

Herramientas para la Computación en la Nube Dirigida a  
Inteligencia Artificial

---

# Tema 6. Ingeniería de prompts

# Índice

## Esquema

### Ideas clave

- 6.1. Introducción y objetivos
- 6.2. Generación de textos con ChatGPT
- 6.3. Técnicas avanzadas para generación de textos con LangChain
- 6.4. Uso de FAISS y Pinecone
- 6.5. Agentes autónomos
- 6.6. Generación de imágenes con diffusion models
- 6.7. Generación de imágenes con Midjourney
- 6.8. Técnicas avanzadas de generación de imágenes con Stable Diffusion
- 6.9. Cuaderno de ejercicios
- 6.10. Referencias bibliográficas

### A fondo

- ChatGPT prompt engineering for developers
- Prompt engineering for ChatGPT

### Test

| INGENIERÍA DE PROMPTS   |   |  |  |  |  |
|---|---|--|--|--|--|
| CHATGPT   | LANGCHAIN   | VECTOR STORES  | AGENTES  | IMÁGENES   |  |
| <ul style="list-style-type: none"><li>▶ <b>Formatos de salida</b><ul style="list-style-type: none"><li>• JSON, YAML, flowchart</li><li>• Datasets</li></ul></li></ul>   | <ul style="list-style-type: none"><li>▶ <b>Generación de prompts</b></li></ul>  | <ul style="list-style-type: none"><li>▶ <b>FAISS</b><ul style="list-style-type: none"><li>• Creación de índices</li></ul></li></ul>    | <ul style="list-style-type: none"><li>▶ <b>ReAct</b><ul style="list-style-type: none"><li>• Agentes con LangChain</li></ul></li></ul>            | <ul style="list-style-type: none"><li>▶ <b>Ingeniería de prompts</b></li></ul>   |  |
| <ul style="list-style-type: none"><li>▶ <b>Precisión de resultados</b><ul style="list-style-type: none"><li>• Contexto</li><li>• Referencias</li></ul></li></ul>  | <ul style="list-style-type: none"><li>▶ <b>Técnicas avanzadas</b><ul style="list-style-type: none"><li>• Streaming de respuestas</li><li>• Paralelización</li><li>• Plantillas</li><li>• Pipelines</li><li>• Datos estructurados</li><li>• Funciones</li><li>• Documentos</li></ul></li></ul> | <ul style="list-style-type: none"><li>▶ <b>Pinecone</b><ul style="list-style-type: none"><li>• Creación de índices</li></ul></li></ul> | <ul style="list-style-type: none"><li>▶ <b>OpenAI functions</b><ul style="list-style-type: none"><li>• Agentes con LangChain</li></ul></li></ul> | <ul style="list-style-type: none"><li>▶ <b>Midjourney</b><ul style="list-style-type: none"><li>• Quality boosters</li><li>• Inpainting</li><li>• Outpainting</li><li>• Consistencia</li><li>• Estilos indirectos</li><li>• Bases de datos de prompts</li></ul></li></ul> |  |
| <ul style="list-style-type: none"><li>▶ <b>Generación de código fuente</b></li></ul>  |   |  |  | <ul style="list-style-type: none"><li>▶ <b>Stable Diffusion</b><ul style="list-style-type: none"><li>• Python SDK</li><li>• AUTOMATIC1111</li></ul></li></ul>  |  |
| <ul style="list-style-type: none"><li>▶ <b>Modelos de clasificación</b></li></ul>   |   |  |  |  |  |
| <ul style="list-style-type: none"><li>▶ <b>Modificación de estilo</b><ul style="list-style-type: none"><li>• Perfil de usuario</li><li>• Estilo</li><li>• Rol</li></ul></li></ul>                                       |   |  |  |  |  |
| <ul style="list-style-type: none"><li>▶ <b>Generación de listas</b><ul style="list-style-type: none"><li>• Personalización</li><li>• Listas jerárquicas</li></ul></li></ul>   |   |  |  |  |  |
| <ul style="list-style-type: none"><li>▶ <b>Análisis de contenido</b><ul style="list-style-type: none"><li>• Entidades</li><li>• Resúmenes</li><li>• Textos largos</li><li>• Análisis de sentimiento</li></ul></li></ul> |   |  |  |  |  |
| <ul style="list-style-type: none"><li>▶ <b>Traducción</b></li></ul>   |   |  |  |  |  |
| <ul style="list-style-type: none"><li>▶ <b>Extracción de tokens</b></li></ul>   |   |  |  |  |  |

## 6.1. Introducción y objetivos

En este tema se describe la ingeniería de *prompts* o *prompt engineering*, disciplina imprescindible para el uso de modelos de IA conversacional (LLMs), desde diferentes puntos de vista. Un *prompt* es una instrucción, pregunta o texto que envía el usuario al modelo para obtener una respuesta, un resultado o la ejecución de una acción.

En primer lugar, se proporcionan ejemplos prácticos de **construcción de prompts en ChatGPT**. Se recomienda registrarse en OpenAI para poder reproducir estos *prompts* y comprobar los resultados obtenidos, así como para experimentar con variaciones de estos. A continuación, se describen la **ingeniería de prompts** desde la perspectiva de LangChain, el orquestador de LLMs más popular del mercado. También se describen las herramientas **FAISS** y **Pinecone**, que permite incorporar *vector stores* a la arquitectura para disponer de búsquedas basadas en *embeddings* y la posibilidad de comparar el grado de similitud entre textos. Asimismo, se describe el concepto de **agente autónomo** y la manera en que puede implementarse en LangChain con dos aproximaciones: el *framework* ReAct y las OpenAPI *functions*.

El resto de las secciones cubren la **generación de imágenes** a partir de *prompts*, con sus propias técnicas de *prompting*. Los ejemplos se basan en DALL-E, Midjourney y Stable Diffusion XL.

Por último, este tema se plantea los siguientes objetivos para el alumno:

- ▶ Familiarizarse con las técnicas avanzadas de ingeniería de *prompts* para resolver condicionantes y escenarios personalizados en el ámbito de la IA conversacional.
- ▶ Desarrollar aplicaciones con IA generativa basada en *prompts*.
- ▶ Disponer de las habilidades para generar contenido digital sin *copyright* a través de diferentes plataformas de generación de imágenes con IA.

## 6.2. Generación de textos con ChatGPT

### Introducción

El uso de ChatGPT es actualmente **generalizado**, principalmente porque la curva de aprendizaje para materializar sus beneficios es prácticamente nula: se emplea el mismo tipo de lenguaje que utilizamos a diario en una conversación. Sin embargo, la mayoría de los usuarios no son conscientes de la enorme capacidad de comprensión y razonamiento de este chatbot, lo que deriva en un uso limitado. Si aplicamos *prompts* más elaborados, la calidad de las respuestas se incrementa notablemente, posibilitando nuevos casos de uso.

En esta sección se describe un **conjunto de técnicas** para personalizar y ajustar de manera precisa el contenido de las respuestas de ChatGPT. El objetivo es que dichas respuestas cumplan con exactitud los requisitos de nuestro caso de uso.

En muchos casos, estas técnicas son especialmente útiles en el contexto de una aplicación que utiliza GPT —el modelo en el que se basa ChatGPT— u otros modelos similares, como, por ejemplo, BERT (desarrollado por Google). Por tanto, el conocimiento de estas puede propiciar la aparición de ideas para mejorar un determinado *software*, incorporando funcionalidades basadas en las capacidades conversacionales de la **IA generativa**.

### Formatos de salida

Por defecto, ChatGPT genera respuestas en **lenguaje natural**. Sin embargo, existe la opción de especificar un formato distinto, de modo que esta respuesta pueda ser tratada más fácilmente por una aplicación.

