**DOCUMENTO DE DECISIONES DE DISEÑO Y TECNOLOGÍAS SELECCIONADAS**

**PROYECTO:**

**DESARROLLO DE UN CHATBOT CON IA GENERATIVA**

[1. Documento Inicial: Decisiones de Diseño y Tecnologías Seleccionadas 3](#_Toc200472595)

[1.1. Introducción 3](#_Toc200472596)

[1.2. Decisiones Iniciales 3](#_Toc200472597)

[1.2.1. Priorización de precisión sobre velocidad 3](#_Toc200472598)

[1.2.2. Base de datos relacional más conveniente 3](#_Toc200472599)

[1.2.3. Aplicación de arquitectura RAG (Retrieval-Augmented Generation) 3](#_Toc200472600)

[1.2.4. Concurrencia y desempeño 3](#_Toc200472601)

[1.2.5. Resumen de comparación entre tecnologías 4](#_Toc200472602)

[1.2.6 Resumen de tecnologías utilizadas 4](#_Toc200472603)

[1.3. Recopilación de Preguntas y Respuestas de Discord para crear una base de conocimiento 4](#_Toc200472604)

[1.3.1. Tratamiento, filtrado y procesamiento de mensajes 4](#_Toc200472605)

[1.3.2. Modelo de Datos 2](#_Toc200472606)

[1.3.2.1. Diagrama de Entidad - Relación (DER) 2](#_Toc200472607)

[1.3.2.1.1. El uso de campo orden (numerico) para organizar las respuestas 2](#_Toc200472608)

[1.3.2.1.2. Ventajas 2](#_Toc200472609)

[1.3.2.1.3. Desventajas 3](#_Toc200472610)

[1.4. Registro de logs y trazabilidad del procesamiento 3](#_Toc200472611)

[1.4.1. Registros que se generan durante el filtrado 3](#_Toc200472612)

[1.4.2. Registros que se generan durante la clasificación 3](#_Toc200472613)

[1.4.3. Registros generados durante la persistencia 3](#_Toc200472614)

[1.4.4. Registros generados durante la generación de embeddings y la búsqueda semántica 3](#_Toc200472615)

[1.5. Base de datos vectorial y búsqueda semántica 4](#_Toc200472616)

[1.6. Consideraciones adicionales, validaciones y decisiones futuras 6](#_Toc200472617)

[1.6.1. Comparación entre heurísticas y modelo LLaMA IA para detección de preguntas 6](#_Toc200472618)

[1.6.2. Muestreo y validación manual del procesamiento 7](#_Toc200472619)

[1.6.3. Decisiones posibles de diseño a futuro 7](#_Toc200472620)

# Documento Inicial: Decisiones de Diseño y Tecnologías Seleccionadas

## Introducción

El presente documento describe las decisiones de diseño más relevantes tomadas durante la implementación del proyecto “Desarrollo de un Chatbot con IA Generativa”.  
El flujo de trabajo abarca desde la carga de los archivos en formato JSON hasta la búsqueda semántica de los mensajes, e incluye etapas clave como el filtrado, la clasificación, la persistencia en bases de datos y la generación de embeddings.  
A lo largo del documento se detallan las principales elecciones técnicas y las herramientas utilizadas, junto con su respectiva justificación, en función de criterios como eficiencia, claridad, precisión y escalabilidad.

## Decisiones Iniciales

### Priorización de precisión sobre velocidad

En este proyecto se privilegia la precisión de las respuestas del chatbot, incluso a costa de la velocidad. Esta decisión impacta en varias elecciones técnicas:

* Se selecciona ChromaDB como base de datos vectorial por su capacidad de gestionar búsquedas semánticas precisas, descartando alternativas más rápidas como Redis o Celeris.
* Para la sincronización entre la base relacional y la base vectorial, se utilizarán triggers en PostgreSQL, descartando sistemas de mensajería como RabbitMQ o Kafka, por ser innecesariamente complejos para un flujo de datos moderado.
* Se elige no utilizar un servidor web (como FastAPI o Flask), dado que el chatbot funciona de manera exclusiva dentro de Discord.

### Base de datos relacional más conveniente

Ya que en el proyecto se manejan relaciones complejas entre preguntas, respuestas y embeddings, PostgreSQL es la mejor opción para optimizar consultas avanzadas y manejar estructuras complejas (MySQL se descarta porque está optimizado para consultas simples).

