

Evaluation Event Final - SlateLeapingDeer

Leyla Khasiyeva (*khasiyevl*) Necati Furkan Colak (*colakn*)

December 8, 2025

1 Introduction

The chatbot **SlateLeapingDeer** is a conversational agent that is designed to interact with users via the Speakeasy platform. It provides factual movie knowledge, deducts missing data using embeddings, returns content-based recommendations, and presents the queried multimedia content directly within the Speakeasy chat.

Key features include:

- **Robust Factual Querying:** Retrieves precise answers from a local RDF Knowledge Graph using SPARQL.
- **Inference Fallback:** Utilizes Knowledge Graph Embeddings (KGE) to return predicted answers for the missing data.
- **Advanced Recommendation:** Offers movie suggestions for single-hop queries using a Weighted Jaccard Similarity algorithm, supporting theme-based searches and constraints.
- **Intelligent Multimedia Retrieval:** Returns multimedia images. It distinguishes user intent (Person vs. Movie) to display the correct image type: Headshots for actors, Posters for films, and Backdrops for scenes—directly in the chat interface.

The final phase focused on improving the bot to handle wider range of recommendation questions and improve the recommendation logic, and to implement a robust multimedia pipeline that resolves local file paths for remote rendering.

2 Capabilities

Agent Architecture Overview

Our agent operates on a modular pipeline:

1. **Input:** The user query is received in a natural language format within the Speakeasy chat.

2. **Intent Detection & Routing:** The `responser.py` agent first checks for **Multimedia** queries by applying regex patterns. If detected, it delegates the input to the Image Resolver. If multimedia is not detected, it checks the input for **Recommendations**; if found, it passes the input to the Recommender Engine. Finally, if the input did not qualify for any other pipeline, by default, the input is sent to the **Factual/Embedding** pipeline.
3. **Execution:**
 - *Multimedia:* Resolves entity names, determines image type that is being asked (Profile/Poster/Backdrop), and fetches the correct file path based on type.
 - *Recommendation:* Extracts seeds and applies constraints if relevant, and calculates similarity scores to return the most similar and relevant movies as recommendations.
 - *Factual/Embedding:* Attempts SPARQL; if it fails, falls back to TransE embedding prediction to return the most likely response for the missing data.
4. **Output:** Returns a formatted natural language response or a formatted image path that is loaded as a visual output (`image:FOLDER/HASH`) to the user.

Question Type Capabilities

- **Factual Questions**
Our agent answers factual questions in three steps: Entity/Relation Extraction, SPARQL Generation, and Resolution. We use regex-based extraction (`core/entity.py`) to identify entities and map relations to Wiki-data properties. We handle inverse queries (e.g., “Which movies did X direct?”) and apply a label resolver to ensure only human-readable answers are returned.
- **Embedding Questions**
When factual data is missing, the agent applies a **TransE-based embedding model**. It $t_{target} = h + r$ where (h) is head entity and (r) is relation. It finds the nearest neighbor in the vector space, returning the predicted entity along with its type to provide the most probable answer.
- **Multimedia Questions**
Our agent distinguishes between requests for **Profiles** (“Show me Brad Pitt”), **Posters** (“Poster of Titanic”), and **Backdrops** (“Scene from Inception”). It resolves the entity to an IMDb ID ($P345$) and retrieves the corresponding local file path from `images.json` map. The path is formatted specifically for Speakeasy rendering.

- **Recommendation Questions**

The agent supports both **Seed-Based** (“Movies like Matrix”) and **Theme-Based** (“Recommend horror movies”) queries. It extracts optional constraints (Genre, Rating, Language) and filters the results, ensuring high relevance by applying prioritization techniques.

