector.from_existing_graph()

```
vector_index = Neo4jVector.from_existing_graph(
    embedding= OpenAIEmbeddings(),
    url=os.environ.get('NEO4J_URI'),
    username=os.environ.get('NEO4J_USERNAME'),
    password=os.environ.get('NEO4J_PASSWORD'),
    index_name='overview_embeddings2',
    node_label="Movie",
    text_node_properties=['overview','keyword','collection'],
    embedding_node_property='embedding',
    search_type = 'VECTOR'
)
```

3. This step is key to enable vector search. Running langchain creates the embeddings in the node property we created in step two. Once this embedding is created it gets auto-indexed to appear in the same space as the other overviews present in the other nodes.

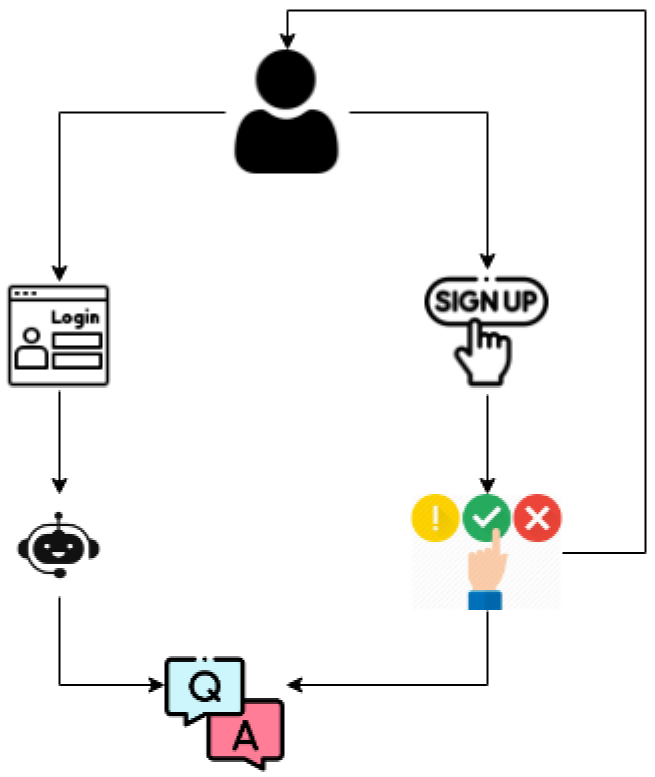
4. So now we can use the Genai plugin of neo4j to leverage vector search. If a question is asked the OpenAI key converts the question to an embedding similar to the embedding present in the node. Once the embeddings are generated we get the values that are closest by distance to each other.

```
WITH genai.vector.encode(
    $question,
    "OpenAI",
    {
      token: $openAiApiKey,
      endpoint: $openAiEndpoint
    }) AS question_embedding
CALL db.index.vector.queryNodes(
    'overview_embeddings2',
    $top_k,
    question_embedding
) YIELD node AS movie, score
WHERE movie.name IS NOT NULL  and
(:SpokenLanguage{name:'English'})-[:LANGUAGE]->(movie)
        RETURN movie.name, movie.overview, score
```

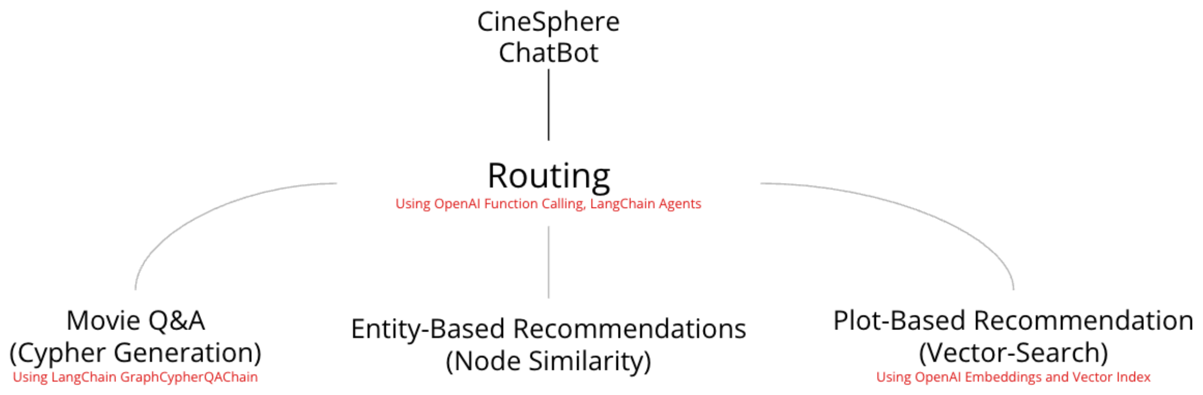
Personalized Recommendation

The application also leverages similarity to help recommend similar movies to users who sign up for the product.