### Aplicación de arquitectura RAG (Retrieval-Augmented Generation)

Se plantea un diseño de manera que la información se mantenga actualizada sin depender exclusivamente de datos estáticos:

* Las preguntas y respuestas publicadas en Discord en 2024 se utilizan como base de conocimiento inicial.
* Las nuevas preguntas se van a vectorizar únicamente cuando se encuentren cerradas, asegurando que lo que se almacenará como embedding tendrá contexto completo.
* A medida que se identifiquen nuevas respuestas, se actualizará la base de datos relacional, manteniendo sincronización con la base vectorial en todo momento.

### Concurrencia y desempeño

discord.py es monohilo por lo que se aplicará asyncio para evitar bloqueos en tareas intensivas.

También, se evaluará el uso de multiprocesamiento o multihilos para consultas a bases de datos, aunque estas estrategias se van a considerar para etapas futuras si surgiera necesidad de escalar.

### Resumen de comparación entre tecnologías

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Recurso Evaluado | Opciones | Criterios evaluados | Resultado | Decisión |
| Base vectorial | Redis, Celeris, ChromaDB | Precisión vs velocidad | ChromaDB tiene mayor precisión en búsquedas semánticas | Se elige ChromaDB |
| Comunicación entre bases | RabbitMQ, Kafka, triggers | Complejidad, escalabilidad, necesidad actual | RabbitMQ y Kafka muy complejos para el flujo de datos del sistema | Se eligen triggers |
| Framework web vs API de Discord | FastAPI, Flask, API de Discord | Uso exclusivo en Discord vs necesidad de servidor web | No es necesario servidor externo | Se usa la API de Discord (discord.py) |
| Modelo de lenguaje para detección de preguntas | LLaMA IA vs función heurística basada en patrones (signos, palabras clave) | Precisión vs rendimiento | Resultados similares, pero LLaMA IA es significativamente más lento | Se mantiene la función heurística por eficiencia |
| Base de datos relacional | MySQL vs PostgreSQL | Complejidad de relaciones, performance de consultas | PostgreSQL maneja mejor relaciones n:n y subconsultas complejas | Se elige PostgreSQL |

### Resumen de tecnologías utilizadas

|  |  |
| --- | --- |
| **Tecnología / Herramienta** | **Uso principal** |
| Python + Pandas | Procesamiento de mensajes |
| PostgreSQL | Base de datos relacional |
| ChromaDB | Base vectorial para embeddings |
| HuggingFace (all-MiniLM-L6-v2) | Generación de embeddings |
| Pattern Strategy | Filtrado de mensajes |
| discord.py + asyncio | Integración con Discord y concurrencia |

## Recopilación de Preguntas y Respuestas de Discord para crear una base de conocimiento

### Tratamiento, filtrado y procesamiento de mensajes

En este proyecto se construye una base de conocimiento a partir de una exportación en archivos JSON de los mensajes intercambiados entre alumnos y docentes en Discord.

Se utiliza para analizar y procesar mensajes, el lenguaje de programación Python, junto con la librería pandas, que permite cargar los datos en una estructura de tabla llamada DataFrame.

Esta elección se justifica por las capacidades de pandas para manipular datos estructurados de forma eficiente y flexible, facilitando tareas como filtrado, limpieza y análisis. Estas operaciones serían más complejas si se trabajara directamente con listas o diccionarios de Python.

Una vez cargados los mensajes en el DataFrame, se aplican filtros para eliminar aquellos que no son útiles para construir la base de conocimiento. Los mensajes filtrados son:

* vacíos o compuestos solo por espacios.
* contienen únicamente emojis, GIFs o enlaces (por ejemplo, de Tenor o Giphy).
* formados solo por símbolos o números sin texto (como “+1” o “?”).

Cada tipo de mensaje no útil requiere una lógica de detección particular. Para poder aplicar estas lógicas de manera ordenada, extensible y desacoplada, se implementa el patrón de diseño Strategy.

Diagrama

El contenido generado por IA puede ser incorrecto.

Este patrón permite encapsular cada criterio de filtrado en una clase independiente que implementa una interfaz común llamada EstrategiaFiltro. Esta interfaz define dos métodos:

* aplicar(texto: str) -> bool: determina si un mensaje debe eliminarse según el criterio.
* nombre() -> str: devuelve el nombre del filtro, útil para registrar logs.

La función principal aplicar\_filtros\_mensajes\_json recibe un DataFrame y una lista de estrategias, y aplica cada filtro de forma desacoplada. Los mensajes filtrados se registran y se exportan por separado, permitiendo trazabilidad y análisis posterior.