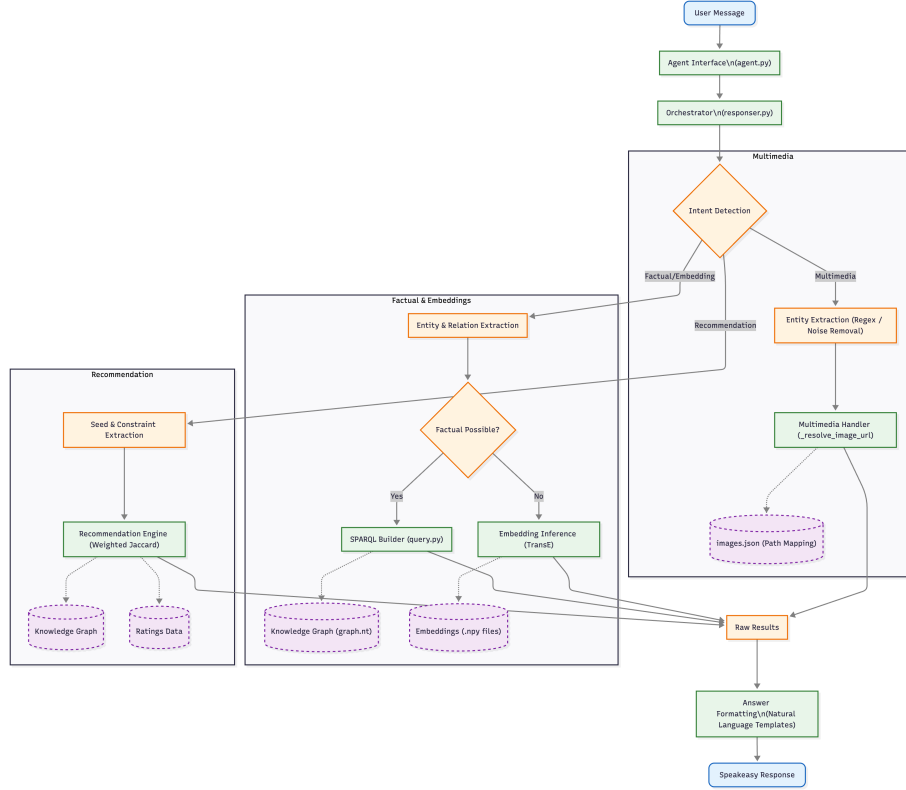


Figure 1: Architecture Overview: The system flow from User Input through the Orchestrator (Responser), splitting into specific logic pipelines (Multimedia, Recommendation, Factual), retrieving data from the abstracted Data Layer, and formatting the final output.

What We Have Added Since the Last Milestone

- We start by extracting the part of the sentence that contains the example films and split that fragment by commas. Then we scan for any “and” or “or.” If there is only a single conjunction, we compare the lengths of the text on each side and check whether the right side begins with *the*, *a*, or *an*. This keeps titles like *Beauty and the Beast* intact, while phrases such

as *Forrest Gump* and *The Lord of the Rings* are correctly split into two separate titles.

- When a fragment contains multiple conjunctions, we switch to a more aggressive rule: whenever the next chunk starts with a capitalized word (or with *the/a/an* after a longer chunk), we treat it as a new movie title. Otherwise, we keep the words together so that *Pride and Prejudice* does not get broken apart.
- Finally, we strip off leading “and/or,” remove punctuation, and clean up repeated pieces before adding the resulting titles to the recommendation seed list.

Pseudo-Recommendation Questions (Added after Milestone 3)

- Pseudo-recommendation questions look like recommendation requests but actually ask for a full catalog of works tied to a person, company, or language. Instead of going through the top-2 recommender, the system detects this pattern, rewrites the message into a factual query (e.g., “Which biographical movies has Meryl Streep as cast member?”), and routes it through the knowledge-graph QA pipeline.
- **Actor and genre detection**
The actor detector identifies the phrasing “I like Meryl Streep, recommend me some biographical films?” and extracts both the actor and the genre “biographical.”
- **Message rewrite**
`generate_response` rewrites the input as “Which biographical movies has Meryl Streep as cast member?”, converting the request into a direct SPARQL query (P161 + P136=Q645928).
- **Skipping the recommendation branch**
The rewritten message no longer matches recommendation heuristics, so the system bypasses the recommender and proceeds through the factual QA path.
- **Factual query execution**
The pipeline resolves “Meryl Streep” to her URI, recognizes the query as inverse (films where she appears in P161), and constructs a SPARQL query that:
 - selects movies where ?movie wdt:P161 wd:Q8733,
 - filters to films (Q11424),
 - applies the biographical genre filter (Q645928).This returns all matching films, not just two recommendations.
- **Answer formatting**
Results are returned as a simple list joined with the word “and”.