- A new user who signs up will have access to the chat feature as well as a recommendation
- The recommendation works on a similar feature to the similarity algorithm mentioned in previous sections.
- The user will be created in the graph after they sign up and will be presented with some of the popular movies that are available in the application. They will have to be selected to understand their preferences.
- Once the movies are selected the similarity algorithm will start suggesting movies based on other user interactions.
- The more the user interacts with the applications, the more accurate the recommendations get.



Chatbot functionality (LangChain + OpenAI GPT LLMs)


CineSphere ChatBot
Routing
Using OpenAI Function Calling, LangChain Agents
Movie Q&A (Cypher Generation)
Using LangChain GraphCypherQAChain
Entity-Based Recommendations (Node Similarity)
Plot-Based Recommendation (Vector-Search)
Using OpenAI Embeddings and Vector Index

The chatbot provides a platform for the culmination of the various functionalities mentioned in the previous sections:

a. Node Similarity
b. Vector Search
c. Q&A Bot - Cypher Generation

We use a multi-agent tool created with the help of LangChain and the GPT3.5 model, which is used to answer any natural language question from the chat window. The model would understand the question and intelligently route it to either of the 3 functionalities mentioned on top. Users can ask questions ranging from general movie-related questions like “Give me the top-rated movies of Tom Cruise” to questions surrounding movies they feel like watching like “I am in the zone for a movie where the good guys lose?”

LangChain - GraphCypherQAChain, OpenAI Function Calling & OpenAI Embeddings

The application gets complex as we try to instate 3 different types of functionalities in our chatbot to make it robust to multiple types of questions. Once we separately create each of the functionalities, we add them to the model as a tool, using the OpenAI function calling feature. Below we explain each functionality and tools used to create it -

1) Q&A Bot - In case any user has general questions regarding movies, actors, overview, or crew in general this functionality can be used. It can be as simple a question like ‘Who were the actors in Dunkirk?’. With LangChain’s GraphCypherQAChain we create a structural prompt instructing the LLM to generate Cypher Queries based on the question asked by the User, which is passed to KG, and the answer is retrieved. We give few shot examples in the prompt for the LLM to get accustomed to the entities and relationships in the graph, For eg -

```
Human: Which actors have the most movies? What is the total number of
movies they acted in?
Assistant: ```MATCH (p:Person) -[a:ACTED_IN]-> (m:Movie) RETURN p.name as
Actor, count(distinct m) as totalmovies ORDER BY totalmovies DESC LIMIT
10```
```

2) Similarity - As it was explained in earlier documentation we’re using Node similarity for getting similar nodes in the Graph. Using the OpenAI Function calling feature we define this function as a tool and if the User has a query regarding getting similar entities to another entity the LLM would route the question to this tool. (The relationship can be among - user ratings, actor, genre, director, similar actor, nonsimilar actor, production house, country, language, work)
For eg.

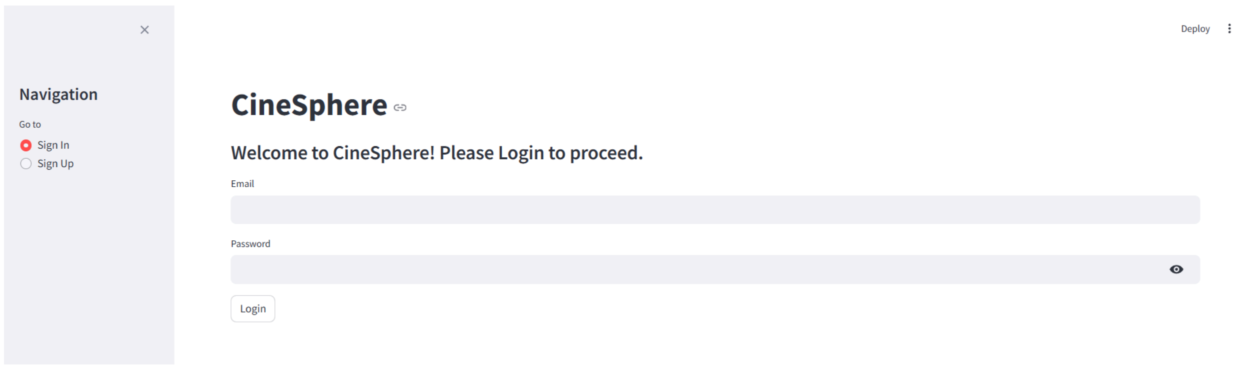
```
User : Recommend me similar actors like Tom Cruise
Here the LLM would detect entity as Tom Cruise and relationship as 'similar
actors' and find the answer
using Node Similarity
```