Una vez aplicado el filtrado, los mensajes restantes son clasificados en preguntas y respuestas. Esta tarea es realizada por una clase llamada Procesador.

Tabla

El contenido generado por IA puede ser incorrecto.

Dado a que el intercambio de preguntas y respuestas puede darse entre varios participantes de ambos roles (alumnos y docentes) y los temas a tratarse pueden mezclarse, se han definido reglas para identificar cuándo un mensaje es pregunta, respuesta y en el último caso, a qué pregunta corresponde y si es de cierre.

Entonces las situaciones que se pueden tener:

* mensaje de un docente:
  + se considera que siempre es una respuesta.
  + se asigna a las preguntas que corresponda o se tendrá un mensaje suelto.

Ejemplo de mensaje suelto de un docente (el primer mensaje del JSON que da la bienvenida a los alumnos al canal de consultas): “Este canal será utilizado para realizar consultas (TPA, temas de la asignatura, parciales, etc.). Esperamos que sea lo más colaborativo posible. Siéntanse libres de contestar las dudas de sus compañeros y/o aportar nuevas ideas 🙂”

* mensaje de alumno:
  + puede ser tanto una pregunta como una respuesta.
  + para determinar en qué categoría corresponde se analiza lo siguiente:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Llega un mensaje de cierto autor para procesar | | | | | | | | |
| ¿Hay preguntas abiertas? | | | | | | | | |
| SI | | | | | | NO | | |
| ¿Hay preguntas del mismo autor del mensaje? | | | | | | ¿Es pregunta? | | |
| SI | | | | NO | | SI | NO | |
| ¿Es parte de la pregunta a analizar?  (la pregunta es del mismo autor del mensaje y no tiene respuestas) | | | | ¿Es pregunta? | | El mensaje es una nueva pregunta | ¿Hay preguntas cerradas? | |
| NO | | | SI | SI | NO |  | SI | NO |
| ¿Es una respuesta de cierre? | | | El mensaje es la continuación de una pregunta (se concatena con ella) | El mensaje es una nueva pregunta | Se agrega respuesta a cada una de las que está en preguntas abiertas |  | El mensaje se asigna como respuesta a la última o las últimas dos preguntas cerradas | El mensaje del alumno queda suelto ya que no hay lista de preguntas abiertas y cerradas |
| SI | | NO |  |  |  |  |  |  |
| ¿Tiene Respuesta Validada? | | Es una respuesta a la pregunta |  |  |  |  |  |  |
| SI | NO |  |  |  |  |  |  |  |
| Se cierra pregunta, se guarda en lista de preguntas cerradas, se quita de preguntas abiertas | Es una respuesta a la pregunta |  |  |  |  |  |  |  |

La clase Procesador mantiene una relación de asociación con las clases Mensaje, Pregunta y Respuesta, ya que las utiliza para llevar a cabo la clasificación de los datos extraídos desde archivos JSON.

Aunque las clases Pregunta y Respuesta comparten algunos atributos con la clase Mensaje (como autor, contenido, id, attachments, timestamp y origen), se decidió no aplicar herencia, ya que sus comportamientos son diferentes y su finalidad dentro del procesamiento también lo es.

Esta decisión se alinea con el principio de responsabilidad única de la programación orientada a objetos, que sugiere que cada clase debe tener una única responsabilidad clara.

Una captura de pantalla de un celular con letras

El contenido generado por IA puede ser incorrecto.

Se tiene una lista de instancias de la clase Procesador (una por cada JSON analizado), que consta de:

* la lista de pregunta cerradas, con sus respuestas (necesarias para la persistencia de datos en la base de datos relacional que se implementa en postgreSQL).
* lista de preguntas abiertas (en este caso no debe contener elementos).
* contador de preguntas, cantidad de concatenaciones, cantidad de mensajes de cierre de alumnos y contador de mensajes que son respuestas. Son necesarios para la trazabilidad ya que permite determinar la cantidad total de mensajes analizados.

En cada caso, se va a determinar si:

* una pregunta tiene o no contexto (es decir, es una pregunta muy corta, menos de 6 palabras)
* una pregunta es administrativa (por ejemplo, es una pregunta que se refiere a la hora de un parcial o final o a qué equipo le tienen que corregir el trabajo práctico anual)
* una respuesta es o no corta (se trata de textos que tiene menos de 14 caracteres)

Este análisis es necesario, dado a que las preguntas que sean administrativas y no tengan contexto, no se van a convertir en embeddings (en vectores).

Se procede a conectar la base de datos con Python para que puedan persistirse las preguntas y respuestas.

