



Universidad Nacional de Rosario  
Facultad de Ciencias Exactas, Ingeniería y Agrimensura

## **IA4.2 Procesamiento del Lenguaje Natural**

### **Trabajo Práctico N°2**

Tecnicatura Universitaria en Inteligencia Artificial

Integrantes:

Gonzalo Asad, A-4595/1

Docentes:

Juan Pablo Manson

Alan Geary

Constantino Ferrucci

Dolores Sollberger

Fecha: 18/12/2024

<b>1. RESUMEN</b>	<b>4</b>
<b>2. INTRODUCCIÓN</b>	<b>5</b>
<b>3. METODOLOGÍA</b>	<b>6</b>
3.1 Entorno de trabajo	6
3.2 Fuentes de datos	7
3.3 Métodos y técnicas	9
3.3.1 Web-scraping	9
3.3.2 Embedding	9
3.3.3 Random Forest	9
3.3.4 LLM	9
3.3.5 Agente	9
3.4 Pasos para la resolución	9
<b>4. EJERCICIO 1: RAG</b>	<b>11</b>
4.1 Resumen	11
4.2 Desarrollo e implementación	12
4.2.1 Hito 1: Generación de bases de datos	12
4.2.2 Hito 2: Creación de clasificadores	16
4.2.3 Hito 3: Creación del retriever	17
4.2.4 Hito 4: Creación del reranker	19
4.2.5 Hito 5: Implementación del RAG	20
4.2.6 Hito 6: Programa principal para el primer algoritmo (Main)	21
4.3 Resultados y conclusiones	23
4.4 Enlaces a modelos y librerías utilizadas	23
4.4.1 Enlaces a librerías	23
4.4.2 Enlaces a modelos	24
<b>5. EJERCICIO 2: AGENTE</b>	<b>25</b>
5.1 Resumen	25
5.2 Desarrollo e implementación	26
5.2.1 Hito 7: Implementación del Agente	26
5.2.2 Hito 8: Programa principal para el segundo algoritmo (Main)	27
5.3 Resultados y conclusiones	27
5.3.1 Uso de múltiples herramientas	28

5.3.2	Situaciones de falla	30
5.3.3	Ideas finales	32
5.4	Enlaces a modelos y librerías utilizadas	33
5.4.1	Enlaces a librerías	33
5.4.2	Enlaces a modelos	33

## 1. RESUMEN

---

El presente trabajo consistió en el desarrollo de dos soluciones para un chatbot enfocado al juego de mesa “Rajas de Ganges”. La primera solución fue implementada mediante el desarrollo de un RAG, contemplando todas sus etapas (prompt del usuario -> clasificador -> retriever -> re-ranker -> LLM) haciendo uso de tres bases de datos de distintos tipos: vectorial, de grafos y tabular. La segunda solución se implementó mediante el uso de un agente basado en Ollama y re-aprovechando funciones y bases de datos originadas para el RAG.

Se observará que el RAG mantiene un buen desempeño en general, respondiendo correctamente la pregunta del usuario la mayoría de las veces, mientras que el agente tiende alucinar con frecuencia.

## 2. INTRODUCCIÓN

Este trabajo se desarrolló pensando en brindarle un servicio a una persona que necesite no únicamente conocer las reglas del juego Rajas de Ganges, sino también conocer información relevante del mismo, como stats, creadores, expansiones y reseñas. Con este motivo, se desarrollaron dos alternativas a un chatbot que pudiera informar al usuario en lo que necesite conocer.

El objetivo principal del trabajo fue el de desarrollar dos algoritmos que lograran comprender la pregunta del usuario y hacer uso de bases de datos provistas para armar una respuesta que tuviera sentido. Para esto, fue necesario lograr filtrar con exactitud la información contextual de las bases de datos correspondientes, de manera que un LLM pudiera usarla en conjunto al prompt del usuario para generar una respuesta *sin salir a buscar información de otros lugares*.

El algoritmo del RAG se considera que superó las expectativas, luego de haber superado una serie de pruebas exhaustivas, basadas en aspectos objetivos y subjetivos.

El presente informe comienza abordando la metodología empleada a lo largo del trabajo, continuando con los detalles sobre su desarrollo e implementación. Luego se muestran resultados de distintas evaluaciones y posteriormente se exponen las conclusiones alcanzadas. Por último, se incluyen referencias y anexos.

### 3. METODOLOGÍA

#### 3.1 ENTORNO DE TRABAJO

Para el desarrollo de ambos algoritmos, se optó por utilizar el lenguaje de programación Python debido a su amplio uso en la carrera y a la abundante documentación disponible en la web. El código se escribió en el entorno de desarrollo integrado Google Colab, que ofrece herramientas no solo para la escritura de código, sino también para la depuración de problemas, ejecutándose en un servidor virtual en la nube con recursos de hardware proporcionados por Google.

En el espacio de trabajo se debieron instalar las siguientes librerías:

- *Para base de datos de grafos*
  - NetworkX: Para crear redes de grafos.
  - Matplotlib: Para imprimir imágenes.
  - Redis: Para crear un servidor a donde consultar la base de datos de grafos usando Cypher.
  - Redisgraph: Para crear la base de datos de grafos.
- *Para base de datos vectorial*
  - Tensorflow\_text: Para crear embeddings de textos.
  - Tensorflow\_hub: Para cargar modelos de embeddings.
  - ChromaDB: Para crear colecciones de vectores.
  - Langchain: Para particionar cadenas de textos.
  - YouTube-Transcript-API: Para extraer transcripciones de videos.
  - PyPDF2: Para extraer textos de archivos PDF.
- *Para web-scraping*
  - Requests: Para hacer consultar HTTP.
  - BeautifulSoup4: Para trabajar con HTML.
- *Para uso general*
  - GDown: Para descargar archivos de un Drive de Google.
- *Para clasificación*
  - Sentence\_Transformers: Para generar embeddings de texto.
- *Para RAG*
  - Python-Decouple: Para hacer configuraciones.
  - LLM-Templates: Para formatear prompts en base al modelo Zephyr.
- *Para el Agente*
  - Llama-Index-LLMs-OLLama: Para modelos LLM de OLLama.
  - Llama-Index: Para utilizar OLLama.

También, se hizo uso de las siguientes librerías que no necesitaron instalación adicional:

- *Pandas*: Para generar DataFrames.

- *JobLib*: Para cargar al entorno de trabajo modelos pre-entrenados de clasificación.
- *Warnings*: Para filtrar advertencias.
- *Time*: Para medir tiempos de ejecución.
- *Re*: Para utilizar expresiones regulares.
- *Json*: Para poder trabajar con archivos Json.
- *HuggingFace\_Hub*: Para utilizar modelos de Hugging Face.
- *SickitLearn*: Para generar modelos de machine learning y evaluarlos.
- *Numpy*: Para trabajar con arreglos de números.

## 3.2 FUENTES DE DATOS

Se generaron tres bases de datos para brindar información contextual a las preguntas del usuario:

- *Base de datos tabular*: consiste en un DataFrame que contiene información referente a los stats y características principales del juego extraídas del sitio web de BGG. Contiene los datos:
  - Game: nombre del juego.
  - Publishing Year: año de publicación del juego.
  - Min. Amount Players: mínimo número de jugadores.
  - Max. Amount Players: máximo número de jugadores.
  - Best Amount Players: sugerencia sobre el número óptimo de jugadores basado en reseñas.
  - Min. Playing Time (mins): cantidad mínima de minutos de juego estimada.
  - Max. Playing Time (mins): cantidad máxima de minutos de juego estimada.
  - Ages: rango de edades recomendado para jugarlo.
  - Avg. Rating: puntaje promedio dado por usuarios del sitio BGG Rank: posición dentro del ranking.
  - No. of Ratings: número de votantes para la puntuación pública del juego.
  - Std. Deviation: desviación estándar del puntaje del juego.
  - Weight: dificultad del juego en escala del 1 al 5.
  - Comments: cantidad de comentarios (en miles) en la web de BGG.
  - Fans: cantidad de fans del juego.
  - Page Views: cantidad de vistas de la página de BGG sobre el juego.
  - Overall Rank: rango dentro del listado de BGG.
  - Strategy Rank: rango dentro de los juegos de estrategia en el listado de BGG.
  - All Time Plays: cantidad de veces que se registró el juego como jugado en toda su historia.
  - This Month Plays: cantidad de veces que se registró el juego como jugado el mes actual.
  - Own: cantidad de gente que posee el juego.
  - Prev Owned: cantidad de gente que alguna vez poseyó el juego.

- For Trade: cantidad de gente que desea dar el juego para intercambiarlo por otro.
- Want In Trade: cantidad de gente que está dispuesto a tomar el juego como parte de un intercambio.
- Whishlist: cantidad de gente que desea tener el juego.
- Has Parts: cantidad de gente que posee repuestos para el juego.
- Want Parts: cantidad de gente que necesita repuestos para el juego.
- **Base de datos de grafos:** consiste en nodos inter relacionados que contienen información acerca de los créditos del juego y las relaciones que hay entre los diseñadores, artistas y otras características según el sitio web de BGG. Contiene las siguientes entidades:
  - Diseñadores
  - Artistas
  - Nombre primario
  - Nombres alternativos
  - Categorías
  - Mecanismos
  - Publicadores
  - Juegos relacionados
  - Expansiones

Tiene también información sobre las siguientes relaciones:

  - Tiene diseñador (HAS\_DESIGNER)
  - Tiene artista (HAS\_ARTIST)
  - Tiene nombre alternativo (HAS\_ALTERNATIVE\_NAME)
  - Tiene categoría (HAS\_CATEGORY)
  - Tiene mecanismo (HAS\_MECHANISM)
  - Tiene publicador (HAS\_PUBLISHER)
  - Diseñó (DESIGNED)
  - Hizo el arte (ARTWORKED)
  - Tiene expansión (HAS\_EXPANSION)
- **Base de datos vectorial:** consiste en documentos de texto y transcripciones de videos con reglas, objetivos, reseñas y opiniones del juego, que son vectorizados y luego almacenados en una colección de ChromaDB. Para las reglas y objetivos, se tienen como origen cuatro documentos PDF para sus cuatro modos de juego: Normal, AI, Solo y Automa. Para las reseñas y opiniones, se extrajeron las transcripciones de cuatro videos de YouTube.