## Formato JSON

Por ejemplo, ChatGPT puede formatear el resultado en formato JSON:

Prompt: Genera una lista de materiales válidos para construcción con sus características en formato JSON

## Formato YAML

También es posible solicitar a ChatGPT que formatee la salida en YAML:

Prompt: Si el producto A vale 10€, el producto B vale 20€ y el producto C vale 30€, genera estos datos en formato YAML

## Diagrama de flujo

El formato de salida puede adoptar el aspecto de un *flowchart*:

Prompt: Crea un diagrama de flujo con sintaxis de Mermaid que refleje las etapas de la fotosíntesis

El resultado es el siguiente (abreviado):

```
graph TD;
  A[Fase luminosa] --> B[Captación de luz];
  B --> C[Producción de ATP y NADPH];
  ...
```

Este resultado puede transformarse en un gráfico, utilizando un editor *online* de Mermaid. En la Figura 1 se muestra el resultado:

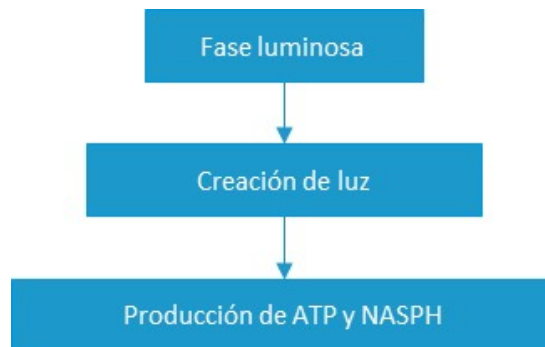


Figura 1. Diagrama generado con Mermaid a partir de la salida de ChatGPT. Fuente: elaboración propia.

## Generación de *datasets*

Por último, ChatGPT puede generar ficheros en formato CSV, incluyendo el contenido de este a partir del nombre de las columnas:

Prompt: Genera un csv con una muestra de personas, indicando nombre, edad y profesión

Esta opción permite generar rápidamente *datasets* de prueba para prototipar un algoritmo o comparar el rendimiento de varios algoritmos. También es útil para disponer de datos ficticios (sintéticos), sin necesidad de comprometer la confidencialidad de datos reales sensibles o tener que acudir a técnicas de ofuscación.

## Precisión de resultados

### Evaluación de contexto

En ocasiones, ChatGPT proporciona respuestas demasiado generales, que no cumplen con las expectativas del usuario. Sin embargo, el usuario puede influir en el grado de detalle obtenido, proporcionando **contexto**.

Por ejemplo:

Prompt: ¿Debería comprar un coche de marca Fiat o Opel? mi preocupación principal es la seguridad.

Prompt: ¿Debería ir de vacaciones a Canarias o a Alaska? no me gusta pasar frío.

Como se puede observar, el contexto puede aparecer de forma implícita. A pesar de ello, ChatGPT considera los condicionantes facilitados por el usuario para generar una respuesta más precisa.

### **Información de referencia o *few-shot learning***

ChatGPT puede basar su respuesta en un **texto de referencia** facilitado en el propio *prompt*. Se trata de una forma sencilla de reentrenar el modelo para que las respuestas se ajusten a un contenido determinado.

Por ejemplo:

Prompt: Utiliza el siguiente texto que incluyo entre comillas triples para responder a las preguntas. Si no encuentras la respuesta en ese texto, simplemente di que no lo sabes.

"""

La capital de España es Madrid.

"""

¿Cuál es la capital de Francia?

En el ejemplo anterior, la respuesta será que no lo sabe. La explicación es que ChatGPT se basa exclusivamente en la **información de referencia** proporcionada por el usuario para generar la respuesta, y en este caso es insuficiente. De esta forma, nos aseguramos de que ChatGPT no proporcione respuestas imprecisas, descontextualizadas o que no se ajustan a nuestro caso de uso.



Esta técnica también es un mecanismo para evitar las denominadas «**alucinaciones**», es decir, cuando ChatGPT inventa una respuesta sin sentido, en lugar de admitir que no conoce la respuesta.

### Generación de código fuente

ChatGPT puede utilizarse para esbozar el diseño de una aplicación, implementar cada una de sus funciones y generar los tests apropiados.

Por ejemplo, para describir los **pasos** que deben seguirse en la implementación de una aplicación, utilizamos el siguiente *prompt*:

Prompt: Resume la arquitectura de una aplicación "Hello, world!" en Flask.

A continuación, se incluye un extracto de la respuesta:

En resumen, la arquitectura de una aplicación "Hello, world!" en Flask implica una función de vista que maneja una solicitud HTTP y devuelve una respuesta que incluye el mensaje "Hello, world!", junto con una plantilla HTML si es necesario.

Para la **implementación** de esta aplicación, podemos solicitar directamente que escriba el código fuente en Python:

Prompt: Escribe el código fuente correspondiente a la aplicación anterior.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')

def hello_world():

    return 'Hello, world!'

if __name__ == '__main__':
```

```
app.run()
```

Finalmente, ChatGPT puede escribir el código que permite probar la **aplicación**:

Prompt: Escribe el código fuente que permite probar la aplicación anterior.

```
import requests

url = 'http://127.0.0.1:5000/'

response = requests.get(url)

print(response.text)
```

## Modelos de clasificación

La técnica de ***few-shot learning*** o información de referencia descrita previamente puede emplearse también para generar modelos de clasificación a partir de ejemplos proporcionados en el *prompt*. De esta forma, ChatGPT puede asignar **etiquetas o categorías** a un texto nuevo, basándose en dichos ejemplos.

Por ejemplo:

Clasifica una frase como "halago", "queja" o "neutral" a partir de los siguientes ejemplos:

1. "El restaurante está abierto." - Neutral
2. "Los camareros son muy agradables" - Halago
3. "Mi experiencia durante la cena fue terrible" - Queja

Debes tener en cuenta:

- Responde sólo con la categoría.
- Clasifica el texto incluido entre comillas triples.