### Modelo de Datos

#### Diagrama de Entidad - Relación (DER)

Diagrama

El contenido generado por IA puede ser incorrecto.

* mensajes: contiene todos los mensajes extraídos del JSON, exceptuando los mensajes de cierre de alumnos.
* preguntas: incluye únicamente los mensajes identificados como preguntas (relación 1:1 con mensajes).
* respuestas: incluye únicamente los mensajes identificados como respuestas (relación 1:1 con mensajes, n:1 con preguntas).
* autores: representa a todos los usuarios distintos detectados en las conversaciones de Discord (relación 1:n con mensajes).

##### El uso de campo orden (numerico) para organizar las respuestas

Se necesita para garantizar que las respuestas se presenten de manera coherente cuando el chatbot los muestre por pantalla.

En bases de datos relacionales, agregar un campo orden es una de las formas más eficientes de mantener el orden. Con un id\_vinculante que apunte al fragmento anterior, se debe recorrer la base de datos preguntando quién es el siguiente en cada paso, lo que haría que las consultas sean más lentas.

#### Ventajas

* Separación explícita de preguntas y respuestas:
  + Permite aplicar reglas específicas a cada tipo de mensaje.
  + Las validaciones corresponden solo a las respuestas.
  + La clasificación de preguntas (como administrativas o sin contexto) es exclusiva de la entidad pregunta.
  + Los embeddings se generan únicamente a partir de las preguntas.
  + Se puede reconstruir la conversación con claridad, distinguiendo preguntas y respuestas.
* Escalabilidad y mantenibilidad:
  + Si en el futuro se desea incorporar una nueva categoría (por ejemplo, calificaciones, consultas sobre trabajos prácticos, etc.), el modelo lo admite sin grandes refactorizaciones.
  + La estructura resulta clara y entendible para otros desarrolladores.
* Alineación con el flujo del chatbot:
  + El sistema necesita identificar preguntas válidas y sus respuestas correspondientes para pasárselas al modelo Mistral.
  + Tener estas entidades bien diferenciadas simplifica el flujo RAG (Retrieval-Augmented Generation).

#### Desventajas

* Consultas más complejas:
  + Las consultas a la base requieren combinaciones de tablas mediante joins para obtener toda la información relevante.

## Registro de logs y trazabilidad del procesamiento

Se implementa un sistema de logging que permite garantizar la trazabilidad de cada transformación aplicada a los datos. Esta estrategia facilita el análisis exploratorio, el control de calidad del procesamiento y la validación de los resultados.

### Registros que se generan durante el filtrado

Cuando se aplican las reglas de filtrado (mensajes vacíos; solo stickers, emojis o GIFs; solo números; solo símbolos), se genera un archivo CSV específico, para cada una, que almacena los mensajes descartados por ese criterio.

Estos archivos (que se visualizan en Excel) permiten auditar el proceso, identificar posibles errores y ajustar las reglas si fuera necesario.

### Registros que se generan durante la clasificación

Los logs que se originan en la etapa de clasificación (detección de preguntas, respuestas, mensajes de cierre y mensajes concatenados) registran:

* La cantidad total de mensajes procesados.
* La cantidad de respuestas detectadas.
* La cantidad de mensajes de cierre de alumnos.
* La cantidad de mensajes sueltos (respuestas sin pregunta asociada).
* La cantidad de mensajes concatenados (cuando una pregunta se realiza en varios mensajes consecutivos por un mismo autor).
* Cuando se detecta una nueva pregunta o respuesta, el contenido de cada una, el autor y la fecha correspondiente.

Estas decisiones de diseño reflejan una estrategia robusta y auditable, alineada con buenas prácticas de Ingeniería de Software y desarrollo de sistemas de Inteligencia Artificial (IA).

### Registros generados durante la persistencia

Durante la persistencia se generan registros que incluyen: identificador del mensaje, autor, fecha, y adjuntos (en caso de existir).

Esto permite verificar que los datos almacenados coincidan con los procesados y facilita el control del volumen de datos.

### Registros generados durante la generación de embeddings y la búsqueda semántica

Finalmente, se generan dos archivos de log adicionales vinculados al procesamiento semántico:

* Un listado de todas las preguntas utilizadas para generar embeddings, con su ID y contenido.
* Un log de los resultados de la búsqueda semántica, donde se registra:
  + La pregunta consultada.
  + Los cinco resultados más similares.
  + El valor de similitud semántica correspondiente (solo si la similitud fue mayor a cero, o hasta que se alcanzó ese umbral).