Adicionalmente, se debió generar una base de datos que tuviera diferentes palabras o frases que tuvieran posibles preguntas junto con etiquetas clasificatorias, para entrenar un modelo que sirviera como clasificador. Dicho dataset contiene únicamente dos variables:

- **classification:** etiqueta clasificatoria del prompt del usuario, puede asumir cuatro valores: review, rules, credits y stats.



- prompts: posibles preguntas sobre el juego.

Esta base de datos fue generada con la ayuda de ChatGPT, que tras muchas iteraciones de consiguió brindar una base de datos balanceada de más de 100 elementos.

### 3.3 MÉTODOS Y TÉCNICAS

Si bien el detalle de la implementación se describe en la unidad siguiente, aquí describimos los métodos y técnicas de PLN principales utilizados por el algoritmo.

#### 3.3.1 Web-scraping

Es un conjunto de técnicas para la extracción de datos contenidos en el código *html* de una página web. Apoyados en las librerías de Python *Beautiful Soup* y *Requests*, se navegó el portal de *BGG* para extraer el código *html* de su página con el contenido de los créditos del juego de “Rajas de Ganges”. Luego, se utilizaron funciones para filtrar contenido dentro del código, extrayendo las porciones de texto que serían de interés a la hora de armar la base de datos.

#### 3.3.2 Embedding

Consiste en generar representaciones vectoriales dentro de un espacio multidimensional de frases o palabras. Se utilizaron diversas librerías en distintos puntos del algoritmo, donde se obtuvieron incrustaciones (embeddings) de frases, usando como base modelos pre-entrenados.

#### 3.3.3 Random Forest

Modelo que permite clasificar dentro de distintas categorías a los datos, habiendo sido entrenado previamente con un dataset etiquetado (aprendizaje supervisado). Se utilizó para generar el modelo que clasifica las preguntas del usuario dentro de cuatro categorías: review, rules, credits y stats.

#### 3.3.4 LLM

Son modelos de lenguaje enormes, que permiten no solamente realizar embeddings, sino también realizar tareas y responder preguntas de usuarios. Se usaron muchos LLMs en diferentes puntos del algoritmo, sobre todo en clasificación y en RAG.

#### 3.3.5 Agente

Un agente en PLN es un sistema de software diseñado para interactuar con usuarios o con otros sistemas, utilizando el lenguaje natural como medio de comunicación. Se debió utilizar para el segundo algoritmo.

### 3.4 PASOS PARA LA RESOLUCIÓN

El trabajo fue abordado por etapas o hitos. Se dividió el problema en subproblemas a resolver con la generación de distintas funciones para atacarlos.

- *Hito 1*: generación de bases de datos de distintas fuentes y características.

- *Hito 2*: creación de los clasificadores para determinar las ubicaciones de los datos contextuales asociados al prompt del usuario.
- *Hito 3*: desarrollo del retriever que se encarga generar queries dinámicas, dónde únicamente la información útil es extraída.
- *Hito 4*: desarrollo del re-ranker para calificar y refinar la búsqueda sobre los datos de la base de datos vectorial.
- *Hito 5*: implementación del RAG para enviar el prompt más los datos contextuales que luego serían utilizados por un LLM.
- *Hito 6*: desarrollo de la función Main para el primer algoritmo.
- *Hito 7*: generación del agente para el segundo algoritmo del chatbot.
- *Hito 8*: desarrollo de la función Main para el segundo algoritmo.

## 4. EJERCICIO 1: RAG

### 4.1 RESUMEN

Consistió en crear un chatbot experto en un juego de mesa estilo Eurogame, puntualmente “Rajas de Ganges”, usando la técnica RAG (Retrieval Augmented Generation). Como fuentes de conocimiento, se debieron utilizar al menos tres fuentes:

- Documentos de texto
- Datos numéricos en formato tabular
- Base de datos de grafos

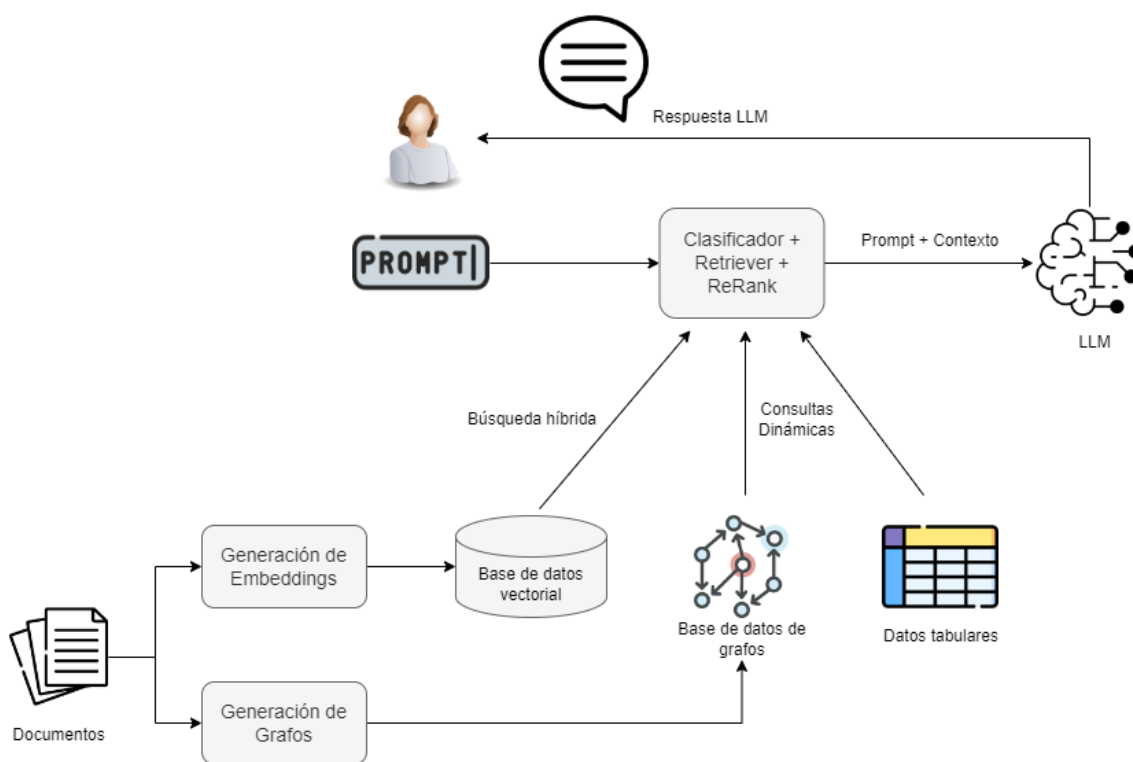


Figura 1: Esquema de RAG utilizado.

El sistema puede llevar a cabo una conversación en lenguaje español o inglés. El usuario puede hacer preguntas que el chatbot intenta responder a partir de datos de algunas de sus fuentes. El asistente debe poder clasificar las preguntas, para saber qué fuentes de datos utilizar como contexto para generar una respuesta. Las respuestas deben ser en el mismo lenguaje de la consulta del usuario.

Se le sumaron también una serie de requisitos a cumplir, describiendo cosas como el tipo de técnicas a abordar y algunas restricciones.

## 4.2 DESARROLLO E IMPLEMENTACIÓN

Tal como fue descrito en la anteriormente, la resolución del algoritmo fue abordada en muchas etapas o hitos, descomponiendo el problema en varios subproblemas. Durante esta sección, se describe la resolución de cada uno de ellos.

### 4.2.1 Hito 1: Generación de bases de datos

Se abordó la generación de cada base de datos como un problema individual, desarrollando funciones que trabajaran de forma independiente, donde parte de la colecta de datos se realizó de forma automática y parte de forma manual.

#### 4.2.1.1 Base de datos vectorial

Se realizó la descarga de 13 documentos que contenían guías, objetivos y reglas del juego para cuatro modos de juego distintos: normal, contra una IA, en solitario y Automa. Se clasificaron los textos en esas cuatro categorías y se generaron cuatro nuevos documentos que los contemplaran por completo, haciendo también una pequeña limpieza de formato y disponiéndolos públicamente en un Drive de Google. Por otro lado, se buscaron reseñas del juego en YouTube y se guardaron los enlaces de sus videos. A continuación, se trabajaron todos los documentos en el notebook en Google Colab.

##### 4.2.1.1.1 La función `getTranscripts`

La función crea una lista con las transcripciones de lo que se comenta en videos. La función recibe como único parámetro:

- `Video_ids[list]`: Lista que contiene los IDs de los enlaces de videos de YouTube.

Esta función itera sobre los IDs de los videos y usa una API de YouTube dedicada a realizar transcripciones, generando una lista con las mismas. A continuación, realiza una limpieza de saltos de línea y espacios dobles, retornando la lista de transcripciones refinada a su salida.

##### 4.2.1.1.2 La función `getPDFs`

La función crea una lista con los textos extraídos de documentos PDF. La función recibe como único parámetro:

- `File_ids[list]`: Lista que contiene los IDs de los enlaces públicos de archivos PDFs almacenados en una cuenta de Google Drive.

Esta función itera sobre los IDs de los enlaces de los documentos y realiza la descarga al entorno de trabajo mediante el uso de *Gdown*. Luego abre los archivos de a uno, extrayendo las páginas con funciones de la librería *PyPDF2*. A continuación, realiza una limpieza de saltos de línea y espacios dobles, retornando la lista de textos extraídos refinada a su salida.

#### 4.2.1.1.3 La función `splitter`

La función crea particiones de textos recibidos, además de crear sus metadatos y sus IDs para usar en una colección de *ChromaDB*. La función recibe como parámetros:

- `Texts[list]` : Lista con los textos a particionar.
- `Metadatas[list]` : Lista con los nombres a usar como metadatos.
- `Ids_names[str]` : Cadena de texto con el nombre a usar en los IDs.