```
"""El menú es muy completo e incluye alimentos saludables"""
```

ChatGPT etiqueta la frase anterior como halago. Esta capacidad puede emplearse para un amplio abanico de casos de uso:

- ▶ Opiniones de clientes.
- ▶ Clasificación de correos electrónicos.
- ▶ Análisis de comentarios en medios sociales.
- ▶ Categorización de noticias.
- ▶ Análisis de currículums.

## Modificación de estilo

Por defecto, ChatGPT adapta el estilo de la respuesta a la temática y enfoque de la pregunta. Sin embargo, el usuario puede solicitar un cambio en el estilo de la respuesta.

## Adaptación al perfil del usuario

Por ejemplo, para adaptarla al conocimiento de un **perfil de usuario concreto**:

Prompt: Describe resumidamente en qué consiste una gripe. Explícalo para un niño de 10 años.

Esta posibilidad permite generar **contenido dinámico** en una aplicación, en función del perfil del usuario que consulta la información.

## Rasgos de estilo

Por otro lado, el usuario puede facilitar un texto y pedirle a ChatGPT que extraiga los rasgos de estilo de dicho texto:

Prompt: Analiza el siguiente texto e identificar rasgos de estilo que podría utilizar para escribir un texto similar.

Analiza el siguiente texto:

La operadora francesa celebró este jueves la inauguración de su ruta entre Valladolid y Madrid con la que amplía su presencia en el mercado de la alta velocidad española, y lo hace con una oferta promocional con billetes a un euro durante dos semanas.

A continuación, se muestra una parte de la salida:

El texto proporciona información clara y concisa sobre un evento específico, utilizando un estilo periodístico que se centra en los detalles relevantes. Algunos rasgos de estilo que podrías utilizar para escribir un texto similar son:

Objetividad: El texto presenta los hechos de manera objetiva, sin opinión subjetiva o juicio de valor.

Utilizando el histórico de la conversación, el usuario puede solicitar que escriba un texto con esos mismos **rasgos de estilo**:

Prompt: Escribe una noticia ficticia sobre un aumento de impuestos, utilizando el estilo del texto anterior.

Esta posibilidad permite a una aplicación **homogeneizar el estilo** de una colección de contenidos de forma automática, corrigiendo las posibles variaciones de estilo introducidas por los autores originales.

## Adopción de rol

De forma similar, ChatGPT puede adoptar en la respuesta el rol de un personaje o perfil de persona, narrando la respuesta con el estilo propio del personaje o perfil.

Por ejemplo:

Prompt: Describe la empresa Nvidia como si fueras un analista financiero.

[...]. Como analista financiero, es importante destacar que NVIDIA ha experimentado un crecimiento significativo en los últimos años, impulsado en gran medida por la creciente demanda de sus productos en el mercado de

juegos, así como por su éxito en áreas emergentes como la inteligencia artificial y el aprendizaje automático. [...]

Esta capacidad puede emplearse en el **contexto de UX Writing** tanto para la generación de contenidos en un sitio web como para el diseño de interfaces de usuario de aplicaciones. Permite generar texto con el estilo apropiado, fomentando la creatividad o el rigor, dependiendo del propósito del contenido. Este grado de personalización de las narrativas provoca un mayor compromiso o vinculación del usuario que consume el contenido.

### Generación de listas

ChatGPT tiene posibilidad de generar una lista de la temática solicitada. Por ejemplo, puede responder con una lista de alimentos saludables.

### Personalización de listas

Sin embargo, es posible introducir varios requisitos adicionales a las listas. En el siguiente ejemplo se introducen varias **limitaciones**:

Prompt: Genera una lista numerada de alimentos saludables. No incluyas ejemplos. Debe incluir un máximo de tres elementos.

### Listas jerárquicas

También es posible generar listas organizadas de forma jerárquica. Por ejemplo:

Prompt: Genera una lista jerárquica que describa los tipos de deportes con ejemplos.

Estas listas pueden ser procesadas por una aplicación para, por ejemplo, generar opciones de forma dinámica en una interfaz de usuario.

## Análisis de contenido

ChatGPT puede analizar un texto dado y extraer información relevante, de acuerdo con los criterios de análisis que proporcione el usuario.

## Extracción de entidades

Como ejemplo, con esta técnica se pueden identificar las **palabras clave** de un texto. De esta forma, una aplicación podría recuperar información adicional actualizada, realizando una consulta en un buscador:

Prompt: analiza el siguiente texto y extrae las entidades clave: jueces piden a la UE que la devolución de los gastos hipotecarios no prescriba. No incluyas descripciones.

Las entidades que enumera en la respuesta son: jueces, UE, devolución, gastos hipotecarios, prescriba.

## Resumen de textos

También puede proporcionar un resumen de un texto, incluso de textos o contenidos no proporcionados por el usuario, pero disponibles en Internet:

Prompt: resume la película E.T. en una línea.

## Manejo de textos largos

Cuando el texto que se pretende resumir es demasiado extenso y no puede proporcionarse como contexto en el *prompt*, es habitual dividirlo en **fragmentos** y solicitar el resumen de cada uno de los fragmentos.

La extensión máxima permitida es un elemento de diseño del LLM, y por lo tanto depende del LLM utilizado. En otros casos, la limitación no es en sí la extensión del texto en el *prompt*, sino el número de tokens que el modelo es capaz de procesar en cada interacción.

Existen múltiples técnicas para realizar **chunking, o división en fragmentos**, de un texto, sin embargo, es importante considerar que la elección de esta debe estar guiada por la **preservación del significado**, de manera que ChatGPT pueda interpretar el contenido y extraer de él las ideas clave.

### Análisis de sentimiento

ChatGPT también puede llevar a cabo un análisis de sentimiento, es decir, si el **tono** del texto facilitado es positivo, negativo o neutro.

Por ejemplo:

Prompt: ¿Es este texto positivo o negativo? "Me gusta el diseño de este coche, pero es demasiado caro"

El resultado es el siguiente:

Este texto tiene una connotación negativa. Aunque expresa un gusto por el diseño del coche, el sentimiento negativo surge al calificarlo como "demasiado caro", lo que sugiere una percepción negativa debido al precio elevado.

### Traducción

ChatGPT incorpora por defecto la capacidad de traducir textos, incluyendo las respuestas que él mismo genera:

Prompt: define qué es ChatGPT en inglés.

La **capacidad** de traducción de ChatGPT depende de la **cantidad de texto disponible** en Internet en los idiomas de origen y destino de la traducción. Es decir, para idiomas poco presentes en Internet, la calidad de la traducción será significativamente menor que la de idiomas predominantes, como inglés y español.

### Extracción de tokens

Uno de los aspectos que debe considerarse cuando se utilicen modelos GPT (el utilizado por ChatGPT) en Azure OpenAI es la **cuota de tokens máximos** permitidos por minuto. En la Figura 2, se incluye una referencia de dichas *quotas* en función del modelo, extraída de la web de Azure OpenAI:

| Region        | GPT-4 | GPT-4-32K | GPT-4-Turbo | GPT-4-Turbo-V | GPT-35-Turbo | GPT-35-Turbo-Instruct | Text-Embedding-Ada-002 |
|---------------|-------|-----------|-------------|---------------|--------------|-----------------------|------------------------|
| australiaeast | 40 K  | 80 K      | 80 K        | 30 K          | 300 K        | -                     | 350 K                  |
| brazilsouth   | -     | -         | -           | -             | -            | -                     | 350 K                  |
| canadaeast    | 40 K  | 80 K      | 80 K        | -             | 300 K        | -                     | 350 K                  |
| eastus        | -     | -         | 80 K        | -             | 240 K        | 240 K                 | 240 K                  |
| eastus2       | -     | 80 K      | 80 K        | -             | 300 K        | -                     | 350 K                  |
| francecentral | 20 K  | 60 K      | 80 K        | -             | 240 K        | -                     | 240 K                  |

Figura 2. Valores de *cuota* por región y modelo de OpenAI. Fuente: Mrbullwinkle, ChrisHMSFT, PatrickFarley, nitinme, nkgami, learn-build-service-prod[bot], aahill y eric-urban, 2024.

Estos datos son especialmente útiles cuando la generación de tokens y la extracción de respuestas se realiza desde una **aplicación**. En ese caso, es posible utilizar una librería como Tiktoken para contabilizar los tokens consumidos en cada interacción, y de esa forma identificar de forma anticipada cuándo se va a superar la *cuota* establecida.



A continuación, se muestra un ejemplo en Python:

```
import tiktoken

encoding = tiktoken.get_encoding("cl100k_base")

print(len(encoding.encode("Learning how to use Tiktoken is fun!")))
```

El **sistema de codificación cl100k\_base** es el empleado por los modelos de OpenAI GPT-4, GPT-3.5-turbo y text-embedding-ada-002. Por tanto, se trata de la codificación utilizada por ChatGPT.

## 6.3. Técnicas avanzadas para generación de textos con LangChain

LangChain es una plataforma *open-source* de orquestación de LLMs, que permite utilizar una **interfaz común en Python** para acceder a estos LLMs y a otras funcionalidades y herramientas para potenciar sus resultados, tales como: *vector stores*, *data augmented generation* (DAG), agentes o uso de memorias.

En esta sección, veremos un ejemplo básico de uso y, a continuación, describiremos algunas técnicas para usuarios avanzados de LangChain.

### Generación de prompts

En el siguiente ejemplo se genera y envía un *prompt* a OpenAI, y se recibe e imprime la respuesta. Está escrito en Python y utiliza el SDK de LangChain para este lenguaje, que puede instalarse como paquete pip.

```
from langchain_openai.chat_models import ChatOpenAI

from langchain.schema import AIMessage, HumanMessage, SystemMessage

chat = ChatOpenAI(temperature=0.5)

messages = [SystemMessage(content='''Actúa como un ingeniero de
software.'''),

HumanMessage(content='''¿Cuáles son las ventajas del uso de Python frente a
Java?''')]

response = chat.invoke(input=messages)

print(response.content)
```

El **parámetro de temperatura** influye en el grado de **determinismo o creatividad** de la respuesta. El *system message* se utiliza para dar instrucciones al chatbot acerca de la manera en que debe responder. Por último, la función `invoke()` envía el *prompt* y recibe la respuesta.

### Técnicas avanzadas

#### **Streaming** de respuestas

ChatGPT escribe las respuestas de forma **secuencial**, en lugar de presentar el texto completo una vez generado todo el contenido. Esta técnica permite evitar tiempo de espera al usuario, porque la información se visualiza a medida que se va generando. Esta técnica puede emularse desde LangChain con la función `stream()` :

```
for chunk in chat.stream(messages):  
  
    print(chunk.content, end="", flush=True)
```

#### **Paralelización** de respuestas

LangChain ofrece la opción de **paralelizar el envío** de *prompts* para generar respuestas de manera más ágil, sin tener que esperar a la recepción de una respuesta para generar el siguiente *prompt*.

Por ejemplo, en el siguiente fragmento de código paralelizamos la generación de respuestas con un máximo de cinco peticiones simultáneas:

```
from langchain_core.runnables.config import RunnableConfig  
  
config = RunnableConfig(max_concurrency=5)  
  
results = chat.batch([messages, messages], config=config)
```

## Definición de plantillas

LangChain proporciona la técnica de **apantillado** para poder reutilizar un *prompt* con diferentes valores de uno o varios parámetros. Esta técnica facilita la reutilización y simplificación del código fuente que accede a LLMs.

Por ejemplo:

```
alimento = "Plátano"
```

```
prompt = f"¿Cuáles son las principales propiedades nutritivas del  
{alimento}?"
```

## Generación de *pipelines*

LangChain permite definir *pipelines*, es decir, secuencias de acciones que se ejecutan una a continuación de la otra, utilizando la salida de la anterior. Para ello, se utiliza el operador « | ». Esta funcionalidad permite estructurar más fácilmente las interacciones con los LLMs.

Por ejemplo:

```
model = ChatOpenAI()
```

```
system = SystemMessagePromptTemplate.from_template(plantilla)
```

```
prompt = ChatPromptTemplate.from_messages([system])
```

```
chain = prompt | model
```

## Extracción de datos estructurados

LangChain permite extraer **datos estructurados** de una respuesta, tales como: fechas, listas o documentos XML. Para ello se utilizan dos funciones: `get_format_instructions()`, que incorpora al *prompt* las instrucciones necesarias para conseguir el formato de salida deseado; y `parse()`, que se encarga de extraer los datos estructurados de la respuesta.

También puede utilizarse la librería de validación de datos **Pydantic**. En el siguiente ejemplo, se extraen los datos de nombre y edad de los empleados de una empresa a través de un *prompt* que incluye esta información en lenguaje natural:

```
from langchain.chains.openai_tools import create_extraction_chain_pydantic

from langchain_openai.chat_models import ChatOpenAI

from langchain_core.pydantic_v1 import BaseModel, Field


class Empleado(BaseModel):

    nombre: str = Field(..., description="Nombre del empleado")

    edad: int = Field(..., description="Edad del empleado")

model = ChatOpenAI(model=[MODELO])

chain = create_extraction_chain_pydantic(Empleado, model)

chain.invoke({'input': '''Juan tiene 43 años. Trabaja en la oficina de
Madrid como supervisor. Susana tiene 25 años. Trabaja en la oficina de
Barcelona como gerente.'''})
```

El resultado obtenido es el siguiente:

```
[Empleado(name='Juan', age=43), Empleado(name='Susana', age=25)]
```

## Llamada a funciones

Los modelos de OpenAI pueden generar como **respuesta** la llamada a una función. Esta característica puede utilizarse desde LangChain, definiendo la función o funciones que pueden ser llamadas como respuesta a un *prompt* y pasándolas como argumento en `invoke()`.

A continuación, se muestra un ejemplo con código (abreviado) en Python. En primer lugar, se define el *prompt*; a continuación, se incluyen las definiciones e implementaciones de las funciones; por último, se envía el *prompt* con la definición de las funciones como parámetro y se ejecuta la función correspondiente si la respuesta incluye una llamada a función.

```
mensajes = [

    {

        "role": "user",

        "content": '''Reserva una mesa en el restaurante Asador Madrid el
2024-04-19 a las 14:00 para dos personas.'''

    }

]

def funcion1():

    [IMPLEMENTACIÓN]

funciones_json=[DEFINICIÓN]

funciones_implementacion= { 'nombre1': funcion1 }

response = client.chat.completions.create(

    model="gpt-3.5-turbo-1106", messages=mensajes, tools=funciones_json,
```

```
)

response = response.choices[0].message

if response.tool_calls:

    referencia_funcion = response.tool_calls[0]

    nombre_funcion = referencia_funcion.function.name

    funcion = funciones_implementacion(nombre_funcion)

    retorno = funcion(**argumentos)
```

### Análisis de documentos

En ocasiones, es necesario llevar a cabo un proceso de *fine-tuning* de los modelos de OpenAI para que adquiera una nueva base de conocimiento. Para ello, se le debe facilitar datos acerca de la temática en cuestión. LangChain facilita esta tarea, proporcionando clases que pueden extraer el contenido de un documento.

Por ejemplo, el siguiente código en Python convierte en texto todas las páginas de un documento PDF:

```
from langchain_community.document_loaders import PyPDFLoader

loader = PyPDFLoader("procedimiento_interno_soporte_usuarios.pdf")

pages = loader.load_and_split()
```

## 6.4. Uso de FAISS y Pinecone

### Conceptos generales

Una **base de datos de vectores**, o *vector store*, almacena la representación numérica (*embeddings*) de palabras o textos. Esta representación alternativa permite realizar búsquedas de **información relacionada** o detectar similitudes entre documentos.

El uso de **información numérica** representa un avance respecto a técnicas tradicionales, como *keyword matching*, en la que la lista de resultados de búsqueda se basa en que las *keywords* especificadas por el usuario estén presentes de forma exacta en los posibles resultados. Mediante el uso de *embeddings*, en cambio, es posible hacer *match* de términos diferentes, pero que tienen el mismo significado — puesto que tendrán valores numéricos similares—.

Por otro lado, las bases de datos de vectores son parte integral de la **retrieval augmented generation** (RAG). El RAG implica que se realiza una búsqueda de documentos similares al *prompt* del usuario y se inyectan como contexto en el *prompt*. Esta técnica permite a un chatbot recordar las interacciones anteriores de manera eficiente, es decir, sin tener que almacenar y recuperar todas ellas en cada nueva interacción.

### Uso de FAISS

FAISS es una librería desarrollada por Facebook para realizar búsquedas de información similar y agrupación (*clustering*) de *embeddings*. En el siguiente ejemplo se muestra cómo utilizar FAISS para obtener resultados de búsqueda apropiados para una consulta del usuario:

```
import numpy as np

import faiss
```



```
embeddings = [ get_embeddings(fragmento) for fragmento in fragmentos_texto
]

vectores_texto = np.array(embeddings)

index = faiss.IndexFlatL2(vectores_texto.shape[1])

index.add(vectores_texto)

def busqueda_de_vectores(query, k=1):

    vectores_query = get_embeddings(query)

    distancias, indices = index.search(np.array([ vectores_query ]), k)

    return [(fragmentos_texto[i], float(d)) for d,i in zip(distancias[0],
indices[0])]

query = "¿Qué tipos de flores existen?"

resultados = busqueda_de_vectores(query)

print(resultados)
```

En primer lugar, se obtienen los *embeddings* de cada fragmento de un texto. Cada fragmento contiene un tipo de información diferente. A continuación, se genera un índice en FAISS para agilizar el *retrieval*. La función `busqueda_de_vectores()` obtiene los *embeddings* de la consulta del usuario y busca en el índice el fragmento de texto con mayor similitud ( $k=1$ ). Por último, se realiza una llamada a esta función con una consulta determinada ( “¿Qué tipo de flores existen?”) y se imprime el resultado, que incluye el grado de similitud entre el fragmento y la consulta.

Este resultado puede inyectarse como contexto a un *prompt* de OpenAI o cualquier otro LLM para incrementar la precisión de la respuesta.

### Uso de Pinecone

Pinecone es un servicio gestionado en la **nube de vector store**. Incorpora todas las ventajas propias de la computación en la nube: no es necesario provisionar ni mantener infraestructura, proporciona la escalabilidad necesaria, el nivel de servicio es elevado e implementa los mecanismos de seguridad adecuados. Los **índices** de Pinecone pueden desplegarse en cualquiera de los principales proveedores de nube: Azure, AWS, Google Cloud.

En el siguiente ejemplo, se muestra cómo crear un **índice** en Pinecone con Python:

```
from pinecone import Pinecone, ServerlessSpec

import os

os.environ["PINECONE_API_KEY"] = [API_KEY]

index_name = "indice1"

environment = "us-east-1"

pc = Pinecone()

if index_name not in pc.list_indexes().names():

    pc.create_index(

        index_name,

        dimension=1536,

        metric="cosine",

        spec=ServerlessSpec(cloud="aws", region=environment),
```

```
)  
  
index = pc.Index(index_name)  
  
index.describe_index_stats()
```

En primer lugar, se define la variable de entorno con la API key , el nombre del índice y la región en la que se despliega. A continuación, se crea una instancia de Pinecone y se comprueba que el índice que queremos crear no existe aún. En ese caso, se crea en AWS con vectores de dimensión 1536, que es la misma que utiliza el modelo textembedding-ada-002 de OpenAI. Por último, se obtiene una referencia al índice recién creado y se imprimen los detalles de este.

## 6.5. Agentes autónomos

### Conceptos generales

Los chatbots como ChatGPT implementan el concepto de **cadena de razonamiento**, o *chain of thought* (CoT), para descomponer un problema complejo en tareas sencillas que pueden ser procesadas de forma individual por el LLM. Este concepto se puede hacer realidad mediante **agentes**, que son sistemas que ejecutan acciones, observan el resultado y deciden a partir de él qué acción debe ejecutarse a continuación.

En el ámbito de los **LLMs**, un agente resuelve una tarea (*prompt*) con las acciones disponibles (*tools*). Además, los agentes disponen de: estrategias de ejecución, memoria (a través de *vector stores* y *embeddings*) y posibilidad de consultar fuentes de información externas.

### Implementación de agente con ReAct

ReAct es un *framework* basado en descomposición de tareas, razonamiento iterativo y otras herramientas necesarias para que un agente pueda interpretar el entorno y tomar decisiones hasta completar una tarea.

Una **implementación** típica de ReAct con LLMs consiste en:

- ▶ Identificar qué herramienta (*tool*) quiere utilizar el LLM en cada paso y los parámetros que necesita dicha herramienta. Una *tool* es una función a la que puede llamar el LLM para resolver una tarea concreta. Cada *framework* implementa el concepto de herramienta de una forma diferente. Por ejemplo, en LangChain está disponible a través de la clase `Tool`.
- ▶ Comprobar si el LLM considera que ha encontrado la respuesta (es decir, ha completado la tarea). En ese caso, la ejecución ha terminado.

LangChain implementa un mecanismo de agentes con ReAct. En el siguiente ejemplo se muestra una implementación sencilla:

```
from langchain.agents import AgentExecutor, create_react_agent

from langchain import hub

from langchain_openai import ChatOpenAI

from langchain.tools import Tool

model = ChatOpenAI()

def cuenta_caracteres(string):

    return len(string)

herramientas = [

    Tool.from_function(

        func=cuenta_caracteres,

        name="Cuenta caracteres",

        description="Cuenta el número de caracteres en una cadena de texto",

    )

]

prompt = hub.pull("hwchase17/react")
```

```

agente = create_react_agent(model, herramientas, prompt)

ejecutor_agente = AgentExecutor(agent=agente, tools=herramientas,
verbose=True)

ejecutor_agente.invoke({"input": "'¿Cuántos caracteres tiene la palabra
'hola'''})

```

En primer lugar, se instancia un chat con OpenAI y se definen las herramientas disponibles —en este caso, una función sencilla que cuenta el número de caracteres—. A continuación, se consigue un puntero al *prompt* basado en ReAct. Se crea un agente a partir del modelo (ChatOpenAI), las herramientas y el *prompt*, y un ejecutor de este agente. Finalmente, se lanza una pregunta que puede resolverse con las *tools* proporcionadas.

## Implementación de agente con OpenAI functions

OpenAI proporciona un paradigma distinto a ReAct para la implementación de agentes, denominado **OpenAI functions**.

LangChain soporta la implementación de agentes basada en funciones OpenAI. En el siguiente ejemplo, se muestra una implementación sencilla:

```

from langchain.chains import (

    LLMChain,

)

from langchain import hub

from langchain.agents import create_openai_functions_agent, Tool,
AgentExecutor

from langchain_openai.chat_models import ChatOpenAI

model = ChatOpenAI()

```

```
prompt = hub.pull("hwchase17/openai-functions-agent")

llm_math_chain = LLMMathChain.from_llm(llm=model, verbose=True)

def busqueda_google(query: str) -> str:

    return "Pedro Fernández tiene 35 años."

herramientas = [

    Tool(

        func=llm_math_chain.run,

        name="Calculadora",

        description="Responde preguntas de matemáticas",

    ),

    Tool(

        func= busqueda_google,

        name="Búsqueda en Google",

        description="Útil para averiguar la edad de una persona.",

    ),

]

agente = create_openai_functions_agent(llm=model, tools= herramientas,
prompt=prompt)

ejecutor_agente = AgentExecutor(agent= agente, tools= herramientas,
verbose=True)

result = ejecutor_agente.invoke(
```

```
{
    "input": """Task: Busca en Google la edad de Pedro Fernández.

    A continuación elévala al cuadrado."""}
}

print(result)
```

En primer lugar, se inicializan el **modelo** y el **prompt** como en el caso anterior. La clase `LLMMathChain` se utiliza como herramienta (*tool*) para realizar cálculos, mientras que la función `busqueda_google()` simula una búsqueda en Google con las *keywords* facilitadas como parámetro.

A continuación, se definen las **dos herramientas** que puede utilizar el LLM: la primera, que permite hacer cálculos, y la segunda, que es la función de búsqueda en Google, si bien en este caso la definimos como herramienta útil para averiguar la edad de una persona.

El siguiente paso es crear un **agente** basado en funciones OpenAI con `create_openai_functions_agent()`, que recibe las herramientas previamente definidas, y un ejecutor de agentes basado en dicho agente. Finalmente, se lanza un *prompt*, que es resuelto por el LLM a partir de las dos herramientas facilitadas.

La **salida** es la siguiente:

Pedro Fernández tiene 35 años. El cuadrado de este valor es 1225.



### Conclusiones

Los dos ejemplos anteriores ilustran escenarios sencillos. Sin embargo, la resolución de un *prompt* puede conllevar secuencias de razonamiento y herramientas mucho más complejas.

Dependiendo de la **dificultad** de nuestro escenario, debemos utilizar una u otra aproximación. Para casos de uso sencillos que solo involucran búsquedas o extracciones de datos, OpenAI functions es la mejor elección. Sin embargo, para casos de uso más elaborados, con múltiples herramientas involucradas y la necesidad de un análisis detallado de las acciones anteriores y sus resultados, el *framework* ReAct es la mejor opción.

## 6.6. Generación de imágenes con diffusion models

### Conceptos generales

Los modelos basados en **stable diffusion** utilizan el siguiente mecanismo: primero, se añade ruido a una imagen, a continuación, se trata de recomponerla durante el proceso de entrenamiento, utilizando un texto descriptivo de la imagen como parámetro. Este texto ha sido previamente transformado en *embeddings* para que su representación numérica pueda efectivamente intervenir en el proceso de reconstrucción de la imagen original.

Al incorporar el texto como condicionante del proceso de entrenamiento, la imagen queda vinculada a este. De esa forma, cuando se reciba un *prompt*, es posible utilizar esta vinculación para determinar qué combinación de imágenes dará como resultado la que ha solicitado el usuario. En la Figura 3 se ilustra el proceso:

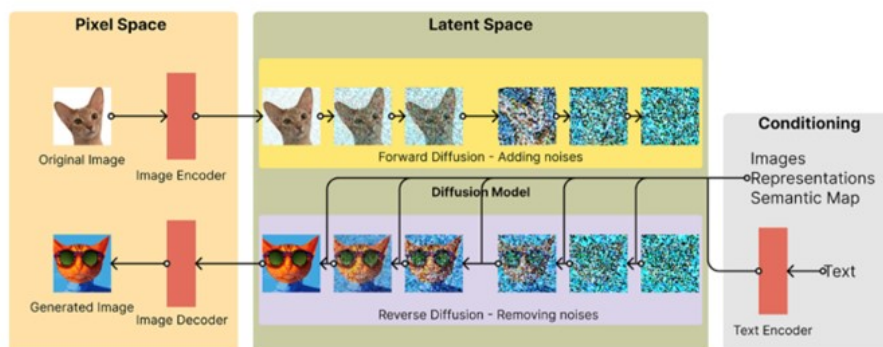


Figura 3. Proceso de reconstrucción de imágenes con *stable diffusion*. Fuente: Zhao, 2023.

Por tanto, ahora el usuario puede escribir la descripción de una imagen que incluya elementos de varias imágenes con las que ha sido entrenada y obtener una nueva imagen con dichos elementos perfectamente integrados. Los principales modelos implementan generación de imágenes a partir de un texto o *prompt* son DALL-E, de OpenAI y Midjourney.

## Ingeniería de prompts

A continuación, se describen algunas **técnicas** de *prompting* para generar imágenes con Midjourney.

En primer lugar, es posible especificar el **formato de la imagen**: imagen de banco de fotografías, pintura con diferentes estilos, dibujos animados, etc.

Por ejemplo:

Prompt: imagen de banco de fotos en la que aparezcan varias personas reunidas en una oficina

Prompt: pintura al óleo en la que aparezcan varias personas reunidas en una oficina

También es posible especificar el **nombre de un artista** cuyo estilo queremos imitar en la composición de la nueva imagen:

Prompt: imagen de un automóvil con el estilo de Andy Warhol

De hecho, se puede **combinar** el estilo de dos artistas, indicando el peso que debería tener el estilo de cada uno de ellos:

Prompt: imagen de un automóvil con el estilo de Warhol::0.8 y Picasso::0.2

En otros casos, la necesidad es generar un *prompt* a partir de una imagen. Esta imagen puede ser original o generada por IA. A esta técnica se la conoce con el nombre de **ingeniería inversa**. Midjourney soporta esta funcionalidad con el comando `/describe`. El *prompt* proporcionado puede utilizarse para generar una imagen similar a la original. Como alternativa, es posible especificar como *prompt* una imagen para que genere una similar, ahorrando el paso intermedio de generación de la descripción.

Por último, existe la opción de especificar lo que no queremos que aparezca en la imagen. A esta técnica se la denomina ***negative prompting***.

## 6.7. Generación de imágenes con Midjourney

En esta sección se describen otras técnicas útiles para explotar al máximo el potencial de Midjourney, con el propósito de completar la visión esbozada en la sección anterior sobre las posibilidades de un modelo de generación de imágenes.

Es importante resaltar que, si bien la apariencia de los *prompts* en diferentes plataformas es muy similar, cada modelo dispone de **características específicas** que introducen mejoras respecto a la funcionalidad estándar. Por ello, es importante conocer qué elementos ofrece cada plataforma y de qué forma pueden ser explotados desde los *prompts*.

### **Quality boosters**

Para aumentar la **calidad de la imagen** resultante, pueden utilizarse los *quality boosters*. Por ejemplo:

Prompt: imagen de una nave especial, que esté de moda en ArtStation.

**ArtStation** es un mercado de activos digitales que incluye imágenes. Como DALL-E y Midjourney han sido entrenados con estas imágenes, son capaces de **imitar** el estilo de los artistas más populares.

### **Inpainting**

Es una técnica que permite **borrar una parte de una imagen y regenerarla** a partir de un **prompt**. Por ejemplo, si eliminamos la ropa de un personaje, podemos escribir una descripción del tipo de ropa que nos gustaría incorporar, y Midjourney se encarga de generarla e integrarla en la imagen.

## ***Outpainting***

Esta técnica permite incorporar un **espacio adicional** en la imagen y pedirle a Midjourney que rellene ese espacio a partir de la descripción proporcionada. De esta forma, es posible ajustar una imagen a un *aspect ratio* determinado.

## **Consistencia de personajes**

Esta técnica parte de ***inpainting/outpainting*** para generar de manera consistente la imagen de un personaje desde diferentes ángulos. De esta forma, es posible utilizar un mismo personaje en diferentes escenas.

## **Definición indirecta de estilos**

En la sección anterior hemos visto cómo generar una imagen a partir del estilo de un autor. Sin embargo, esto puede ocasionar conflictos con el autor que creó dicho estilo. Para evitarlo, es posible pedirle a ChatGPT que describa el **estilo de un artista**, y después utilizar esa descripción para reemplazar el nombre del artista en el *prompt* a Midjourney.

## **Bases de datos de prompts**

Para lograr los mejores resultados en el proceso de generación de imágenes, resulta clave explotar el **conocimiento colectivo** de los miles de personas que escriben *prompts* cada día. Con este propósito se han creado **comunidades** en Internet que incluyen *prompts* y las imágenes generadas a partir de ellos. Las más relevantes son **Lexica.art** y la comunidad en **Discord** de Midjourney.

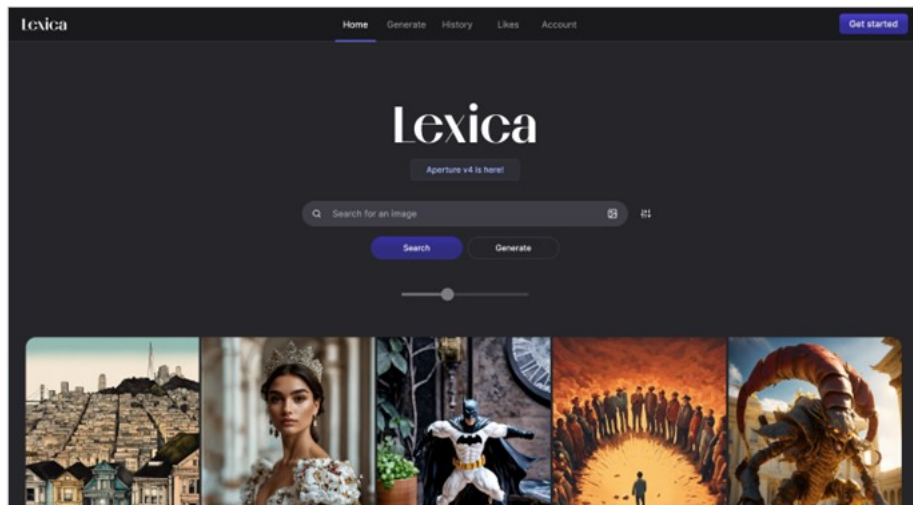


Figura 4. Página principal de Lexica.art. Fuente: Léxica, s. f.

## 6.8. Técnicas avanzadas de generación de imágenes con Stable Diffusion

En los ejemplos de esta sección, se utiliza Stable Diffusion XL, un modelo *open-source*, que puede utilizarse de forma gratuita en un PC con *hardware* GPU.

### Python SDK

Es posible acceder de forma programática a Stable Diffusion XL, utilizando el SDK de Python. Para ello, primero hay que instalar las siguientes **dependencias**:

```
pip install diffusers==0.11.1
```

```
pip install transformers scipy ftfy accelerate
```

Una vez completada la instalación, el siguiente paso es definir el *pipeline* de inferencia a partir del modelo preentrenado. Este *pipeline* define aquello que se ejecuta cuando se solicita la generación de una imagen a partir de un *prompt*.

```
import torch
```

```
from diffusers import StableDiffusionPipeline
```

```
pipeline = StableDiffusionPipeline.from_pretrained("CompVis/stable-diffusion-v1-4", torch_dtype=torch.float16)
```

```
pipeline = pipeline.to("cuda")
```

A partir de este momento, es posible empezar a generar imágenes:

```
prompt = "fotografía de un astronauta montado en un caballo"
```

```
generador = torch.Generator("cuda").manual_seed(1024)
```



```
imagen = pipeline(prompt, num_inference_steps=50, guidance_scale=7,  
generator=generador).images[0]
```

```
imagen.save(f"astronaut_rides_horse.png")
```

El parámetro `guidance_scale` se utiliza para determinar la influencia del *prompt* en la generación de la imagen.

### Interfaz web

**AUTOMATIC1111** es una interfaz web para utilizar Stable Diffusion. Incorpora las extensiones desarrolladas por los usuarios para Stable Diffusion, por lo que ofrece un amplio abanico de funcionalidades que no están disponibles en el servicio original. Por ejemplo, es posible subir una imagen para que se combine con el *prompt*, de manera que la imagen resultante esté influenciada por ambos. Esta funcionalidad se denomina **Img2Img**.

También es posible generar múltiples imágenes con **diferentes valores de un parámetro**. De esta forma, se determina la influencia del parámetro y es posible elegir el valor que proporciona el resultado óptimo. Sería el equivalente al ajuste de hiperparámetros que se realiza en el entrenamiento de otros tipos de modelo. Otra opción es escalar la imagen para que tenga mejor resolución. En la Figura 5 se muestra un ejemplo:



Figura 5. Proceso de *upscaling* en AUTOMATIC1111. Fuente: Barretto, 2023.

AUTOMATIC1111 también implementa la técnica de ingeniería inversa vista en la sección anterior. Para utilizarla, es necesario hacer clic en el botón «Interrogate», como se muestra en la Figura 6:

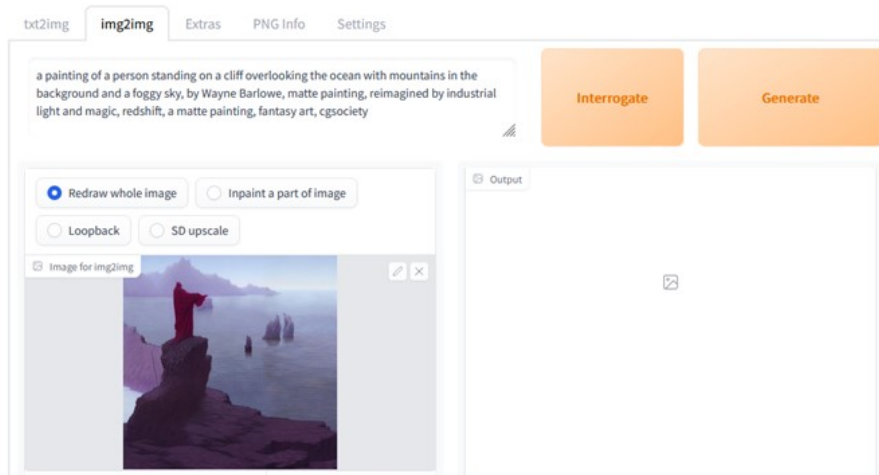


Figura 6. CLIP interrogator en AUTOMATIC1111. Fuente: Automatic, s. f.

También soporta las técnicas de ***inpainting*** y ***outpainting*** vistas en Midjourney para rellenar zonas de la imagen borradas o añadidas a partir de un nuevo *prompt*.

Por último, es posible incorporar controles o condiciones adicionales en la imagen resultante con la extensión **ControlNet**. Cada funcionalidad de ControlNet está asociada a un **modelo preentrenado**.

Algunos de los **modelos** más populares son:

- ▶ **Detección de lados: Canny edge, Hough lines, HED Edge.** Todos ellos identifican los lados de los objetos que aparecen en una imagen, lo que permite reemplazar fácilmente objetos en ella. Por ejemplo, puede utilizarse para aplicar la pose de un personaje a otro o sustituir el escenario que aparece detrás.
- ▶ **User scribble.** Permite generar una imagen a partir de un *prompt* y un esbozo dibujado a mano.
- ▶ **Normal maps.** Permite realizar modificaciones a una imagen, relacionadas con la textura, iluminación y superficies.

## 6.9. Cuaderno de ejercicios

1. Escribir el código fuente para generar un *prompt* y leer la respuesta de GPT en LangChain , ejecutando los siguientes pasos:

- a. Importar las clases necesarias.
- b. Definir la variable de entorno `OPENAI_API_KEY` .
- c. Instanciar la clase de LangChain para utilizar modelos de chat de OpenAI.
- d. Crear un *prompt* con un *system message* que configure un chat informal y un *human message* que pregunte cuál de dos deportistas de élite ha obtenido más triunfos.
- e. Imprimir la respuesta de OpenAI al *prompt*.

```
from langchain.llms import OpenAI

from langchain.chat_models import ChatOpenAI

from langchain.schema import (AIMessage, HumanMessage, SystemMessage)

import os

os.environ["OPENAI_API_KEY"] = "[API key]"

chat = ChatOpenAI()

messages = [ SystemMessage(content="You are a friendly, informal
assistant"), HumanMessage(content="Convince me that Djokovic is better than
Federer") ]

print(chat(messages))
```

2. Escribir el código fuente para crear un agente en LangChain y utilizarlo para resolver un *prompt* con las siguientes características:

- ▶ El agente debe ser de tipo `ZERO_SHOT_REACT_DESCRIPTION`.
- ▶ Debe utilizar `gpt-3.5-turbo`.
- ▶ Debe utilizar las herramientas (*tools*) `serpapi` y `llm-math`.
- ▶ *Prompt*: “¿Cuánta energía produjeron las turbinas de viento en 2022?”

```
from langchain.agents import load_tools

from langchain.agents import initialize_agent

from langchain.agents import AgentType

from langchain.llms import OpenAI

import os

os.environ["OPENAI_API_KEY"] = "[API key]"

os.environ["SERPAPI_API_KEY"] = "[API key]"

OpenAI.api_key = "[API key]"

llm = OpenAI(model="gpt-3.5-turbo", temperature=0)

tools = load_tools(["serpapi", "llm-math"], llm=llm)

agent = initialize_agent(tools, llm,
agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)

agent.run("¿Cuánta energía produjeron las turbinas de viento en 2022?")
```

3. Escribir el código fuente para generar un *prompt* LangChain mediante una plantilla, con las siguientes características:

- ▶ Plantilla: "¿Cuál es un buen título para un/una {medio} sobre {tema}?"
- ▶ Temperatura: 0.9 .
- ▶ Chain: LLMChain .
- ▶ Parámetros: "película de acción", "el futuro" .

```
from langchain.llms import OpenAI

from langchain.prompts import PromptTemplate

from langchain.chains import LLMChain

import os

os.environ["OPENAI_API_KEY"] = "[API key]"

llm = OpenAI(temperature=0.9)

prompt = PromptTemplate(input_variables=["medio", "tema"], template="¿Cuál es un buen título para un/una {medio} sobre {tema}?" )

chain = LLMChain(llm=llm, prompt=prompt)

print(chain.run({ 'medio': "película de acción", 'tema': "el futuro" })))
```

4. Escribir el código fuente para crear un agente de tipo Pandas *dataframe* en LangChain con las siguientes características:

- ▶ CSV: `titulos_género_año_peliculas.csv`
- ▶ *Prompt*: “¿En qué año se lanzaron la mayoría de las comedias?”
- ▶ Temperatura: 0.0.
- ▶ Modelo: OpenAI .