Este enfoque permite revisar el rendimiento de la búsqueda semántica y comprobar el grado de similitud entre preguntas, clave para evaluar la eficacia del sistema.

## Base de datos vectorial y búsqueda semántica

Se procede a la generación de embeddings de las preguntas para permitir búsquedas semánticas.

Para esto, se utiliza el modelo all-MiniLM-L6-v2, disponible en la librería HuggingFaceEmbeddings, resultado de un análisis comparativo que consideró precisión semántica, velocidad de procesamiento y costo de cómputo:

|  |  |  |  |
| --- | --- | --- | --- |
| Modelo | Precisión Semántica | Velocidad | Notas |
| all-MiniLM-L6-v2 | Muy buena | Muy rápida | Ideal para búsquedas eficientes |
| all-MiniLM-L12-v2 | Mejor que L6 | Más lenta | Mayor precisión, pero más costoso |
| all-mpnet-base-v2 | Excelente | Más lenta | Alta precisión semántica |
| multi-qa-MiniLM-L6-cos-v1 | Buena | Rápida | Diseñado para recuperación de QA |

Razones para elegir all-MiniLM-L6-v2:

1. Equilibrio óptimo entre velocidad y precisión.
2. Buen rendimiento semántico incluso en recursos limitados.
3. Escalable en producción sin requerir GPU o infraestructura costosa.

Una vez generados los embeddings, se realizan pruebas de búsqueda semántica en Python.  
El sistema permite definir:

* Una pregunta de prueba.
* Un parámetro top-k, que indica cuántas preguntas similares recuperar (usualmente entre 2 y 5).

Si el valor es demasiado bajo, puede omitir resultados relevantes. Si es muy alto, puede incluir preguntas poco relacionadas.

Se muestran únicamente las preguntas cuya similitud semántica es positiva.

Ejemplo de búsqueda semántica:

Texto

El contenido generado por IA puede ser incorrecto.Imagen que contiene Interfaz de usuario gráfica

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.Texto

El contenido generado por IA puede ser incorrecto.

## Consideraciones adicionales, validaciones y decisiones futuras

### Comparación entre heurísticas y modelo LLaMA IA para detección de preguntas

Durante el desarrollo del sistema, se evaluó la posibilidad de utilizar un modelo de lenguaje (LLaMA IA) para asistir en la detección de nuevas preguntas. En particular, se lo probó cuando llegaba un mensaje nuevo y no existía una pregunta abierta en curso, para determinar si dicho mensaje debía tratarse como una pregunta inicial o como una respuesta.

Para esta evaluación, se compararon los resultados del modelo con los obtenidos mediante una función heurística implementada previamente (es\_pregunta), que identifica preguntas a partir de:

* signos de interrogación,
* palabras clave típicas de consultas académicas (como “tengo una consulta”, “tengo una duda”, “cómo”, “cuándo”, “por qué”, “debo”, “necesito”, etcétera).

Los resultados entre ambas aproximaciones fueron prácticamente equivalentes. Sin embargo, el modelo LLaMA requirió aproximadamente 5 minutos para procesar entre 50 y 90 mensajes, mientras que el enfoque basado en reglas llevó menos de un minuto para la misma cantidad.

En base a esta evidencia, se decidió priorizar la función heurística, tanto por su rendimiento como por su simplicidad de integración. El uso de modelos de lenguaje más complejos podría considerarse para futuras versiones, en etapas donde el sistema ya esté estabilizado y se busque aumentar la precisión con técnicas más sofisticadas.

### Muestreo y validación manual del procesamiento

Para verificar la calidad del sistema se tomó una muestra representativa del 33% del total de preguntas detectadas y se analizaron aspectos como:

* Si cada pregunta tenía una respuesta correspondiente.
* Si se detectaron respuestas erróneas (respuestas asociadas a preguntas equivocadas).
* Si había respuestas múltiples o inconsistentes.

Este muestreo permitió ajustar las reglas, verificar la coherencia del sistema y garantizar un funcionamiento confiable antes de pasar a etapas posteriores.

### Decisiones posibles de diseño a futuro

Algunas de ellas incluyen:

* Aplicar un modelo de lenguaje para todo el procesamiento de clasificación de pregunta y respuestas, para luego comparar resultados con la solución aplicada.
* Usar patrones de diseño (como Observer, Factory o Command) para flexibilizar y extender la arquitectura sin comprometer la mantenibilidad.
* Incorporar más validaciones para que la aplicación sea más robusta y descriptiva en los errores o fallas que puedan presentarse.