Comienza creando un particionador usando *RecursiveCharacterTextSplitter* para generar *chunks* de texto con cierto *overlap*. Luego itera la lista de textos a particionar y aplicándoles el particionador. Por cada partición generada, se crea una metadata nueva, usando de base el nombre entregado como argumento a la función y agregando un índice por cada partición. Luego, se crean los IDs usando de base el nombre recibido como argumento y agregando un índice por cada partición.

Finalmente, se retorna a la salida tres listas: una con las particiones del texto, una con la metadata y otra con los IDs.

#### 4.2.1.1.4 La función `createCollection`

Esta función es la encargada de generar la base de datos como tal haciendo uso de las funciones anteriores y aplicando otras técnicas, retornando una colección de datos de *ChromaDB* a su salida. La función no recibe ningún argumento.

Comienza usando las funciones *getTranscripts* y *getPDFs* para obtener las listas de transcripciones y textos de videos y documentos. A continuación, usando la función *splitter* obtiene las particiones de los textos, sus metadatos y sus IDs para usar en colecciones de *ChromaDB*.

Luego, genera una colección de *ChromaDB* usando un cliente que, en su momento, estará generado en el entorno de trabajo. Combina las listas de textos, metadatos e IDs de videos y documentos para luego generar los *embeddings* de los textos, usando un modelo de codificador universal multilingüe de oraciones, que también será generado en el entorno de trabajo más adelante.

Finalmente, crea la colección combinando los textos, con sus incrustaciones, metadatos e IDs, retornándola a la salida.

#### 4.2.1.2 Base de datos tabular

Se relevó información referente a características básicas del juego, como cantidad de jugadores, minutos de juego y edades, además de los stats de el sitio dedicado a “Rajas de Ganges” en BGG. Se generó luego un archivo CSV que contuviera todos los datos y se dispuso públicamente en un Drive de Google. Luego, se trabajó el archivo mediante funciones personalizadas para generar un *DataFrame* dentro del entorno de trabajo.

#### 4.2.1.2.1 La función `getDataFrame`

Esta función crea un *DataFrame* a partir de un archivo CSV alojado en Google Drive, retornándolo a su salida. Recibe como argumentos:

- `File_id[str]`: ID del archivo (codificado según Google Drive).
- `Encoding[str]`: Codificación de caracteres que se utilizará para leer el archivo.
- `Delimiter[str]`: Especifica el caracter que separa los valores en el archivo.
- `Show_head[bool]`: Si es `True`, muestra las primeras filas del *DataFrame*.

La función `getDataFrame` comienza construyendo la *URL* de descarga del archivo, concatenando la URL base de Google Drive con el ID proporcionado como argumento. Luego, utiliza la función `gdown.download` para cargar el archivo en el entorno de trabajo, empleando la URL generada y un nombre para el archivo de salida. Finalmente, crea un *DataFrame* a partir del archivo CSV descargado, aplicando las configuraciones de codificación y delimitadores recibidos como argumentos, y lo retorna como salida.

#### 4.2.1.2.2 La función `createTables`

Esta función crea un *DataFrame* del CSV generado para esta base de datos, usando la función descrita anteriormente. No recibe ningún parámetro.

#### 4.2.1.3 Base de datos de grafos

Consiste en información del juego con sus relaciones, principalmente de los créditos, pero también de sus mecánicas, categorías, expansiones y juegos alternativos desarrollados por sus creadores. Parte de la colecta de datos para el grafo se realiza de forma automática, mientras que otra parte se realiza de forma manual.

##### 4.2.1.3.1 La función `getValidation`

Valida la petición *GET* a una *URL* y retorna la respuesta si el caso es exitoso. Caso contrario, implementa reintentos con *backoff* exponenciales para manejar los errores de *timeout*. La función recibe como argumentos:

- `Url[str]`: URL a la cual se le quiere realizar la petición.
- `Retries[int]`: Cantidad de reintentos de reconexión.
- `Backoff_factor[float]`: Factor que afecta el tiempo de espera antes de reintentar una reconexión.

La función `getValidation` realiza un *try-except* extrayendo el estado de la transacción durante el pedido *GET*. En caso de recibir una excepción, devuelve el código de error y realiza la cantidad de reintentos especificados como argumento de la función, aumentando el tiempo de espera entre reintentos de acuerdo al factor también ingresado como argumento, para no sobrecargar de solicitudes a la página. De superarse la cantidad de reintentos devuelve un mensaje de error de solicitud. Si la solicitud es exitosa, retorna el mensaje *HTML* completo recibido como respuesta.

#### 4.2.1.3.2 La función `createGameList`

Crea una lista con nombres de juegos asociados de una persona mediante el uso de *web-scraping*. La función recibe los parámetros:

- `Url[str]` : *URL* sobre la que se realizará el *web-scraping*.
- `Credit_types[str]` : Cadena de texto que indica el tipo de crédito (*boardgamedesigner* o *boardgameartist*).

Comienza entregando la *URL* a la función *getValidation* y recibiendo como respuesta el mensaje *HTML* de la página. Luego extrae el texto y mediante el uso de marcadores, realiza una búsqueda de sus índices en la cadena.

Finalmente, genera una cadena *Json* que es iterada para extraer los nombres de los juegos de acuerdo al tipo de crédito ingresado como parámetro, retornando una lista con todos los nombres de juegos extraídos a su salida.

#### 4.2.1.3.3 La función `createCreditData`

Genera *DataFrames* con datos de créditos, mecánicas y demás informaciones del juego. La función recibe los parámetros:

- `game_names_inka_brand[list]` : Lista con nombres de juegos de mesa de Inka Brand.
- `game_names_markus_brand[list]` : Lista con nombres de juegos de mesa de Markus Brand.
- `game_names_dennis_lohaussen[list]` : Lista con nombres de juegos de mesa de Dennis Lohaussen.

La función genera *DataFrames* de *Pandas* con información de:

- Diseñadores
- Artistas
- Nombre primario del juego
- Nombres alternativos del juego
- Categorías del juego
- Mecanismos del juego
- Publicadores
- Juegos diseñados por Inka Brand
- Juegos diseñados por Markus Brand
- Juegos trabajados artísticamente por Dennis Lohaussen
- Expansiones

Retornándolos a la salida.

#### 4.2.1.3.4 La función `createGraph`

Crea un grafo de `RedisGraph` usando la información generada por las funciones anteriores. La función no recibe parámetros, pero hace uso de un grafo que será creado en el entorno de trabajo más adelante.

Comienza con el *web-scraping* de los juegos creados por los diseñadores y artistas usando la función `createGameList`, para luego enviar las listas generadas a la función `createCreditData`. A continuación, se crean los nodos para cada tipo de entidad con sus propiedades correspondientes y agregando etiquetas, iterando sobre cada `DataFrame` y agregándolos a listas.

Se itera cada lista de nodos para añadirlos al grafo. Luego, se van iterando las listas de nodos agregando las aristas que representan las relaciones entre entidades.

### 4.2.2 Hito 2: Creación de clasificadores

Se crearon dos clasificadores distintos, según fue pedido en el enunciado del trabajo: uno que trabajase con un modelo entrenado con ejemplos y *embeddings*; y otro basado en LLM. Esto se realizó con el objetivo de comparar resultados y elegir uno de ellos para el código principal. Los clasificadores sirven para determinar a qué base de datos ir a buscar información, ya que la clasificación del texto les da una identidad a las preguntas provistas por el usuario. Por este motivo, se crearon diversas funciones que luego serían ejecutadas en el código principal.

#### 4.2.2.1.1 La función `randomForestClassifierCreator`

Esta función genera y retorna un modelo de Random Forest para la clasificación de textos en cuatro categorías: review, rules, stats y credits. Recibe un único argumento:

- `Production[bool]`: Si es Falsa, se calculan y muestran las métricas de prueba.

La función `randomForestClassifierCreator` carga al entorno de trabajo una base de datos para entrenamiento y un modelo pre-entrenado, específicamente el modelo “distiluse-base-multilingual-cased-v1” usando `SentenceTransformer`. A continuación, recorre la base de datos y la almacena en una lista de tuplas para su procesamiento, separando las variables de entrada (prompt) y de salida (classification). A continuación, divide el dataset en datos de entrenamiento y de prueba, en relación 80-20 y calcula las incrustaciones de los datos de entrenamiento usando el modelo de `SentenceTransformer`. Finalmente, crea un modelo de Random Forest y lo entrena con los datos de entrenamiento, retornándolo como salida. En el caso donde la variable *production* es Falsa, se calculan las incrustaciones de los datos de prueba y se muestran en pantalla la exactitud del modelo junto con el reporte de clasificación del Random Forest.

#### 4.2.2.1.2 La función `rfClassifier`

Esta función realiza la clasificación del texto en sus cuatro categorías descriptas con anterioridad. Recibe como parámetros:



- `Modelo[RandomForestClassifier]`: Modelo de Random Forest entrenado para clasificación de prompts.
- `Prompt[str]`: Prompt el usuario a ser clasificado.

Carga un modelo pre-entrenado, específicamente el modelo “distiluse-base-multilingual-cased-v1” usando *SentenceTransformer*. A continuación, adapta la cadena de texto correspondiente al prompt del usuario convirtiéndola a minúsculas y calcula su *embedding* usando el modelo de *SentenceTransformer*. Finalmente, utiliza el modelo de Random Forest para predecir su clasificación.

#### 4.2.2.1.3 La función `llmClassifier`

Esta función realiza la clasificación del texto en sus cuatro categorías descriptas con anterioridad, utilizando un LLM. Recibe como único parámetro:

- `Query_str[str]`: Prompt el usuario a ser clasificado.

Comienza con el armado del prompt para realizarle la consulta al modelo de LLM. Este prompt contiene indicaciones detalladas sobre cómo realizar el proceso de clasificación e incluye el prompt del usuario a ser clasificado. Luego, se generan los mensajes que serán enviados al modelo: el primero con rol de sistema, para configurar su comportamiento frente al prompt y el segundo con rol de usuario, donde se incluye el prompt mencionado anteriormente. A continuación, se realiza la inferencia del modelo elegido (Qwen2.5-72B-Instruct) usando un cliente que será generado en el entorno de trabajo más adelante y los mensajes generados con anterioridad.

El resultado de la inferencia es retornado a la salida.

#### 4.2.2.1.4 Comparación de modelos

Si bien ambos modelos resuelven con bastante facilidad la tarea de clasificación, al ser entrenado con un dataset no muy grande, el modelo de Random Forest se encuentra subajustado y no consigue resolver prompts más complejos. Esto incluso habiendo mostrado una exactitud del 95% siendo entrenado con un dataset balanceado. Por la flexibilidad y versatilidad que provee el modelo de LLM, además del potencial que tiene para clasificaciones complejas, se optó por usarlo como método principal de clasificación.

### 4.2.3 Hito 3: Creación del retriever

Con el objetivo de extraer información de las bases de datos y conociendo a cuál apuntar, el retriever hace la tarea de “filtrado” de información, refinado de búsqueda y comparación con los contenidos en las bases de datos. Para tal propósito, se crearon diversas funciones, cada una encargada de extraer información de una base de datos específica.

#### 4.2.3.1 La función `vectorialDbRetriever`

Esta función recibe un prompt por parte del usuario y recupera la información contextual asociada en la base de datos vectorial. Recibe como único parámetro:

- `Query[str]`: Prompt el usuario a ser clasificado.

Comienza realizando un *embedding* del prompt del usuario. A continuación, para definir la búsqueda sobre la base de datos vectorial, se usa el mismo modelo de LLM utilizado durante la clasificación para refinar el criterio. Si la clasificación es de “rules”, se procede a armar un prompt que pide categorizarlo en reglas normales, con IA, en solitario y Automa y que por defecto utilizara la clasificación de reglas normales. De forma similar a como se hizo en la clasificación, se le consulta al cliente generado en el entorno de trabajo y se obtiene una respuesta. En caso que la clasificación del clasificador haya sido “review”, no se realiza este proceso de refinado.

Luego, de la colección de *ChromaDB* se filtran los metadatos que contienen la categoría de la pregunta del usuario y se guardan en una lista. De esta manera, se tienen guardados los nombres de los metadatos que coinciden con el criterio de clasificación + refinado.

Con el *embedding* del prompt del usuario y los nombres de los metadatos involucrados, se realiza una *query* a la colección de *ChromaDB*, pidiendo los diez resultados más cercanos a los *embeddings* que contengan los metadatos filtrados. Finalmente, se iteran los documentos obtenidos, agregándolos a una lista y retornándola a la salida.

#### 4.2.3.2 La función *tableDbRetriever*

Esta función recibe un prompt por parte del usuario y recupera la información contextual asociada en la base de datos tabular. Recibe como único parámetro:

- `Query[str]`: Prompt el usuario a ser clasificado.

Comienza extrayendo los nombres de las columnas del *DataFrame* de la base de datos tabular. Nuevamente, se hace uso del modelo de LLM para refinar el criterio de búsqueda, especificándole al modelo que clasifique estrictamente el prompt del usuario en alguno de los nombres de las columnas del *DataFrame*. De esta manera, el cliente devuelve como resultado de la inferencia la columna correcta de la base de datos tabular. Este proceso es importante para poder extraer únicamente la información necesaria de la base de datos.

Finalmente, se retorna a la salida una cadena de texto con el nombre de la columna seleccionada, seguida del contenido de la tabla.

#### 4.2.3.3 La función *graphDbRetriever*

Esta función recibe un prompt por parte del usuario y recupera la información contextual asociada en la base de datos de grafos. Recibe como único parámetro:

- `Query[str]`: Prompt el usuario a ser clasificado.

Comienza listando las relaciones y las entidades (con sus etiquetas). A continuación, se vuelve a hacer uso del modelo de LLM de la misma forma que en los casos anteriores, con la diferencia que en este caso se debió realizar muchas más especificaciones en los mensajes para que funcionara correctamente. En primer lugar, bajo el rol de sistema, se le pide que no solo fuera experto en juegos de mesa, sino que también lo fuera en *Cypher*. Así, se le describe el contenido de la base de datos de grafos con sus entidades y relaciones para

luego pedirle que sea capaz de crear *queries* de *Cypher* usando únicamente los datos provistos. Durante el desarrollo se descubrió que también era necesario darle un ejemplo de cómo debían escribirse dichas *queries* ya que, en ocasiones previas a este agregado, el modelo entregaba *queries* que no tenían sentido y no podían ser procesadas por el grafo. En segundo lugar, bajo el rol de usuario, se le solicita que cree una *query* de *Cypher* basada en el prompt del usuario, respetando no solamente su sintaxis, sino también las entidades y relaciones provistas.

El resultado obtenido por parte del modelo de LLM bajo las especificaciones resulta siempre en una *query* de *Cypher* válida a la cuál se le ajustan cuestiones de texto como saltos de línea y caracteres especiales. Esta *query* es utilizada en el grafo para obtener así los nombres de las entidades pretendidas, que son agregadas a una cadena de texto junto con su *label* de manera de poder identificarlas correctamente, ayudando en los pasos posteriores.

#### 4.2.3.4 La función *retriever*

Esta función recibe un prompt por parte del usuario y llama a las funciones descriptas anteriormente para la recuperación de datos, basado en la clasificación obtenida por el clasificador. Recibe como único parámetro:

- `Query[str]`: Prompt el usuario a ser clasificado.

Comienza identificando la clasificación obtenida por el clasificador para usarla dentro de condicionales. Si la clasificación es “credits” se llama a la función *graphDbRetriever*, si la clasificación es “stats” se llama a la función *tableDbRetriever* y, por último, si la clasificación es “review” o “rules” se llama a la función *vectorialDbRetriever*. Adicionalmente y únicamente para este último caso donde se consulta la base de datos vectorial, los resultados obtenidos se envían a una función llamada *reranker* que será descrita en la siguiente sección.

Finalmente, se retornan los resultados obtenidos de la búsqueda: cadenas de texto con datos contextuales basados en el prompt del usuario.

### 4.2.4 Hito 4: Creación del *reranker*

El re-ranker tiene como objetivo ordenar los resultados obtenidos mediante una consulta a la base de datos vectorial según la relevancia o proximidad al prompt del usuario. De esta manera se busca obtener únicamente los datos contextuales más relevantes en una refinación de la selección. Con este propósito, se desarrolló una función que luego sería utilizada por el *retriever* una vez haya obtenido los datos contextuales de la base de datos vectorial.

#### 4.2.4.1 La función *reranker*

Esta función realiza un proceso de re-ranking sobre los documentos obtenidos tras una búsqueda en la base de datos vectorial. Recibe como parámetros:

- `Query[str]`: Prompt el usuario a ser clasificado.

- `Documents[list]`: Lista con documentos obtenidos de la búsqueda en la base de datos vectorial.
- `Top_k[int]`: Cantidad de documentos a retornar.

Se carga en primera instancia un modelo de codificación cruzada, específicamente “ms-marco-MiniLM-L-6-v2”. Luego, se genera una lista de tuplas, donde se emparejan el prompt del usuario con cada documento recibido como argumento (hasta diez documentos) y se hace una inferencia sobre el modelo, dando como resultado los puntajes de cada emparejamiento. Luego, se emparejan los puntajes con sus documentos respectivos y se ordenan de mayor a menor de acuerdo al puntaje. De esta manera, se extraen aquellos que tienen mayor puntaje, respetando el límite de documentos definidos por el parámetro *Top\_k*.

Finalmente, se crea una cadena de texto con los documentos seleccionados y se retornan a la salida.

#### 4.2.5 Hito 5: Implementación del RAG

Para completar la implementación del RAG, se debe combinar el prompt del usuario (la pregunta que quiere realizar sobre el juego “Rajas de Ganges”) junto con la información contextual obtenida tras las consultas a las bases de datos. La combinación de estos datos es enviada a un modelo de LLM que, habiendo sido instruido específicamente en su comportamiento, devuelva una respuesta a la pregunta basada exclusivamente en los datos contextuales. Con este objetivo en mente, se desarrolló una función para iniciar una conversación basada en el modelo *Zephyr*, terminando de definir todo lo necesario para la construcción del RAG.

##### 4.2.5.1 La función RAG

Realiza una consulta a un modelo de LLM basado en el prompt del usuario y su información contextual obtenida de las consultas a las bases de datos. Recibe como parámetros:

- `Query_str[str]`: Prompt el usuario a ser clasificado.
- `Context_str[str]`: Datos contextuales obtenidos de las bases de datos
- `Api_key[str]`: Clave de la API de Hugging Face.
- `Add_generation_prompt[bool]`: Indica si se debe agregar el prompt que lo generó.
- `Max_new_tokens[int]`: Número máximo de tokens a generar.

De forma similar a como se hizo en casos anteriores, se genera un prompt para el modelo de LLM con rol de usuario, donde se le brinda la información contextual y se le pide que brinde una respuesta a la pregunta del usuario basado únicamente en la información contextual brindada, sin ningún tipo de conocimiento previo. Dentro de los mensajes para el modelo, se especifica bajo el rol de sistema que se debe comportar como un asistente que solo responde con información real y basada en hechos, siendo experto en juegos de mesa y sus características descriptivas.

A continuación, se genera una conversación basada en el modelo *Zephyr* usando los mensajes generados previamente para posteriormente formatearlo de la forma que el modelo lo requiere.

Se genera el header para la *API* de Hugging Face, solicitando el uso del modelo “zephyr-7b-beta” y se arman los datos para ser enviados en una solicitud *POST*. Estos datos incluyen el prompt formateado para *Zephyr* y una serie de parámetros que definen cosas como la cantidad de parámetros máxima y el nivel de creatividad.

Finalmente, se hace la solicitud *POST* a la *API*, obteniendo una respuesta que es formateada y entrada como una cadena de texto a su salida.

#### 4.2.6 Hito 6: Programa principal para el primer algoritmo (Main)

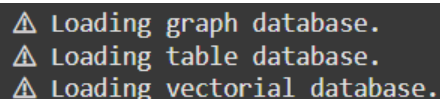
El programa principal integra todas las funciones y bases de datos descritas anteriormente, conectando los componentes individuales de forma ordenada para formar el algoritmo completo.

Comienza con la creación de la base de datos de grafos, donde mediante un *try-except* evalúa si ya existe dentro del entorno de trabajo. De no existir, crea un cliente de *Redis* y en él crea un grafo de *RedisGraph* con el nombre “credits”, haciendo referencia a su contenido. A continuación, ejecuta la función *createGraph* descrita en una sección anterior haciendo uso del grafo creado en el entorno de trabajo y finaliza realizando un *commit* al cliente.

De manera similar, se comprueba la existencia de la base de datos tabular en el entorno de trabajo, caso contrario se ejecuta la función *createTables* para crear el *DataFrame* con el contenido de los “stats” del juego.

Para finalizar la carga de bases de datos, se evalúa la existencia en el entorno de trabajo de la colección de *ChromaDB*, la base de datos vectorial. De no existir, se carga un modelo de *embeddings*, específicamente “universal-sentence-encoder-multilingual/3” y se genera un cliente de *ChromaDB*. Tanto el modelo como el cliente son usados internamente por la función llamada para finalizar la creación, *createCollection*, generando la colección de *ChromaDB* en el entorno de trabajo.

Mediante impresiones en pantalla, se le informa al usuario el estado de carga de las bases de datos al entorno de trabajo.



```
△ Loading graph database.  
△ Loading table database.  
△ Loading vectorial database.
```

Figura 2: Cuando necesita cargar las bases de datos.

```
✓ Graph database loaded.
✓ Table data base loaded.
✓ Vectorial data base loaded.
```

Figura 3: Cuando las bases de datos ya están cargadas.

A continuación, se imprime un mensaje en pantalla pidiéndole al usuario que ingrese su pregunta sobre el juego.

```
////////////////////
Welcome to the Rajas of the Ganges chatbot!

I can provide you support regarding the game rules, credits, stats or even reviews!

How can I help you today?
Question: 
```

Figura 4: Pregunta del chatbot al usuario.

Ingresado el prompt, se crea el cliente de inferencia de LLM utilizado para todas las tareas de clasificación dentro del algoritmo basado en *Qwen*, el cual es consultado via *API* a la página de Hugging Face. Luego, se llama a la función *llmClassifier* que realiza la clasificación de la pregunta del usuario y la almacena en el entorno de trabajo.

A continuación, se llama a la función *retriever*, que usa todos los recursos generados en el entorno de trabajo más el prompt del usuario para obtener los datos contextuales y guardarlos en una variable global.

Finalmente, se llama a la función *RAG*, encargada de tomar el prompt del usuario, los datos contextuales y consultar al modelo de LLM basado en *Zephyr* para generar una respuesta y mostrarla en pantalla.

```
✓ Graph database loaded.
✓ Table data base loaded.
✓ Vectorial data base loaded.

////////////////////
Welcome to the Rajas of the Ganges chatbot!

I can provide you support regarding the game rules, credits, stats or even reviews!

How can I help you today?
Question: What's the maximum playtime?

-----

Here's what I found!

The maximum playing time, as provided in the contextual information, is 75 minutes.
```

Figura 5: Ejemplo de resultado final.

### 4.3 RESULTADOS Y CONCLUSIONES

El algoritmo responde sorprendentemente bien a las preguntas planteadas por el usuario. Se destaca el potencial de los modelos de LLM no únicamente para la generación de respuestas, sino también para las tareas de clasificación y refinamiento de criterios de búsqueda realizadas por el algoritmo. Esto se debe a un cuidadoso trabajo en la redacción de los prompts para los modelos, donde fue clave la especificidad a la hora de imponerle la forma en la que debían trabajar. Fue en estos lugares donde se tuvo que invertir bastante tiempo, siendo el mejor ejemplo el prompt generado para la función *graphDbRetriever* donde fue necesario inclusive brindarle un ejemplo concreto de lo que se esperaba como su respuesta para obtener un buen desempeño.

A pesar de lo mencionado anteriormente, sí se han detectado situaciones en donde las respuestas basadas en contexto de la base de datos vectorial no respondían exactamente lo que se esperaba. Una posible solución a esto sería ajustar el método para particionar el texto en categorías aún más específicas y detallándolas en los metadatos. Otra posible mejora para esto se podría aplicar en el re-ranker, quizá usando otro modelo de codificación cruzada.

Otro comportamiento curioso que se detectó durante los ensayos del algoritmo, fue el “agotamiento” del modelo basado en *Qwen*, que tras muchas pruebas seguidas afirmó “estar ocupado” y que intentara consultarle nuevamente más tarde.

Finalmente, se concluyen tres aspectos importantes:

- Al apoyarse exclusivamente en la información contextual provista, el RAG necesita que las bases de datos estén con la información más completa y ordenada posible para poder responder de forma precisa y correcta. Se deben invertir muchas horas de trabajo en el refinado de estas bases de datos con el objetivo claro de poder responder a todas las preguntas que pudieran surgir.
- A pesar de lo mencionado en el punto anterior, los modelos de LLM son tan poderosos que logran compensar la falta o la desprolijidad en la información con su “creatividad”, logrando cubrir muchas de las falencias que podrían tener inherentemente las bases de datos.
- Usar LLMs termina por ser una opción más atractiva que generar modelos propios, ya que no requieren de corpus inmensos, tiempos de entrenamiento enormes y ahora ni siquiera de poder de procesamiento, ya que se encuentran entrenados y se pueden usar en la nube mediante *APIs*.

### 4.4 ENLACES A MODELOS Y LIBRERÍAS UTILIZADAS

#### 4.4.1 Enlaces a librerías

- **NetworkX:** <https://networkx.org/>
- **Matplotlib:** <https://matplotlib.org/>

- **Redis:** <https://redis.io/>
- **RedisGraph:** [invalid URL removed]
- **TensorFlow Text:** [https://www.tensorflow.org/text/guide/tf\\_text\\_intro](https://www.tensorflow.org/text/guide/tf_text_intro)
- **TensorFlow Hub:** <https://www.tensorflow.org/hub/>
- **Pandas:** <https://pandas.pydata.org/>
- **NumPy:** <https://numpy.org/>
- **Scikit-learn:** <https://scikit-learn.org/stable/>
- **LangChain:** <https://www.langchain.com/>
- **Sentence Transformers:** <https://www.sbert.net/>
- **YouTube-Transcript-API:** <https://www.transcribetube.com/youtube-transcript-api>
- **PyPDF2:** <https://pypdf2.readthedocs.io/>
- **Requests:** <https://requests.readthedocs.io/en/latest/>
- **Beautiful Soup 4:** <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>
- **GDown:** <https://github.com/wkentaro/gdown>
- **ChromaDB:** <https://www.chromadb.com/>
- **LLM-Templates:** [https://github.com/jpmanson/llm\\_templates](https://github.com/jpmanson/llm_templates)
- **Joblib:** <https://joblib.readthedocs.io/en/latest/>
- **Warnings:** <https://docs.python.org/3/library/warnings.html>
- **Time:** <https://docs.python.org/3/library/time.html>
- **Re:** <https://docs.python.org/3/library/re.html>
- **Json:** <https://docs.python.org/3/library/json.html>
- **Hugging Face Hub:** <https://huggingface.co/docs/hub/index>
- **Python-Decouple:** <https://pypi.org/project/python-decouple/>

## 4.4.2 Enlaces a modelos

- **universal-sentence-encoder-multilingual:**  
[https://www.tensorflow.org/hub/tutorials/cross\\_lingual\\_similarity\\_with\\_tf\\_hub\\_multilingual\\_universal\\_encoder](https://www.tensorflow.org/hub/tutorials/cross_lingual_similarity_with_tf_hub_multilingual_universal_encoder)
- **distiluse-base-multilingual-cased-v1:** <https://huggingface.co/distilbert-base-multilingual-cased>
- **Qwen/Qwen2.5-72B-Instruct:** <https://huggingface.co/Qwen/Qwen2.5-72B-Instruct>
- **zephyr-7b-beta:** <https://huggingface.co/HuggingFaceH4/zephyr-7b-beta>



## 5. EJERCICIO 2: AGENTE

### 5.1 RESUMEN

Basándose en el ejercicio anterior, se debe incorporar el concepto de Agente usando el concepto ReAct. El agente debe cumplir con los siguientes requisitos:

- Utilizar al menos tres herramientas aprovechando el trabajo anterior:
  - `Doc_search()`: busca información en los documentos. En el caso de este trabajo, se utilizará la función *vectorialDbRetriever*.
  - `Graph_search()`: busca información en la base de datos de grafos. En el caso de este trabajo, se utilizará la función *graphDbRetriever*.
  - `Table_search()`: busca información sobre los datos tabulares. En el caso de este trabajo, se utilizará la función *tableDbRetriever*.
- Se puede implementar alguna nueva herramienta que considere necesaria y que pueda enriquecer las capacidades del agente.
- Utilizar la librería Llama-Index para desarrollar el agente:
  - `Llama_index.core.agent.ReActAgent`
  - `Llama_index.core.tools.FunctionTool`
- Se debe construir el prompt adecuado para incorporar las herramientas al agente ReAct.

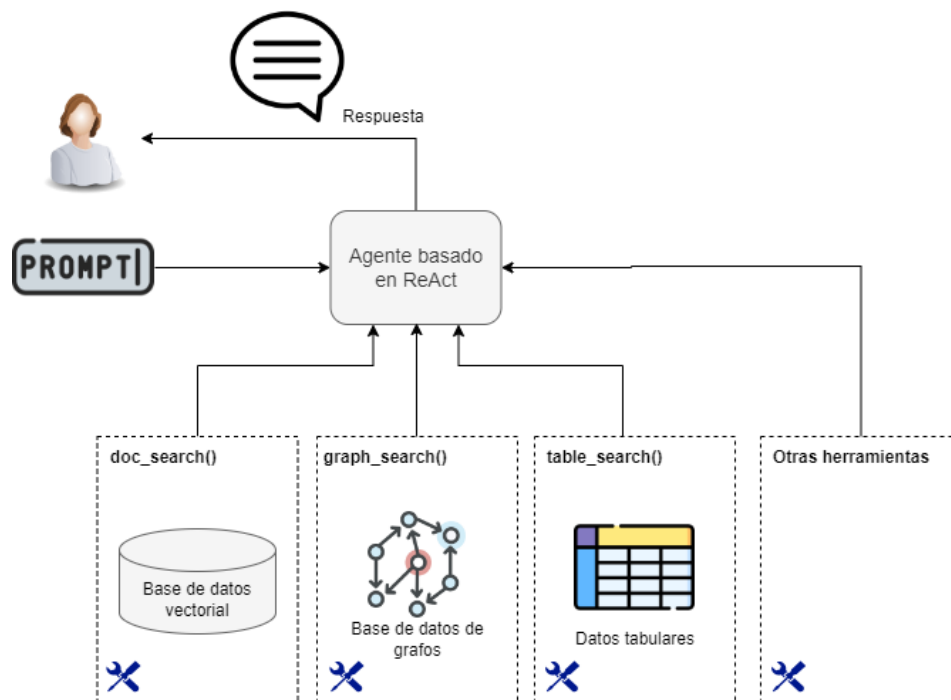


Figura 6: Esquema de Agente utilizado.

En el informe se presentan los resultados:

- Cinco ejemplos de prompts donde se deba recurrir a más de una herramienta para responder al usuario, evaluando los resultados obtenidos.
- Tres ejemplos explicados, donde el agente falla o las respuestas no son precisas.
- Explicar cuáles son las mejoras que sería conveniente realizar para mejorar los resultados.

## 5.2 DESARROLLO E IMPLEMENTACIÓN

Se planteó nuevamente un desarrollo modular para la resolución en este ejercicio. Afortunadamente, se podrían reutilizar la mayoría de las funciones creadas para el ejercicio anterior, por lo que fue necesario desarrollar únicamente dos nuevas secciones: una función para el Agente y un nuevo código Main. Las funciones a reutilizar serían las de generación de bases de datos y los retrievers mencionados en la sección anterior, los cuales serían llamados por el Agente mismo.

El rol del agente es reconocer de forma inteligente cuál de las funciones utilizar de acuerdo al prompt del usuario recibido, interpretar los resultados obtenidos por las bases de datos y desarrollar una respuesta. Caso que no pueda hacerlo, probar con otras funciones y repetir durante una cierta cantidad de iteraciones, entregando finalmente un resultado o una falla.

### 5.2.1 Hito 7: Implementación del Agente

Para este propósito, se decidió centralizar todo en una única función que consolidara la creación del agente como tal y que también sirviera como interfaz para el usuario.

#### 5.2.1.1 La función agent

Esta función se encarga de ejecutar el Agente de principio a fin. Comienza configurando el LLM de Ollama con una temperatura baja para respuestas más deterministas.

A continuación, se definen las herramientas que usará el Agente, donde se incluyen las funciones *vectorialDbRetriever*, *tableDbRetriever* y *graphDbRetriever* junto con una descripción detallada sobre cuándo usar cada una de ellas. Esta descripción ayuda al Agente a entender por cuál función comenzar a buscar información.

Definidos el modelo y las herramientas, se puede finalmente definir el Agente. Además de los dos elementos mencionados anteriormente, se le debe definir el formato del chat (en este caso usa el de ReAct) y el prompt de sistema (del rol de sistema), donde se define el comportamiento que debe tener de forma muy detallada. En el prompt se le definen las siguientes características:

- *Comportamiento*: se le indica que es un asistente útil y que solo responde con información certera. También se le pide que sea un experto en juegos de mesa y que

entienda sus características descriptivas. Finalmente, se le pide que siga estrictamente el formato que se describe en los siguientes ítems.

- *Cadena de pensamiento*: se le pide que primero presente lo que necesita hacer, que luego mencione la herramienta que utilizará y que termine exponiendo el prompt del usuario.
- *Ejemplos*: se le explica con tres ejemplos cómo parametrizar correctamente cada función de su conjunto de herramientas.
- *Observación*: se le pide que presente el resultado de la herramienta y que repita el proceso de ser necesario.
- *Respuesta final*: se le indica que combine todos los resultados obtenidos en su respuesta final.
- *Qué hacer con cada query*: se le dice que en primer lugar analice la información, que luego utilice las herramientas con el formato correcto y que finalmente combine los resultados obtenidos para la respuesta final.
- *Aclaraciones importantes*: se le exige que siga siempre el formato especificado para cada herramienta, que explore todas las herramientas y que cuando encuentre una respuesta, que asuma que es la correcta.

También se le configura que no guarde historia, para no sesgarse.

Configurado el Agente, se le pide al usuario que ingrese una pregunta. Mediante un *try-except* se evalúa si la pregunta está vacía, caso contrario se inicia un chat con el agente entregándole el prompt del usuario.

Finalmente, se imprime en pantalla la cadena de pensamientos del Agente.

### 5.2.2 Hito 8: Programa principal para el segundo algoritmo (Main)

La sección principal del programa es, en gran medida, idéntica a la del ejercicio anterior, manteniendo todo lo que es la creación de las tres bases de datos posterior a la verificación de su existencia en el entorno de trabajo.

Cargadas las bases de datos, se inicializa el cliente de LLM que utilizan las funciones del retriever, se le da la bienvenida al usuario al chatbot y se ejecuta la función *agent*, comenzando con el proceso de pedido de prompt al usuario y su posterior análisis por el agente.

## 5.3 RESULTADOS Y CONCLUSIONES

El Agente posee un comportamiento errático, en ocasiones consiguiendo obtener las respuestas correctas y en ocasiones no. Incluso se han presentado situaciones en donde no pudo responder una pregunta que anteriormente sí había conseguido encontrarle la respuesta correcta.

Se observa también que no siempre toma las decisiones correctas sobre cuál herramienta utilizar, buscando las respuestas en bases de datos que no se corresponden con la categoría de la pregunta. Es también usual ver al Agente alucinando, consiguiendo la respuesta correcta en su primera consulta para luego seguir buscando con otras herramientas, pensando que no es lo que necesitaba.

A continuación, se presentan diferentes ejemplos del comportamiento del Agente y luego una breve reseña sobre posibles mejoras.

## 5.3.1 Uso de múltiples herramientas

Se presentan cinco situaciones en donde necesitó recurrir a más de una herramienta para obtener un resultado.

### 5.3.1.1 Caso 1

En el siguiente ejemplo, se le pidió al chatbot que buscara información acerca de los diseñadores del juego.

```

////////////////////////////////////
Welcome to the Rajas of the Ganges chatbot!

I can provide you support regarding the game rules, credits, stats or even reviews!

How can I help you today?

Query: Who are the designers of the game?
> Running step 00204482-d583-4392-8009-d54962a08c25. Step input: Who are the designers of the game?
Thought: The current language of the user is: English. I need to use a tool to help me answer the question.
Action: tableDbRetriever
Action Input: {'query': 'designers of the game'}
Observation: Game: Rajas of the Ganges
> Running step b5a7f683-0393-4eec-a4d2-fd2f2c37b0f2. Step input: None
Thought: The output from the previous tool was not sufficient to answer the question. I need to use another tool to get more information.
Action: vectorialDbRetriever
Action Input: {'query': 'Rajas of the Ganges designers'}
Observation: Error: Expected where operand value to be a non-empty list, and all values to be of the same type got [] in query.
> Running step 58db6134-0d6b-4325-815b-cba24d6e48fa. Step input: None
Thought: The tool output indicates an error due to an empty list. I need to rephrase the query to get accurate results.
Action: vectorialDbRetriever
Action Input: {'query': 'designers of Rajas of the Ganges'}
Observation: Error: Expected where operand value to be a non-empty list, and all values to be of the same type got [] in query.
> Running step 11abe4d1-23fd-42ea-ba8d-d76df85e4f76. Step input: None
Thought: The tool output still indicates an error due to an empty list. I need to rephrase the query again to get accurate results.
Action: graphDbRetriever
Action Input: {'query': 'Rajas of the Ganges game designers'}
Observation: Designer: Inka Brand, Designer: Markus Brand
> Running step 5d0049d2-b242-42a4-82c3-88414d9645b6. Step input: None
Thought: The tool output provided information about the designers of the game. I can now answer the user's question.
Answer: The designers of the game "Rajas of the Ganges" are Inka Brand and Markus Brand.

```

Figura 7: Múltiples herramientas - Caso 1

Se observa que en primer lugar intentó buscar en la base de datos tabular y luego realizó dos consultas a la base de datos vectorial hasta que finalmente decidió probar con la base de datos de grafos. Sus primeros intentos fallaron ya que no era la ubicación real de la información necesitada.

### 5.3.1.2 Caso 2

En el siguiente ejemplo, se le pidió al chatbot que buscara información acerca de cómo preparar el juego.

```

Query: How to setup the game?
> Running Step 2e2ba2dc-d6e5-4709-4097-aab8ed3a67e6. Step Input: How to setup the game?
Thought: The current language of the user is: English. I need to use a tool to help me answer the question.
Action: tabledndriver
Action Input: ["query": "setup instructions for the game"]
Observation: ["<h3> Rajas of the Ganges </h3>"]
> Running Step 49053f2a-e34f-4b10-8809-7359408ac20. Step Input: None
Thought: The output from the previous tool was not sufficient, so I'll use another tool to gather more information.
Action: vectorialdriver
Action Input: ["query": "Rajas of the Ganges setup instructions"]
stats
[[{"0.8567010251045227", "0.01842122685658538", "-0.011626442894339561", "0.087074682116560848", "0.02846371755003929", "-0.0328780226409433", "-0.00781695730984211", "0.039727483853883444", "0.0089836654680896731", "-0.06662389636039734", "-0.02844840402"}]
Observation: ["<h3> Expected to appear within 1 to n non-empty list, and all values to be of the same type got 0 in query. </h3>"]
> Running Step 51291af4-d104-4098-b0d3-8c2a78af5f8c. Stop Input: none
Thought: The tool output indicates that the query was not properly formatted, so I'll try rephrasing it.
Action: tabledndriver
Action Input: ["query": "What are the setup instructions for Rajas of the Ganges"]
Observation: ["Error: Game\\nIn the setup instructions for Rajas of the Ganges are as follows:\\n\\n1. **Prepare the Board**:\\nUnfold and place the game board in the center of the play area.\\n2. **Place Starting Tile**:\\nRandomly select a starting tile (the Ganges River tile) on the board. Place it in the designated starting area.\\n3. Set up the Money Supply:\\nPlace the money supply within easy reach of all players.\\n4. Prepare the Cards:\\nShuffle the action cards and deal a hand to each player based on the number of players (3 cards for 2-3 players, 4 cards for 4-5 players). Place the remaining cards face-down to form a draw pile.\\n5. Place River Tiles:\\nSort the river tiles by their numbers and place them face-down in separate stacks within reach of all players.\\n6. Distribute Player Boards and Tokens:\\nEach player receives a player board, a set of boats, and a set of houses in their chosen color.\\n7. Initial Placement of Boats:\\nEach player places one boat on the starting tile.\\n8. Prepare the Victory Points Tokens:\\nPlace the victory points tokens within reach of all players.\\n\\nAfter completing these steps, the game is ready to begin. The starting player is determined by the player who placed the starting tile, and play proceeds clockwise."\\n"]
> Running Step 051744da-3c12-4d65-a317-c7e65a5f1f2d. Step Input: None
Thought: I have gathered enough information about the setup instructions for Rajas of the Ganges.
Answer: To set up the game, follow these steps:
1. Prepare the Board: Unfold and place the game board in the center of the play area.
2. Place Starting Tile: Randomly select a starting player who will place the starting tile (the Ganges River tile) on the board. Place it in the designated starting area.
3. Set up the Money Supply: Place the money supply within easy reach of all players.
4. Prepare the Cards: Shuffle the action cards and deal a hand to each player based on the number of players (3 cards for 2-3 players, 4 cards for 4-5 players). Place the remaining cards face-down to form a draw pile.
5. Place River Tiles: Sort the river tiles by their numbers and place them face-down in separate stacks within reach of all players.
6. Distribute Player Boards and Tokens: Each player receives a player board, a set of boats, and a set of houses in their chosen color.
7. Initial Placement of Boats: Each player places one boat on the starting tile.
8. Prepare the Victory Points Tokens: Place the victory points tokens within reach of all players.
After completing these steps, the game is ready to begin. The starting player is determined by the player who placed the starting tile, and play proceeds clockwise.

Answer: To set up the game, follow these steps:
1. Prepare the Board: Unfold and place the game board in the center of the play area.
2. Place Starting Tile: Randomly select a starting player who will place the starting tile (the Ganges River tile) on the board. Place it in the designated starting area.
3. Set up the Money Supply: Place the money supply within easy reach of all players.
4. Prepare the Cards: Shuffle the action cards and deal a hand to each player based on the number of players (3 cards for 2-3 players, 4 cards for 4-5 players). Place the remaining cards face-down to form a draw pile.
5. Place River Tiles: Sort the river tiles by their numbers and place them face-down in separate stacks within reach of all players.
6. Distribute Player Boards and Tokens: Each player receives a player board, a set of boats, and a set of houses in their chosen color.
7. Initial Placement of Boats: Each player places one boat on the starting tile.
8. Prepare the Victory Points Tokens: Place the victory points tokens within reach of all players.
After completing these steps, the game is ready to begin. The starting player is determined by the player who placed the starting tile, and play proceeds clockwise.

```

Figura 8: Múltiples herramientas - Caso 2

Aquí el Agente comenzó buscando en la base de datos tabular, finalmente encontrando su respuesta al hacer la consulta a la base de datos vectorial, la ubicación correcta de la información.

### 5.3.1.3 Caso 3

En el siguiente ejemplo, se le pidió al chatbot que buscara información acerca de la duración máxima del juego.

```

#####

Welcome to the Rajas of the Ganges chatbot!

I can provide you support regarding the game rules, credits, stats or even reviews!

How can I help you today?

Query: What's the maximum playtime?
> Running step 4c7ae42c-e8fe-4d66-9d78-53c34af3c200. Step input: What's the maximum playtime?
Thought: The current language of the user is: English. I need to use a tool to help me answer the question.
Action: talishibottriever
Action input: {'query': 'maximum playtime'}
Observation: Max. Playing Time (minutes): 75
> Running step bf1ae338-012a-484b-99eb-86eb3beccd7b. Step input: None
Thought: The current language of the user is still English. I need to use another tool to help me answer the question.
Action: grapgdbbottriever
Action input: {'query': 'maximum playtime'}
Observation: Error: error: Invalid input 'P': expected a label, '(', a parameter or ') Line: 3, column: 96, offset: 95 errctx: ...HAS_MAX_PLAYTIME]->(maxPlayTime: Max Play Time)RETURN max playtime,time No... errctxoffset: 40
> Running step 17c15d9f-703c-4a4e-a881-1aa7f1b2eac2. Step input: None
Thought: The tool grapgdbbottriever seems to have returned an error. I'll try again with a different tool or rephrase the query.
Action: talishibottriever
Action input: {'query': 'longest playing time'}
Observation: Max. Playing Time (minutes): 75
> Running step a1e435b-62e1-4241-ba91-701942528db6. Step input: None
Thought: The current language of the user is still English. I can answer without using any more tools. I'll use the user's language to answer.
Answer: The maximum playtime for this game is 75 minutes.

```

*Figura 9: Múltiples herramientas - Caso 3*

En este caso, tuvo un comportamiento extraño. Si bien halló la respuesta en su primer intento, consideró que no tenía información suficiente y la fue a buscar a la base de datos de grafos. Como no la encontró allí, volvió a intentar en la base de datos tabular, la ubicación correcta de la información, decidiéndose finalmente por presentar esa información como la correcta.

#### 5.3.1.4 Caso 4

En el siguiente ejemplo, se le pidió al chatbot que buscara información acerca de la cantidad de gente que posee el juego.



```

////////////////////////////////////
Welcome to the Rajas of the Ganges chatbot!

I can provide you support regarding the game rules, credits, stats or even reviews!

How can I help you today?

Query: How many people own the game?
> Running step 4f7a7597-8ae7-4b48-a9a4-d66d84d9736c. Step input: How many people own the game?
Thought: The current language of the user is: English. I need to use a tool to help me answer the question.
Action: tableDBRetriever
Action Input: {'query': 'number of owners of the game'}
Observation: Own: 19513
> Running step 0bbb695f-2492-4ea3-936f-99b1b477d3ab. Step input: None
Observation: Error: Could not parse output. Please follow the thought-action-input format. Try again.
> Running step 1d3a6b91-3d22-4f3a-9520-b68a385efc0a. Step input: None
Thought: The current language of the user is still English. I need to use a tool to help me answer the question.
Action: tableDBRetriever
Action Input: {'query': 'number of owners of the game'}
Observation: Own: 19513
> Running step d65b9254-47c2-4e70-8a86-68dc27cb3824. Step input: None
Thought: The output from the tableDBRetriever indicates that the number of owners of the game is 19,513. I can use this information to answer the user's question.
Answer: The game has 19,513 owners.

Answer: The game has 19,513 owners.

```

Figura 10: Múltiples herramientas - Caso 4.

Aquí se presenta otra situación extraña, donde si bien logra dar con la respuesta correcta tuvo que consultar dos veces a la misma herramienta. Por los mensajes, se entiende que se debe a que no consiguió *parsear* la salida.

### 5.3.1.5 Caso 5

En el siguiente ejemplo, se le pidió al chatbot que buscara reseñas del juego.

```

////////////////////////////////////
Welcome to the Rajas of the Ganges chatbot!

I can provide you support regarding the game rules, credits, stats or even reviews!

How can I help you today?

Query: What does the reviews say about the game?
> Running step 3617b732-79e7-4a07-bfcc-9ef935664107. Step input: What does the reviews say about the game?
Thought: The current language of the user is: English. I need to use a tool to help me answer the question.
Action: vectorialDBRetriever
Action Input: {'query': 'AttributeDict[{'title': 'Reviews'}, {'type': 'string'}]}
Observation: Error: Binding inputs to if function failed due to 'Attempt to convert a value (AttributeDict[{'title': 'Reviews'}, {'type': 'string'}]) with an unsupported type (class 'dirtyjson.attributed_containers.AttributeDict)'
> Running step 158242b-8ada-4bb0-a507-b60e7492aa50. Step input: None
Thought: The tool vectorialDBRetriever encountered an error while processing the input. I'll try to provide a different format for the query.
Action: vectorialDBRetriever
Action Input: {'query': 'What does the reviews say about the game?'}
Observation: Error: Expected where operand value to be a non-empty list, and all values to be of the same type got [] in query.
> Running step fea0c9b8-da5c-41ec-b400-c540f6e78c0a. Step input: None
Thought: The tool vectorialDBRetriever is still not able to process the input correctly. I'll try breaking down the query into smaller parts.
Action: vectorialDBRetriever
Action Input: {'query': 'What does the reviews say about the game?'}
Observation: Error: Expected where operand value to be a non-empty list, and all values to be of the same type got [] in query.
> Running step 00e0c450-6ab0-4ba1-b006-306a3cab535. Step input: None
Thought: The tool vectorialDBRetriever is still not able to process the input correctly. I'll try providing more context.
Action: tableDBRetriever
Action Input: {'query': 'What does the reviews say about the game?'}
Observation: Comments: 2.12
> Running step 8ce0ff77-5784-41a2-99d2-e124c83e54e. Step input: None
Thought: The tool tableDBRetriever was able to retrieve some information, but it seems limited in its response. I'll try to gather more information from other tools.
Action: graphDBRetriever
Action Input: {'query': 'What does the reviews say about the game?'}
Observation:
Answer: Error processing the query: Reached max iterations.

```

Figura 11: Múltiples herramientas - Caso 5.

Aquí no consiguió obtener una respuesta. Si bien intentó obtenerlas a través de la base de datos vectorial no consiguió obtener resultados, posiblemente porque no realizó el suficiente tiempo de espera. A continuación, intenta buscar la información en las otras dos bases de datos y como no las consiguió antes de llegar a la cantidad máxima de iteraciones, se dio por vencido.