3) Plot Vector Search - We leverage OpenAI’s embedding model to create embeddings for the plot of the movie and create a vector index using cosine similarity on top of them. This is a unique feature in our chatbot and lets the user interact with the knowledge graph very casually to get movie recommendations. A question can be as simple as ‘I feel like watching a movie with Flying cars, robots etc’

Frontend - User Experience

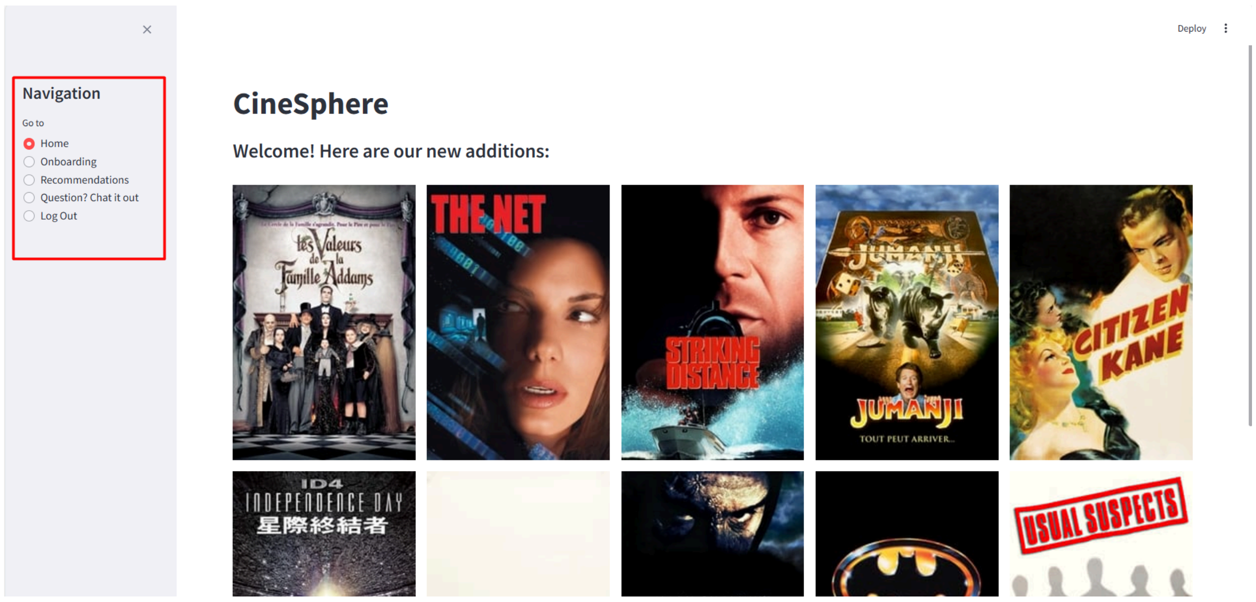
The whole application is wrapped up and presented to the user in a Streamlit Interface, where they can interact, get personalized recommendations, and ask any question they like. The user experience consists of the following steps -

1) Sign up - A new node is created in the graph for each new user
2) Login - As the user logs in, they are directed to the Home Page, with a navigation panel
3) Onboarding - The user is onboarded by selecting movies they like, and in the KG the links will start to create automatically
4) Recommendations - This tab gives recommendations based on their liked movies using Node similarity in the KG
5) Chat Interface - The user can ask any questions they have, with an easy chat interface
6) Logout



CineSphere

Welcome to CineSphere! Please Login to proceed.



CineSphere

Welcome! Here are our new additions: