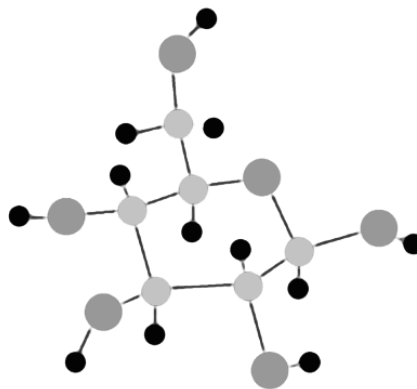


Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura



Agentes Expertos



TUIA - IA 4.2

Procesamiento de Lenguaje Natural

Trabajo Práctico Integrador

Comisión: Única

Docentes:

- Juan Pablo Manson
- Alan Geary
- Andrea Leon Cavallo
- Ariel D'Alessandro

Fecha: 18/12/2023

Grupo:

- Giampaoli Fabio

Introducción

Este proyecto consiste en la creación de un modelo de lenguaje como asistente para conversar con lenguaje natural sobre algún dominio de interés a elección, y la investigación de la posibilidad de convertirlo en un sistema multiagente que pueda conectar con diversas herramientas.

En este caso, mi interés es que el agente sea experto en Genexus. Genexus es un entorno de desarrollo de software que mediante un lenguaje de programación simplificado e integraciones con otras herramientas, se puede generar código fuente para compilar y desplegar aplicaciones.

El objetivo es que el agente pueda tener a disposición la documentación oficial de Genexus para que pueda mantener una conversación al respecto como si Genexus fuera parte de su fuente de su conocimiento, para finalmente darle al usuario que interactúa la sensación de que el agente puede ayudar a resolver dudas y problemas relacionados.

Enunciado - Ejercicio 1

Crear un chatbot experto en un tema a elección, usando la técnica RAG (Retrieval Augmented Generation). Como fuentes de conocimiento se utilizarán al menos las siguientes fuentes:

- Documentos de texto
- Datos numéricos en formato tabular (por ej., Dataframes, CSV, sqlite, etc.)
- Base de datos de grafos (Online o local)
- El sistema debe poder llevar a cabo una conversación en lenguaje español. El usuario podrá hacer preguntas, que el chatbot intentará 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.

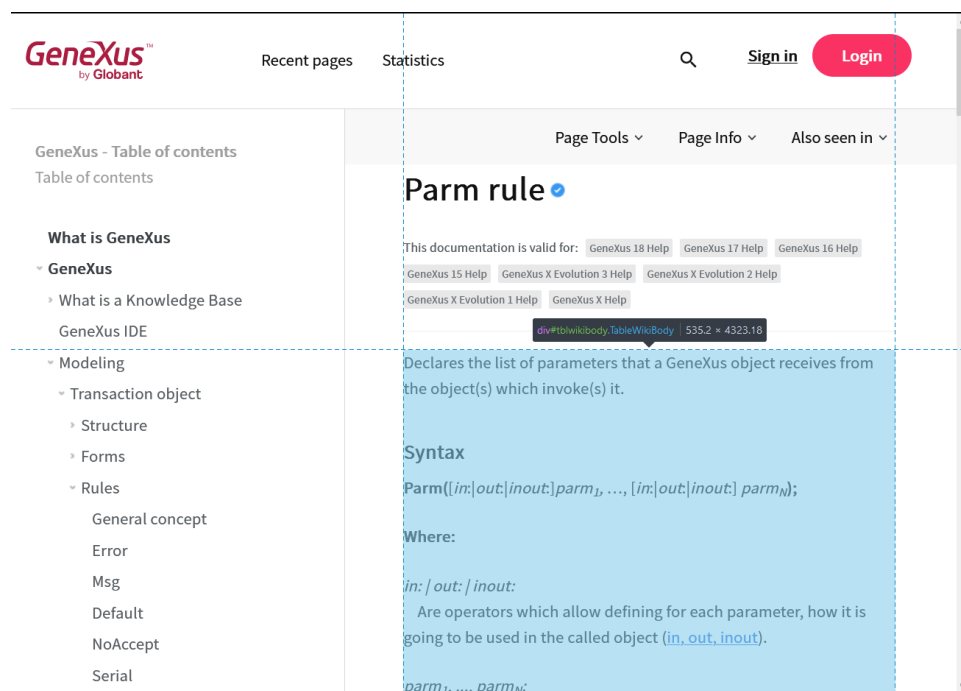
Requerimientos generales:

- Realizar todo el proyecto en un entorno Google Colab
- El conjunto de datos debe tener al menos 100 páginas de texto y un mínimo de 3 documentos.
- Realizar split de textos usando Langchain (RecursiveTextSearch, u otros métodos disponibles). Limpiar el texto según sea conveniente.
- Realizar los embeddings que permitan vectorizar el texto y almacenarlo en una base de datos ChromaDB
- Los modelos de embeddings y LLM para generación de texto son a elección.

Resolución - Ejercicio 1

Para el objetivo de alimentar modelos de lenguaje con información de cómo funciona Genexus se recurre a la documentación oficial de su sitio oficial: [La wiki de Genexus](#).

La estructura de la wiki es particularmente conveniente para este escenario. Podemos notar que la estructura html de la página se puede extraer fácilmente el contenido textual y el código y sintaxis de los ejemplos. De hecho, hay un divisor particular con la clase "TableWikiBody" que contiene toda esta información. Por lo que extraer el texto de este tag particular retorna todo el contenido buscado.



Para extraer el contenido de un link de cualquier página de la Wiki se define un función que hace scrapping para retornar todo el contenido de esta clase. Además, cuenta con un índice de contenidos a la izquierda que es una estructura con desplegaables que te llevan a las diferentes páginas de la Wiki para leer sobre los diferentes temas de interés. Cada elemento del índice representa el título de la página y una referencia al link de dicha página. Por lo que iterar sobre esta estructura también resulta muy simple. De cada link de las referencias se aplica la función de scrapping para obtener su contenido.

También se definen las clases manualmente de cada página para que sea de utilidad para agrupar los tipos de contenidos extraídos. Obteniendo así el siguiente dataframe con pandas:

	Title	Link	Content	Class	Cleaned
0	What is a Knowledge Base	https://wiki.genexus.com//commwiki/wiki?1836,K...	To create a new software system with GeneXus y...	General	create new software system genexus must create...
1	Knowledge Base Creation	https://wiki.genexus.com//commwiki/wiki?9596,N...	You can create a new Knowledge Base by selecti...	General	create new knowledge base select file > new > ...
2	GeneXus IDE	https://wiki.genexus.com//commwiki/wiki?5272,C...	GeneXus provides an intuitive and consistent U...	General	genexus provide intuitive consistent user inte...
(140, 5)					

La etapa de limpieza del texto es sencilla debido a que el texto contiene código, y la eliminación o alteración del código dentro del texto puede tener un impacto negativo al final debido a que podrían ser interpretados palabras del lenguaje de programación que han sido procesadas como si fueran parte del lenguaje.

Además, todo el contenido proviene de la documentación oficial. Por lo que podemos asumir que no hay espacios vacíos, inconsistencias en las explicaciones, o errores ortográficos.

Se utiliza un base de datos vectorial para almacenar todo este conocimiento 'CromaDB' debido a la facilidad de uso y practicidad.

Se define que los documentos serán todos los registros del dataframe depurados, sus índices como strings únicos, y su metadata serán las categorías, url's y títulos. Por lo tanto la colección se forma de la siguiente forma:

```
# creación de la base
chroma_client = chromadb.Client()
collection = chroma_client.create_collection(name="genexus-wiki-tab")

# añade toda la información
collection.add(
    documents=documents,
    metadatas=metadata,
    ids=ids
)
```

Podemos probar por ejemplo una pregunta sobre Genexus a esta base de datos y la misma será capaz de buscar cual es el documento que más se parece a la pregunta utilizando métricas de distancia para medir cercanías en un espacio que captura las representaciones vectoriales de cada documento.

Para darle contexto a los modelos de lenguaje se deben elegir aquellos documentos que más se relacionen con la pregunta del usuario con el fin de preparar previamente al modelo con información que le puede ser de utilidad para responder dicha pregunta.

Pero como un contexto voluminoso para un modelo puede no resultar beneficioso, la idea es separar los documentos cercanos en partes, y quedarse con las partes particulares de estos documentos que más se parezcan a la pregunta inicial del usuario. Esto se logra tokenizando las documentos en oraciones.

Para determinar cuáles son las partes de los documentos más cercanas al contexto de la pregunta, se llevan tanto las oraciones como la pregunta a un espacio vectorial que permitirá determinar qué tan cercanas son las oraciones de la pregunta, y con ello podemos quedarnos con un top de las mejores oraciones para utilizar como contexto.

Por ejemplo, para la pregunta `'what is the variable &trnContext used in transactions events?'`, obtenemos una lista de oraciones cercanas como esta:

```
[  
'syntax event trackcontext (parameters) event_code endevent where: event_code code associate event.',  
'syntax event trn event_code endevent where: event_code code associate event.',  
'define trackcontext event, allow retrieve context information take desire actions.'  
]
```

El modelo de lenguaje que se utilizará para simular conversiones entre un usuario con la documentación de Genexus será [Zephyr](#), un modelo de software libre disponible en Hugging Face para usar mediante Rest API.

Este modelo se alimenta de un rol que el mismo debe cumplir en la conversación, el contexto para ubicarse para responder preguntas, y la pregunta del usuario como tal. La estructura de la consulta está reglada, y debe tener esta forma:

```
# envía la consulta al servidor y responde  
def query(payload):  
    response = requests.post(API_URL, headers=headers, json=payload)  
    return response.json()  
  
# estructura del input para el modelo  
quest_to_model = f""" <|system|>  
{rol}</s>  
<|user|>  
Context: {context}  
Question: {user_query}  
</s>  
<|assistant|> """""  
  
output = query({  
    "inputs": quest_to_model  
})
```

La idea es que una única llamada al modelo con una pregunta pueda resolver toda la lógica asociada en encontrar el contexto adecuado, y recibir la respuesta esperada en la salida. Para ello se crea una función que encuentra oraciones de la documentación que menor distancia de coseno tengan con la pregunta en cuestión, y luego darle dicho contexto al modelo junto con la pregunta y el rol.

De esta manera se logran respuestas esperadas a una llamada como esta:

```
question = 'Explain me about inout parameters in procedures'
print(query_to_model(user=question, assistant=rol))
```

Inout parameters in procedures allow passing values to a procedure and also allowing the procedure to modify those values. This is different from input parameters, where the procedure only receives the value, and output parameters, where the procedure returns a value.

Inout parameters are declared with the "inout" keyword in Genexus, and they can be used to pass complex data structures or large amounts of data between the procedure and the calling program. This can improve performance by avoiding the need to copy large amounts of data between the procedure and the calling program.

Here's an example of how to declare an inout parameter in Genexus:

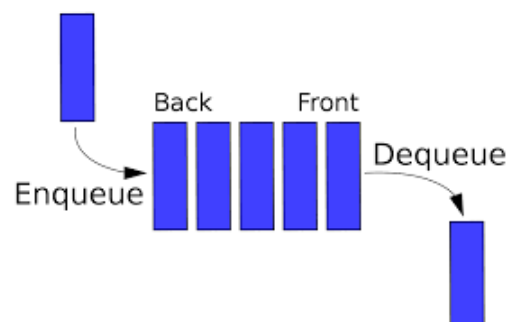
```
...
procedure myProcedure(inout: myStruct)
begin
  // Modify the inout parameter here
end
...
```

Se puede notar una respuesta bastante precisa al menos a nivel conceptual. Aunque al momento de generar código demuestra utilizar syntaxis que originalmente Genexus no utiliza, pero tampoco es tan ajeno la forma general del código a la respuesta esperada.

Un problema que se puede presentar en un chatbot como este es que el modelo no tenga memoria de lo que el usuario o el mismo haya dicho previamente en otro mensaje. Para ello se plantea una estructura de datos para el contexto como una lista de strings que se van modificando a lo largo de la conversación, pero manteniendo el contexto reciente en memoria.

Para ello se define una cola (Queue) donde se almacena el contexto de los documentos más el contexto de la conversación en una cola que se va limpiando a medida que la conversación avanza.

Por ejemplo, una memoria de 8 unidades implica que contempla las 3 oraciones de la documentación más cercanas a la pregunta previa, las tres oraciones más cercanas a la pregunta actual en la documentación, y las tres más cercanas al texto de la respuesta previa. El contexto sería una lista con este comportamiento:



Por lo que se hacen modificaciones sobre las funciones anteriores para que no solo guarde en el contexto las oraciones de la documentación, sino las oraciones más importantes de la respuesta anterior también estén en este contexto. Esto se logra agregando como parámetro un contexto adicional al que ya busca por sí mismo en la documentación.

Otro problema que se presenta es que debido a que la documentación de Genexus está en inglés y no se puede obtener directamente en español, se opta por añadir un paso más a la hora de interactuar, que es la traducción automática del texto tanto ingresado como retornado, debido a que en la interfaz el usuario debe poder conversar en español, pero el modelo de fondo trabaja en inglés. Para ello se utiliza el traductor de Google desde una librería de Python.

```
rol_en = GoogleTranslator(source='es', target='en').translate("Eres un programador experto Genexus que siempre responde utilizando el contexto proporcionado.")
print(rol_en)
```

```
rol_es = GoogleTranslator(source='en', target='es').translate("You are an Genexus expert programmer that always responds using the provided context.")
print(rol_es)
```

You are an expert Genexus programmer who always responds using the context provided.

Eres un programador experto Genexus que siempre responde utilizando el contexto proporcionado.

Un problema que se presenta es que el código no debe ser traducido ya que usa palabras reservadas del lenguaje. Para ello se crea una función para evitar la traducción en bloques de texto evitando aplicar traducción en partes de la respuesta donde esté encerrado entre triples comillas simples al inicio y fin, que representa un bloque de código.

Esto permite generar respuestas más adecuadas como estas:

```
...
5. En el editor de fórmulas, ingrese la siguiente fórmula:
...
```

```
Sum(Price, Where(Level = Level - 1))
...
```

```
6. Guarde la fórmula y cierre el editor de fórmulas.
...
```

Finalmente se logra un modelo de conversación que permite ingresar preguntas en español sobre algún tema de Genexus, y el modelo de lenguaje tomará información relacionada de las fuentes de datos para tratar de responder de forma más precisa en español manteniendo la coherencia para los bloques de código y mantener una memoria de corto plazo para mejor experiencia en la conversación. Y a continuación se presenta la estructura general que permite conversar con el modelo a través de la terminal de Python:

```

# inicializa el contexto y rol
global_context = []
rol = 'You are an Genexus expert programmer that always responds using the provided context
with brief responses and short pieces of code.'
first_question = True

# iteracion de la conversacion
while True:
    user_input = input("\u0001 [User]: ")
    user_input_en = GoogleTranslator(source='es', target='en').translate(user_input)

    print('\n')

    # keyword de escape
    if user_input.lower() == 'exit':
        print("Exiting the conversation.")
        break

    # obtiene respuesta de acuerdo de acuerdo a si usa o no contexto previo
    if first_question:
        model_answer = query_to_model(user_input, rol, memory=6)
        first_question = False
    else:
        model_answer = query_to_model(user_input, rol, context=model_answer, memory=6)

    # traducir al español la respuesta
    model_answer_es = translate_except_code(model_answer, 'en', 'es')

    print("\u0001 [Model]:", model_answer_es, '\n')

```

Este por ejemplo es una breve conversación con el modelo de lenguaje en español sobre consultas particulares sobre desarrollo en Genexus.

[User]: Enseniame a hacer un procedimiento que tome dos numeros y retorno su suma

[Model]: Aquí hay un procedimiento Genexus que toma dos números como entrada y devuelve su suma:

1. Cree un nuevo procedimiento en Genexus y asígnele el nombre "AddNumbers".
2. Agregue dos parámetros de entrada de tipo numérico, denominados "num1" y "num2".
3. Agregue un parámetro de salida de tipo numérico, denominado "suma".
4. En el cuerpo del procedimiento, agregue el siguiente código:

...

```
sum = num1 + num2
```

...

5. Guarde y pruebe el procedimiento.

Aquí está la sintaxis para llamar al procedimiento:


```
...
```

```
CALL PROCEDURE("AddNumbers") PARAM("num1", <num1 value>) PARAM("num2",  
<num2 value>) RESULT("sum")
```

```
...
```

Reemplace `**<valor num1>**` y `**<valor num2>**` con los valores reales que desea agregar. El resultado del procedimiento se almacenará en la variable `**suma**`.

[User]: Esta bien. Y ahora si quisiera que divida los numeros de la entrada en lugar de sumar? debes contemplar el caso de que el divisor no sea 0

[Model]: Para dividir los números ingresados en lugar de sumarlos, se puede crear un nuevo procedimiento Genexus con los siguientes pasos:

1. Crear un nuevo procedimiento en Genexus con el nombre "dividenumbers".
2. Agregue dos parámetros de entrada de tipo numérico con los nombres "num1" y "num2".
3. Agregue un parámetro de salida de tipo numérico con el nombre "resultado" para devolver el resultado de la división.
4. Dentro del procedimiento, agregue el siguiente código:

```
...
```

```
if num2 = 0 then  
  messagebox("Error: Divisor cannot be zero.")  
  return  
end if
```

```
result = num1 / num2
```

```
...
```

Este código verifica si el divisor es cero y muestra un mensaje de error si lo es. Si el divisor no es cero, realiza la división y devuelve el resultado.

Para llamar a este procedimiento, puede utilizar la siguiente sintaxis:

```
...
```

```
call procedure("dividenumbers") param("num1", <num1 value>) param("num2", <num2  
value>) result("result")
```

```
...
```

Reemplace `**<valor num1>**` y `**<valor num2>**` con los valores reales que desea dividir. El resultado se devolverá en el parámetro "resultado".

[User]: ...

Enunciado - Ejercicio 2

Realice una investigación respecto al estado del arte de las aplicaciones actuales de agentes inteligentes usando modelos LLM libres.

Plantee una problemática a solucionar con un sistema multiagente. Defina cada uno de los agentes involucrados en la tarea.

Realice un informe con los resultados de la investigación y con el esquema del sistema multiagente, no olvide incluir fuentes de información.

Opcional: Resolución con código de dicho escenario.

Resolución - Ejercicio 2

Los agentes inteligentes son modelos que pueden percibir su entorno, razonar sobre él y actuar de forma autónoma o semiautónoma para lograr sus objetivos. Los modelos LLM son sistemas de inteligencia artificial que pueden generar y comprender lenguaje natural a gran escala, utilizando una gran cantidad de datos textuales como entrenamiento.

Por lo que un agente basado en LLM tiene gran potencial para resolver problemas reales de cualquier tipo si se lo ubica en un contexto adecuado. Actualmente existe gran variedad de modelos tanto privados como de código libre.

Ejemplos de agentes inteligentes privados son Chat GPT o Bard. Ambos modelos son potencialmente útiles y ampliamente utilizados a lo largo del mundo, y tienen la habilidad de responder a situaciones complejas o simples que presentan los usuarios en diversos contextos usando solo lenguaje natural humano hablado o escrito.

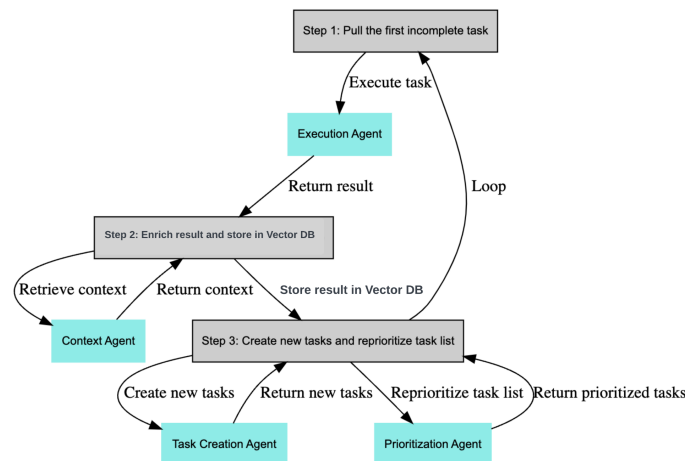
Últimas actualizaciones de modelos como estos han sido las integraciones con otros sistemas para que estos agentes inteligentes orquesta un sistema donde interactúan múltiples agentes para resolver problemas. Una de ellas es la carga y el análisis de archivos, que permite al usuario subir documentos o imágenes y obtener información relevante sobre ellos, como resúmenes, categorías o etiquetas. Otra es la integración con otras herramientas externas, como Navegación, Análisis avanzado de datos y sistemas de generación y reconocimiento de imágenes.

Sin embargo, estos sistemas multiagente por el momento solo están disponibles para aquellos usuarios que deseen abonar por el servicio. Es decir, no son sistemas de libre acceso, y mucho menos, de código libre para que cualquiera que desee pueda interpretar su funcionamiento.

Actualmente existen proyectos públicos de código abierto que intentan lograr sistemas multiagentes como los sistemas anteriores. Un ejemplo de ello es AutoGPT. Es un proyecto de código libre en Github similar a Chat GPT como modelo de lenguaje para resolver problemas genéricos, pero combinado con técnicas de auto-prompting para aumentar la autonomía del modelo.

Últimas actualizaciones del modelo han sido por ejemplo integraciones con 11 Labs, un sistema de IA que a partir del texto genera una voz sintetizada para que puedas "hablar" con Auto-GPT.

Otra alternativa muy interesante es el proyecto BabyAGI, un modelo publicado en Github que tiene la intención de ser un modelo de lenguaje natural interactivo que hace uso de diferentes agentes que interactúan entre sí para resolver problemas complejos. En su repositorio se obtiene una imagen explicativa como esta donde se abstrae el funcionamiento, coordinación e interacción entre los agentes del sistema:



En este proyecto se explica que la intención es que el modelo sea lo más autónomo posible, generando respuestas de acuerdo a la prioridad que le asigna un agente en el sistema a las tareas que crea otro agente, y luego generar el contexto y ejecutar la resolución con otros agentes especializados en dicha tarea.

El estado actual de este proyecto aún está en curso y en etapas tempranas, pero actualmente es posible ejecutar este sistema basado en otros modelos lenguajes mediante uso de API's.

—

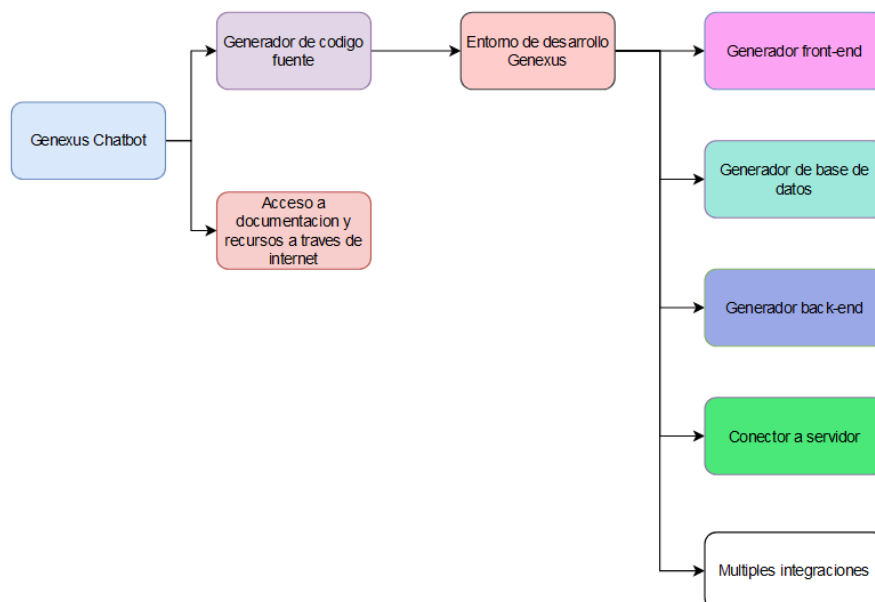
Un problema que se puede solucionar con un sistema multiagente en el contexto de este proyecto, puede ser la integración del chatbot con el generador de código de Genexus. Es decir, Genexus es un modelo que procesa versiones simplificadas de código en su propio lenguaje de programación para posteriormente con generadores de códigos configurados, se traslade esta lógica a un lenguaje de más bajo nivel (con respecto a lenguaje Genexus) como .Net o Java.

De esta forma se podría lograr un agente conectado a un entorno de desarrollo funcional que pueda permitir la ejecución y despliegue de funcionalidades que el usuario desee con prompts simplificados a solo su funcionamiento lógico a nivel de lenguaje natural, y así absorber aún más la complejidad del desarrollo de aplicaciones.

De hecho, sería de gran utilidad si se pudiera integrar este chatbot a un entorno de desarrollo 'vivo' donde el código generado por el chatbot conectado al generador de código de Genexus, pueda ser ejecutado y compilado en tiempo real, para desarrollar aplicaciones de forma mucho más interactiva.

De por sí Genexus abarca muchísimas integraciones de forma simplificada al desarrollador. Por lo que actividades como versionado del proyecto, seguridad, generación de interfaces de usuario, llamada a módulos externos con gran diversidad de funcionalidades no nativas, podrían ser accedidos desde la integración con Genexus, ya que Genexus como entorno sería el puente entre todas las integraciones externas y el chatbot.

El siguiente diagrama sería una representación de los agentes que interactúan en un sistema como el planteado.



Referencias

Integraciones Chat GPT:

<https://blog.clickpanda.com/tecnologia/chat-gpt-y-sus-ultimas-actualizaciones/>

Carga de archivos Chat GPT:

<https://appmaster.io/es/news/nueva-actualizacion-beta-chatgpt-plus>

Navegación y análisis Chat GPT:

<https://www.elgrupoinformatico.com/utilidades/chatgpt-se-renueva-2023/>

Integración de YouTube a Bard:

<https://www.lanacion.com.ar/tecnologia/bard-el-chatgpt-de-google-ahora-va-a-mirar-videos-de-youtube-por-vos-y-armara-un-resumen-nid23112023/>

Que es AutoGPT:

[El siguiente salto de ChatGPT se llama Auto-GPT, genera código de forma "autónoma" y ya está aquí \(xataka.com\)](#)

Repositorio Github AutoGPT:

[GitHub - Significant-Gravitas/AutoGPT: AutoGPT is the vision of accessible AI for everyone, to use and to build on. Our mission is to provide the tools, so that you can focus on what matters.](#)

Repositorio Github BabyAGI:

[GitHub - yoheinakajima/babyagi](#)

Funcionamiento BabyAGI:

[BabyAGI: El revolucionario sistema de gestión de tareas impulsado por IA |2023 !\[\]\(4436e6b00b9d5e62c2a161129eb3e4d0_img.jpg\) \(gptpro.es\)](#)