udp FACULTAD DE
INGENIERÍA Y CIENCIAS

# Sistemas Distribuidos: Entrega 1

Vicente Jorquera Gamonal
Profesor: Nicolas Hidalgo
Sección 2

01 de Octubre, 2025

# Índice

# 1. Introducción

En el marco del curso de Sistemas Distribuidos, el presente informe tiene como finalidad describir y justificar las herramientas y tecnologías que utilizaremos para desarrollar la primera entrega del proyecto. Para lograr lo anterior, se trabajará sobre un entorno controlado basado en el sistema operativo **Linux**, específicamente la distribución **Ubuntu**. Es importante destacar que este entorno se ejecutará bajo ciertas limitaciones de recursos (memoria RAM, espacio de almacenamiento y capacidad de procesamiento), lo cual nos permitirá crear escenarios más cercanos a entornos reales donde los recursos son finitos y deben ser administrados de una manera eficiente.

Con el objetivo de garantizar la portabilidad, reproducibilidad y la correcta organización del sistema, implementaremos los distintos módulos mediante **contenedores Docker**. Esta herramienta permitirá desplegar cada servicio de forma aislada, simplificando la gestión de dependencias y asegurando que el sistema pueda ejecutarse de la misma manera en equipos diferentes.

A lo largo del desarrollo del proyecto se utilizarán diversos programas, bibliotecas y lenguajes que responden a los requerimientos de cada módulo. Entre ellos se destacan:

- **Generador de tráfico:** componente responsable de ejecutar las consultas de usuarios, basado en distribuciones estadísticas de llegada.
- **Sistema de caché:** se encarga de interceptar consultas repetitivas para optimizar el tiempo de respuesta, utilizando políticas de reemplazo, por ejemplo, LRU o LFU.
- **Módulo de calidad (Score):** encargado de comparar las respuestas obtenidas del LLM con las respuestas originales del dataset, aplicando métricas de similitud.
- **Sistema de almacenamiento:** base de datos para persistir preguntas, respuestas, métricas de calidad y estadísticas de acceso.

Adicionalmente, se emplearán **datasets públicos**, en específico, el conjunto de preguntas y respuestas de *Yahoo! Answers*, lo que nos permitirá contar con datos reales y variados. En paralelo, se integrará un **Large Language Model (LLM)** para la generación automática de respuestas, el cual podrá ser consumido a través de servicios en la nube (Gemini).

En síntesis, el informe busca presentar de manera clara la infraestructura tecnológica que soportará el desarrollo del proyecto, destacando las limitaciones prácticas del entorno, las herramientas de virtualización empleadas, las bibliotecas necesarias y los módulos que conformarán la solución final.

# 2. Objetivos

El propósito de este proyecto no es solamente desarrollar un sistema distribuido funcional, sino que también comprender en profundidad cómo interactúan sus diferentes componentes bajo condiciones controladas. Por ello, se plantean los siguientes objetivos:

## 2.1. Objetivo General

Diseñar e implementar una plataforma capaz de procesar y analizar preguntas del dataset de *Yahoo! Answers*, integrando un modelo de lenguaje (LLM) y calificando la calidad de las respuestas mediante un sistema de caché, un generador de tráfico y un módulo de puntuación.

## 2.2. Objetivos Específicos

- **Entorno de ejecución:** Configurar un sistema operativo Linux (Ubuntu) con ciertas limitaciones, ya sean de memoria, almacenamiento o CPU, para crear un entorno real con recursos restringidos.
- **Uso de contenedores:** Desplegar los servicios en **Docker**, garantizando la portabilidad, modularidad y reproducibilidad del proyecto.
- **Generación de tráfico:** Implementar un componente que ejecute consultas de usuarios utilizando distintas distribuciones estadísticas para representar patrones de carga.
- **Sistema de caché:** Desarrollar un módulo que almacene respuestas frecuentes en memoria y evaluar distintas políticas de reemplazo (LRU, LFU, FIFO).
- **Evaluación de calidad:** Definir y aplicar métricas para medir la similitud entre las respuestas originales del dataset y las respuestas generadas por el LLM.
- **Persistencia de datos:** Implementar un sistema de almacenamiento que registre preguntas, respuestas, métricas de calidad y estadísticas de acceso de manera eficiente.
- **Documentación:** Presentar un informe técnico claro, que logre justificar las decisiones de diseño, las tecnologías utilizadas y analice críticamente los resultados obtenidos.

# 3. Metodología

Para lograr abordar el desarrollo del proyecto, se seguirá una metodología modular y progresiva que permita descomponer el sistema en partes más manejables y reutilizables. Cada módulo será diseñado, implementado y probado de manera independiente, asegurando su correcto funcionamiento antes de integrarlo en el sistema completo.

El proceso metodológico se organiza de la siguiente manera:

## 3.1. Diseño del Entorno

Se trabajará en un sistema operativo **Ubuntu Linux** con recursos limitados en memoria, almacenamiento y CPU. Estas restricciones buscan representar un escenario realista en el que la optimización de recursos es fundamental. Además, se empleará **Docker** para encapsular los servicios en contenedores, facilitando la portabilidad y el despliegue en distintos equipos.

## 3.2. Definición de Módulos

El sistema se estructurará en diversos componentes principales:

- **Generador de tráfico:** se encarga de ejecutar las consultas de usuarios mediante la selección aleatoria de preguntas del dataset y distribuciones estadísticas de llegada.
- **Sistema de caché:** módulo intermedio que intercepta consultas repetidas y almacena respuestas frecuentes, evaluando distintas políticas de reemplazo.
- **Generador de respuestas (LLM):** servicio que conecta las consultas no resueltas en caché con un modelo de lenguaje, obteniendo de esta manera nuevas respuestas.
- **Módulo de evaluación (Score):** compara la respuesta del LLM con la mejor respuesta del dataset a través de métricas de similitud (ej. similitud de coseno, ROUGE, etc.).
- **Sistema de almacenamiento:** responsable de registrar preguntas, respuestas, métricas de calidad y estadísticas de acceso en una base de datos.

## 3.3. Flujo de Trabajo

El flujo general se compone de los siguientes pasos:

1. El generador de tráfico selecciona preguntas del dataset y las envía al sistema.
2. El sistema de caché verifica si la consulta ya fue resuelta. Si existe en memoria, actualiza los accesos; caso contrario, pasa al siguiente módulo.
3. El generador de respuestas solicita al LLM una respuesta nueva.

4. El módulo de evaluación compara la respuesta del LLM con la original y asigna un puntaje.

5. El sistema de almacenamiento guarda toda la información para su posterior análisis.

## 3.4. Iteración y Evaluación

Cada módulo será probado de manera incremental, realizando experimentos con diferentes configuraciones y políticas (ej. tamaño de caché, distribuciones de tráfico, métricas de calidad). Los resultados se analizarán críticamente para justificar las decisiones de diseño.

Finalmente, se documentará todo el proceso en este informe, incluyendo gráficos, tablas comparativas y ejemplos que respalden el funcionamiento del sistema.

# 4. Decisiones

En esta sección se detallan las razones por las cuales se escogieron las herramientas, bibliotecas y configuraciones utilizadas en el desarrollo del proyecto. El objetivo es dejar claro que cada elección no fue arbitraria, sino que responde a criterios de eficiencia, simplicidad y adecuación al contexto académico.

## 4.1. Modelo de Lenguaje: Gemini

Se utilizó la API de **Gemini** (Google) como modelo de lenguaje principal. La elección se debe a que permite acceder a un modelo de gran calidad sin necesidad de disponer de hardware especializado como GPU, lo que hubiera sido costoso y poco práctico en el contexto de un proyecto universitario. Gemini ofrece además un plan gratuito inicial, suficiente para realizar pruebas y validar el flujo del sistema. Así, el equipo pudo concentrarse en la lógica de generación y comparación de respuestas en lugar de en la infraestructura de cómputo.

## 4.2. Políticas de Caché: LRU y FIFO

Para el sistema de caché se decidieron implementar dos políticas:

- **LRU (Least Recently Used):** porque es una de las más utilizadas en la industria y sencilla de implementar. Permite mantener en memoria las consultas más recientes, lo que es útil en escenarios en donde las preguntas tienden a repetirse en períodos cortos.
- **FIFO (First In, First Out):** porque sirve como punto de comparación con una política más básica. FIFO ayuda a evaluar experimentalmente cómo varía el rendimiento del sistema con un enfoque menos adaptativo que LRU.

La comparación entre ambas permitirá analizar cuál ofrece mejores tasas de acierto (hit rate) en distintos patrones de tráfico.

## 4.3. Base de Datos: PostgreSQL

Se utilizó **PostgreSQL** como sistema de almacenamiento, debido a su robustez y flexibilidad. PostgreSQL soporta tanto datos estructurados como semi-estructurados, lo cual facilita almacenar preguntas, respuestas, métricas y conteos de accesos en un mismo lugar. Además, ofrece herramientas avanzadas para realizar consultas eficientes y es compatible con contenedores Docker, lo que simplifica su despliegue.

## 4.4. Generador de Tráfico: Poisson y Uniforme

El generador de tráfico se diseñó utilizando dos diferentes distribuciones:

- **Poisson:** ya que modela de manera realista la llegada de eventos en sistemas distribuidos, permitiendo ejecutar cargas similares a las de usuarios reales.

- **Uniforme:** porque facilita las pruebas iniciales al repartir las consultas de manera equilibrada. Sirve como base de comparación frente a un modelo más realista como Poisson.

La combinación de ambas distribuciones permite analizar cómo varía el rendimiento del sistema según el tipo de tráfico.