### 5.3.2 Situaciones de falla

Se presentan tres situaciones en donde el Agente falló al entregar una respuesta correcta.

5.3.2.1 Caso 1

En el siguiente ejemplo, se le pidió al chatbot que buscara información acerca del peso (dificultad) del juego.

```
#####
Welcome to the Rajas of the Ganges chatbot!
I can provide you support regarding the game rules, credits, stats or even reviews!
How can I help you today?

Query: What's the weight of the game?
> Running step 3d566e21-e1e5-49de-bfd1-bb8b4afdefe. Step input: what's the weight of the game?
Thought: The current language of the user is: English. I need to use a tool to help me answer the question.
Action: factoidRetriever
Action Input: {'query': 'weight of the game'}
Observation: Weight: 2.89 / 5
> Running step d6d419c8-3cc0-422b-bfab-3b7b9ce69252. Step input: None
Thought: The user has provided information about the weight of the game, but it's not a direct answer to my question. I need more context or information from another tool.
Action: vectorialDbRetriever
Action Input: {'query': 'weight of the game'}
Observation: Error: Expected where operand value to be a non-empty list, and all values to be of the same type got [] in query.
> Running step 6f883cd4-7c21-4470-8f07-13b9340c2a14. Step input: None
Thought: The tool returned an error message indicating that the input was not in the correct format. I need to reformat the input to match the expected structure.
Action: vectorialDbRetriever
Action Input: {'query': 'weight of game'}
Observation: Error: Expected where operand value to be a non-empty list, and all values to be of the same type got [] in query.
> Running step ebbec25-375d-4bc2-8f6d-f23d97780ec. Step input: None
Thought: The tool is still returning an error message despite reformatting the input. I need to try a different approach or use a different tool.
Action: graphDbRetriever
Action Input: {'query': 'weight of game'}
Observation: Error: errMsg: Invalid input 'T': expected '(', '!', a statement option, a query hint, a clause or a schema command line: 1, column: 1, offset: 0 errMsg: The question "weight of game" does not directly correspond to any
> Running step b71c8239-d1bf-460b-8836-bfff96ae5dd. Step input: None
Thought: The tool is still returning an error message, and it seems that the input 'weight of game' is not a valid query for this tool. I need to try a different approach or use a different tool.
Action: graphDbRetriever
Action Input: {'query': 'game weight'}
Observation: Error: errMsg: Invalid input 'I': expected SET or START line: 1, column: 2, offset: 1 errMsg: Since the game weight is not directly provided in the given entities and rela... errMsgOffset: 1
Answer: Error processing the query: Reached max iterations.
#####
```

Figura 12: Situaciones de falla - Caso 1.

Aquí se observa que el Agente alucina, encontrando la respuesta correcta en su primer intento, pero creyendo que aún no tiene la información necesaria. Esto lo lleva a probar otras herramientas hasta alcanzar su límite máximo de iteraciones.

5.3.2.2 Caso 2

En el siguiente ejemplo, se le pidió al chatbot que buscara información acerca de las categorías del juego.

```
#####
Welcome to the Rajas of the Ganges chatbot!
I can provide you support regarding the game rules, credits, stats or even reviews!
How can I help you today?

Query: What are the categories of the game?
> Running step e9fa58fc-0266-4406-824d-2e35816578b0. Step input: What are the categories of the game?
Thought: The current language of the user is: English. I need to use a tool to help me answer the question.
Action: graphDbRetriever
Action Input: {'query': 'AttributedDict({'title': 'Categories'}, {'type': 'string'})}]}}
Observation: Category: Dice, Category: Economic, Category: Renaissance, Category: Territory Building
> Running step 13b64ac1-90c2-45ac-b040-66ed3752c501. Step input: None
Thought: I have obtained the categories of the game. Now I need to gather more information about the game.
Action: vectorialDbRetriever
Action Input: {'query': 'Categories: Dice, Category: Economic, Category: Renaissance, Category: Territory Building'}
Observation: Error: Expected where operand value to be a non-empty list, and all values to be of the same type got [] in query.
> Running step 18114114-bc09-497b-8aea-3b261150127f. Step input: None
Thought: The tool returned an error because the input was not formatted correctly. I need to reformat the input to match the expected format.
Action: vectorialDbRetriever
Action Input: {'query': 'Categories: Dice, Category: Economic, Category: Renaissance, Category: Territory Building'}
Observation: Error: Expected where operand value to be a non-empty list, and all values to be of the same type got [] in query.
> Running step d3b32e1-696d-454a-9da7-023881c0f436. Step input: None
Thought: The tool still returned an error because the input was not formatted correctly. I need to reformat the input to match the expected format.
Action: vectorialDbRetriever
Action Input: {'query': 'Categories: Dice, Category: Economic, Category: Renaissance, Category: Territory Building', 'properties': 'AttributedDict({'query': 'AttributedDict({'title': 'Categories'}, {'type': 'string'})}]}})}
Observation: Error: vectorialDbRetriever got an unexpected keyword argument 'properties'
> Running step 181dcdcb-bd4f-4207-958a-375ae161c053. Step input: None
Thought: The tool returned another error because the input was not formatted correctly. I need to reformat the input to match the expected format.
Action: factoidRetriever
Action Input: {'query': 'Categories: Dice, Category: Economic, Category: Renaissance, Category: Territory Building'}
Observation: Game: Rajas of the Ganges
Answer: Error processing the query: Reached max iterations.
#####
```

Figura 13: Situaciones de falla - Caso 2.

Una situación con las mismas características que la anterior. Si bien obtiene las respuestas correctas en su primer intento, alucina y piensa que aún no tiene lo que necesita, explorando otras herramientas hasta alcanzar su límite de iteraciones.

5.3.2.3 Caso 3

En el siguiente ejemplo, se le pidió al chatbot que buscara información acerca de cómo preparar el juego.

```
#####
Welcome to the Rajas of the Ganges chatbot!
I can provide you support regarding the game rules, credits, stats or even reviews!
How can I help you today?

Query: How to setup the game
> Running step 596281ba-1387-4163-8243-0834e1e57a68. Step Input: How to setup the game
Thought: The current language of the user is: English. I need to use a tool to help me answer the question.
Action: tabularRetriever
Action Input: {'query': 'setup instructions for [game name]'}
Observation: Game: Rajas of the Ganges
> Running step cd60e893-1c52-48d8-86b8-29cf57f11e6d. Step Input: None
Thought: The user has provided more context. I can use this information to refine my search query.
Action: tabularRetriever
Action Input: {'query': 'Rajas of the Ganges setup instructions'}
Observation: Error: "Chess is a strategy board game played between two players on a checkered gameboard with 64 squares arranged in an 8x8 grid. The game is believed to have originated in India, and it has a rich history dating back to the 6th century."
> Running step eb4e4af0-358c-41a7-a6a3-f60b2727fb08. Step Input: None
Thought: The tool output does not seem relevant to the question about setting up Rajas of the Ganges. I need to try again with a different tool.
Action: vectorialRetriever
Action Input: {'query': 'Rajas of the Ganges setup instructions'}
Observation: Error: Expected where operand value to be a non-empty list, and all values to be of the same type got [] in query.
> Running step f5d1f123-913c-44be-82fc-3843e6430fe9. Step Input: None
Thought: The tool output indicates that the query is not properly formatted. I need to rephrase the question to make it more specific.
Action: graphdbRetriever
Action Input: {'query': 'Game mechanics and setup requirements for Rajas of the Ganges'}
Observation: Mechanism: Connections, Mechanism: Dice Rolling, Mechanism: Race, Mechanism: Tile Placement, Mechanism: Worker Placement, Mechanism: Worker Placement with Dice Workers
> Running step 51ed466b-9e48-4474-9d5e-8cbad86c59d8. Step Input: None
Thought: The tool output provides a list of game mechanics involved in Rajas of the Ganges. I can use this information to provide more specific setup instructions.
Action: graphdbRetriever
Action Input: {'query': 'Rajas of the Ganges setup requirements'}
Observation: Error: errMsg: Invalid input 'o': expected DETACH DELETE, DELETE, DROP INDEX ON or DROP CONSTRAINT ON line: 1, column: 2, offset: 1 errMsgs: does not directly support a concept of "setup requirements" as a specific rel...
Answer: Error processing the query: Reached max iterations.
#####
```

Figura 14: Situaciones de falla - Caso 3.

Aquí comienza buscando en la base de datos tabular, acción que intenta dos veces y fracasa ya que la información no se encuentra allí. En su tercer intento, busca información en la base de datos vectorial pero no consigue extraerla, presumiblemente porque no esperó el tiempo suficiente. Finalmente, realiza dos intentos sobre la base de datos de grafos, alcanzando su límite máximo de iteraciones sin una respuesta que entregar al usuario.

5.3.3 Ideas finales

Habiendo analizado casos como los mencionados anteriormente, se podrían resumir las posibles mejoras en los siguientes puntos:

- Se podría ampliar el tiempo límite de time-out para esperar respuestas por parte de la base de datos vectorial.
- Se podrían refinar los prompts para que entienda cuál herramienta intentar primero.
- Se le podría aclarar que entregue la primera respuesta que haya obtenido por parte de una base de datos.

El Agente es una herramienta que tiene un gran potencial para solucionar ciertas tareas de codificación. Sin embargo, aquí la ingeniería de prompts juega un rol sumamente importante a mismo tiempo que no es una tarea sencilla. Con tiempo, el prompt podría refinarse hasta el punto en que el Agente no dude, para lo cual se necesita de una gran cantidad de inferencias.



## 5.4 ENLACES A MODELOS Y LIBRERÍAS UTILIZADAS

### 5.4.1 Enlaces a librerías

- **Llama Index:** <https://www.llamaindex.ai/>
- **Llama-Index-LLMs-OLlama:** <https://pypi.org/project/llama-index-llms-ollama/>

### 5.4.2 Enlaces a modelos

- **llama3.2:** <https://huggingface.co/meta-llama/Llama-3.2-1B>