3 Adopted Methods

- **RDF/SPARQL**

We used `rdflib` to query our local Knowledge Graph. This ensures 100% accuracy for known facts and allows for filtering (e.g., “What genre is the movie ‘Bandit Queen’?”).

- **Knowledge Graph Embeddings (TransE)**

We adopted TransE to serve as a critical fallback mechanism, ensuring the bot almost always returns an output even when a link is missing in the graph.

- **Weighted Jaccard Similarity (Recommendation)**

For recommendations, we implemented a **Weighted Jaccard Similarity** metric. We assign higher weights to **Genre (4.0)** and **Production Company (4.0)** compared to Cast (0.8), ensuring recommendations are similar enough to the given movie. We also apply a **Genre Data Quality Gate**, discarding candidate movies that lack genre metadata to prevent low-quality suggestions.

- **Strict Noise Removal**

To handle the variability of natural language, we implemented noise stripping in `core/responser.py`. Instead of matching exact phrases, we iteratively strip “noise words” (e.g., “Can I see”, “show me”, “picture of”) to isolate the core entity name. This allows the bot to handle complex phrasings like “*Can I see Rihanna’s picture please?*” as effectively as “*Rihanna pic*”.

- **JSON-Based Image Mapping**

We utilized the `images.json` file to build a sophisticated lookup map. Rather than a simple 1-to-1 mapping, we parse the JSON to categorize images into **Profiles**, **Posters**, and **Backdrops**. This allows us to enforce logic where a request for a person returns a headshot (if available), falling back to a movie poster only if necessary, while explicit requests for posters always retrieve the relevant movie poster.

4 Examples

Examples relevant to the development of this stage:

- **Recommendation Questions**

Given “*Recommend movies like Notting Hill and What Happened Was?*”, the agent extracts the seeds and calculates similarity. It finds shared properties (Romance, Comedy) and returns “*Based on your preferences, I recommend Next Stop Wonderland and Jersey Girl.*”

- **Multimedia Questions**

Given “*Show me a picture of Halle Berry*”, the agent: 1) Detects the intent is **PROFILE** (Person). 2) Removes noise (“Show me a picture of”) to extract “Halle Berry”. 3) Resolves her to Wikidata URI Q73307 and then to IMDb ID nm0000932. 4) Looks up nm0000932 in the **profile** image map loaded from **images.json**. 5) Returns the formatted command: **image:0344/knLBj1WP360Su1A0Btbyy9r1WXH**, which renders her headshot.

For “*Poster of Titanic*”, the agent detects **POSTER** intent, and returns the movie poster image.

5 Additional Features

- **Context-Aware Multimedia Retrieval**

The agent intelligently distinguishes between image types. A query for “Brad Pitt” returns his profile picture. We also handle **Backdrop** queries (e.g., “Show me a scene from Interstellar”), which retrieves specific **type=’backdrop’** images from our JSON map.

- **Speakeasy Image Rendering**

We solved the challenge of displaying local images in the web chat. Instead of returning raw file paths (e.g., **data/images/001.jpg**), we format the output as **image:<FOLDER>/<HASH>** (e.g., **image:0344/knLB...**). This command tells the Speakeasy to render the image directly within the chat.

- **Ambiguity Resolution Heuristic**

To handle ambiguous titles like “Frozen” (2013 vs 2010) or “Titanic” (1997 vs 1953), we implemented a **Prioritization Sort**. When a user asks for a movie (especially for posters), we rank candidates by **Film Type**, then **User Rating**, and finally **Release Date**. This ensures the most popular and relevant movie is selected automatically.

6 Conclusions

For the final evaluation, we successfully integrated all required pipelines: Factual, Embedding, Recommendation, and Multimedia. We created a combination of **strict data logic** (for mapping IDs) and **flexible NLP** (for stripping noise) for a robust agent that is capable of handling a wide variety of real-world natural language queries.