```
from langchain.agents import create_pandas_dataframe_agent

from langchain.chat_models import ChatOpenAI

from langchain.agents.agent_types import AgentType

from langchain.llms import OpenAI

import pandas as pd

import os

os.environ["OPENAI_API_KEY"] = "[API key]"

df = pd.read_csv("títulos_género_año_peliculas.csv")

agent = create_pandas_dataframe_agent(OpenAI(temperature=0), df,
verbose=True)

agent.run("¿En qué año se lanzaron la mayoría de las comedias?")
```

5. Escribir el código fuente para crear un índice en FAISS, con *chunks* solapados de un *dataset*, ejecutando los siguientes pasos:

- a. Importar las librerías necesarias.
- b. Utilizar el *dataset*: `databricks/databricks-dolly-15k`. Columna que almacena el contenido: `context`.
- c. Utilizar la clase `HuggingFaceDatasetLoader` para cargar el *dataset*.
- d. Dividir el contenido del *dataset* en *chunks* de 1000 caracteres, con un *overlap* de 150 caracteres, usando `RecursiveCharacterTextSplitter`.
- e. Utilizar el modelo preentrenado `sentence-transformers/all-MiniLM-l6-v2`. Opciones del modelo: `Device = CPU`. Opciones de codificación: `normalize_embeddings = False`.
- f. Generar *embeddings*.
- g. Crear el índice con la función `from_documents()`.
- h. Guardar el índice con el nombre `índice1`.

```
from langchain.document_loaders import HuggingFaceDatasetLoader

from langchain.text_splitter import RecursiveCharacterTextSplitter

from langchain.embeddings import HuggingFaceEmbeddings

from langchain.vectorstores import FAISS

dataset_name = "databricks/databricks-dolly-15k"

page_content_column = "context"
```



```
loader = HuggingFaceDatasetLoader(dataset_name, page_content_column)

data = loader.load()

text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
chunk_overlap=150)

docs = text_splitter.split_documents(data)


modelPath = "sentence-transformers/all-MiniLM-l6-v2"

model_kwargs = {'device':'cpu'}

encode_kwargs = {'normalize_embeddings': False}

embeddings = HuggingFaceEmbeddings(model_name=modelPath,
model_kwargs=model_kwargs, encode_kwargs=encode_kwargs)


try:

    db = FAISS.from_documents(docs, embeddings)

    db.save_local(folder_path="faiss_db", index_name="índice1")

    print("Se ha creado el índice FAISS ")

except Exception as e:

    print("FAISS error \n",e)
```

## 6.10. Referencias bibliográficas

Automatic (s. f.). *Features*. GitHub. <https://github.com/AUTOMATIC1111/stable-diffusion-webui/wiki/Features>

Barretto, S. (2023, noviembre 23). How to use HiRes.fix to upscale your Stable Diffusion images in A1111. *Medium*. <https://medium.com/rendernet/using-hires-fix-to-upscale-your-stable-diffusion-images-8d8e2826593e>

Mrbullwinkle, ChrisHMSFT, PatrickFarley, nitinme, nkgami, learn-build-service-prod[bot], aahill, eric-urban (2024, agosto 7). Azure OpenAI Service quotas and limits. *Microsoft Learn*. <https://learn.microsoft.com/en-us/azure/ai-services/openai/quotas-limits>

Página de Lexica. <https://lexica.art/>

Zhao, G. (2023, mayo 4). How Stable Diffusion works, explained for non-technical people. *Medium*. <https://bootcamp.uxdesign.cc/how-stable-diffusion-works-explained-for-non-technical-people-be6aa674fa1d>

### ChatGPT prompt engineering for developers

DeepLearning.AI (2024, abril 15). ChatGPT Prompt Engineering for Developers. <https://www.deeplearning.ai/short-courses/chatgpt-prompt-engineering-for-developers/>

En este curso, un desarrollador de *software* puede profundizar en las técnicas de ingeniería de *prompts* específicas para la creación de aplicaciones, incluyendo la construcción de chatbots avanzados. Se muestran ejemplos prácticos, utilizando la API de OpenAPI.

### Prompt engineering for ChatGPT

Vanderbilt University y White, J. (s. f.). *Prompt engineering para Chatgpt*. Coursera.  
<https://www.coursera.org/learn/prompt-engineering>

Este curso de Coursera, creado por la Vanderbilt University, el alumno puede aprender nuevas técnicas de ingeniería de *prompts* para cubrir los casos de uso más avanzados. De esta forma, puede ampliar la información proporcionada en este tema y adquirir más experiencia práctica en el uso de LLMs.

1. Proporcionar contexto a ChatGPT durante la escritura de un *prompt* permite:
  - A. Obtener una respuesta más rápidamente.
  - B. Obtener una respuesta más precisa.
  - C. Obtener una respuesta más completa.
  - D. Es indiferente, ChatGPT no considera la información de contexto proporcionada por el usuario.
  
2. ChatGPT puede utilizarse como asistente para la creación de una aplicación de *software*. En particular, puede ayudarnos con:
  - A. Solamente el esbozo de una arquitectura de alto nivel.
  - B. La arquitectura y la implementación del código fuente.
  - C. Solo con la implementación del código fuente de una función concreta.
  - D. ChatGPT no puede utilizarse de este modo; para este propósito, deben utilizarse herramientas como GitHub Copilot.
  
3. ChatGPT no es solo un chatbot, sino que puede entrenarse para realizar tareas propias de algoritmos de *machine learning*, como la asignación de una categoría a un texto (clasificación):
  - A. Falso.
  - B. Verdadero.
  
4. El conteo de número de tókenes es un aspecto clave para controlar el consumo. Esta tarea puede realizarse con la librería:
  - A. Midjourney.
  - B. Tiktoken.
  - C. ChatGPT.
  - D. OpenAICounter.

5. LangChain proporciona una clase para interactuar con las capacidades de chatbot de OpenAI. El nombre de esta clase es:

- A. LLMChain.
- B. ChatOpenAI.
- C. PromptTemplate.
- D. OpenAIChatModel.

6. ¿En qué consiste el *streaming* de respuestas tras la recepción y procesamiento de un *prompt* en un LLM?

- A. En imprimir varias respuestas simultáneamente.
- B. En imprimir la salida de una respuesta a medida que se va generando, en lugar de esperar a que se genere en su totalidad para imprimirla.
- C. En imprimir la respuesta a varios usuarios conectados a la misma interfaz.
- D. En imprimir respuestas relacionadas de forma paralela.

7. ¿Qué función cumplen las plantillas en LangChain?

- A. Permiten ejecutar una secuencia de acciones predeterminada.
- B. Permite reutilizar y simplificar el código que accede a los LLMs a través de un *prompt* parametrizable.
- C. Permiten ejecutar una secuencia de acciones de forma dinámica, es decir, considerando la salida de las acciones previas.
- D. Permite convertir a *embeddings* el *prompt* para facilitar su reutilización.

8. ¿Para qué se utiliza la librería Pydantic en LangChain?
- A. Para implementar agentes basados en *tools*.
  - B. Para extraer datos estructurados a partir de un *schema* y un texto en lenguaje natural.
  - C. Pydantic no se utiliza con LangChain, es una librería que sirve para validación de datos en otros entornos, como TensorFlow.
  - D. Para extraer datos no estructurados a partir de un texto en lenguaje natural.
9. FAISS y Pinecone son dos alternativas para implementar *vector stores*, ¿cuál es la diferencia de enfoque entre ambos?
- A. Pinecone es una librería *open-source* que puede ejecutarse en cualquier PC, mientras que FAISS es un servicio gestionado en *cloud*.
  - B. FAISS es una librería *open-source* que puede ejecutarse en cualquier PC, mientras que Pinecone es un servicio gestionado en *cloud*.
  - C. FAISS proporciona un enfoque holístico, es decir, integrar muchas funcionalidades que no están presentes en Pinecone.
  - D. Pinecone tiene un rendimiento muy superior al de FAISS, independiente del entorno de computación.

10. ¿En qué casos debe utilizarse ReAct frente a OpenAI functions?
- A. Cuando el caso de uso que estamos considerando es sencillo y solamente implica extracción de datos y consultas a buscadores.
  - B. Cuando el número de herramientas necesarias es elevado y se requiere un análisis detallado de las acciones anteriores y sus resultados.
  - C. En todos los casos, puesto que ReAct es un *framework* con capacidades muy superiores a las de las funciones de OpenAI.
  - D. En todos los casos, puesto que ReAct es mucho más sencillo de aprender e implementar en LangChain que las funciones OpenAI.