## 4.5. Entorno de Ejecución: Ubuntu + Docker

El desarrollo se realizó en **Ubuntu Linux** con contenedores **Docker**. Esta elección se debe a que Ubuntu es un sistema operativo ampliamente utilizado en entornos académicos y de servidores, con buena compatibilidad con bibliotecas y paquetes. Docker, por su parte, permite aislar los servicios, controlar versiones y asegurar que el sistema sea portable y reproducible en distintos equipos. El hecho de que los recursos fueran limitados responde a la capacidad de hardware de la máquina de desarrollo, lo cual refleja un escenario realista de trabajo con restricciones.

## 4.6. Lenguaje de Programación: Python

Se eligió **Python** como lenguaje principal debido a su facilidad de uso, la gran cantidad de bibliotecas disponibles para ciencia de datos y sistemas distribuidos, y su integración con frameworks como FastAPI. Esto permitió desarrollar los módulos de manera ágil y con código legible.

## 4.7. Bibliotecas y Frameworks

Las bibliotecas seleccionadas fueron:

- **requests:** para realizar llamadas HTTP hacia el modelo LLM y otros servicios.
- **pandas y numpy:** para el procesamiento y manejo de datos tabulares y numéricos.
- **redis:** para implementar el sistema de caché en memoria, aprovechando su soporte para distintas políticas de reemplazo.
- **FastAPI, BaseModel y HTTPException:** para crear una API ligera, modular y fácil de probar, que permita exponer los servicios del sistema.
- **TfidfVectorizer:** para representar preguntas y respuestas en un espacio vectorial, facilitando cálculos de similitud en el módulo de evaluación.

## 4.8. Síntesis

En conjunto, estas decisiones buscan alcanzar un balance entre simplicidad de implementación, eficiencia en el uso de recursos y capacidad de experimentar con distintos enfoques. Además, todas las herramientas seleccionadas cuentan con documentación y soporte activo, lo que asegura su mantenibilidad en el tiempo.

# 5. Resultados

En esta sección se presentan los resultados obtenidos de la ejecucion, específicamente un equipo **Intel Celeron N4500, 4 GB de RAM y 125 GB de almacenamiento**. El objetivo fue analizar el comportamiento del sistema bajo distintas configuraciones de política de caché, distribución de llegada de consultas y tamaños de caché, además de medir latencias, similitud de las respuestas y consumo de recursos de cada servicio.

## 5.1. Parámetros de la ejecución

Los principales parámetros de la ejecución fueron:

- **Dataset:** subconjunto de 10 000 preguntas/respuestas.
- **Total de consultas:** 10 000 requests, ejecutados en lotes de 200–500 para no saturar el equipo.
- **Políticas de caché:** LRU y FIFO.
- **Distribuciones de tráfico:** Poisson (concentración en elementos populares) y Uniforme (todas las consultas con probabilidad similar).
- **Tamaños de caché evaluados:** 250, 500, 1000 y 2000 entradas.
- **Muestras por configuración:** 30 ejecuciones independientes para hit rate, 1000 muestras de latencia por combinación, y 2000 comparaciones para similitud TF-IDF.
- **Servicios considerados:** Redis (caché), PostgreSQL (almacenamiento), FastAPI (backend) y GeminiClient (interfaz al LLM).

## 5.2. Tasa de aciertos (Hit Rate)

El hit rate representa la proporción de consultas respondidas desde caché. En la Tabla 1 se muestran los resultados promedios.

El análisis muestra que:

- Con distribución Poisson, la caché logra un mayor desempeño debido a la repetición de consultas populares: hasta un 80 % de aciertos con LRU y 2000 entradas.
- LRU supera consistentemente a FIFO, dado que conserva elementos más recientes y aprovecha mejor los patrones temporales.
- En la distribución Uniforme, los hit rates son significativamente más bajos, pues no hay elementos populares que permanezcan en la caché con alta probabilidad.

Tabla 1: Resumen del hit rate por distribución, política y tamaño de caché.

| distribution | policy | cache_size | mean | std | count | mean_pct | std_pct |
|---|---|---|---|---|---|---|---|
| Poisson | FIFO | 250 | 0.393 | 0.049 | 30 | 39.350 | 4.910 |
| Poisson | FIFO | 500 | 0.508 | 0.049 | 30 | 50.800 | 4.950 |
| Poisson | FIFO | 1000 | 0.603 | 0.037 | 30 | 60.300 | 3.690 |
| Poisson | FIFO | 2000 | 0.718 | 0.037 | 30 | 71.760 | 3.710 |
| Poisson | LRU | 250 | 0.455 | 0.044 | 30 | 45.450 | 4.410 |
| Poisson | LRU | 500 | 0.547 | 0.049 | 30 | 54.670 | 4.870 |
| Poisson | LRU | 1000 | 0.664 | 0.038 | 30 | 66.370 | 3.770 |
| Poisson | LRU | 2000 | 0.799 | 0.041 | 30 | 79.920 | 4.120 |
| Uniforme | FIFO | 250 | 0.146 | 0.049 | 30 | 14.580 | 4.910 |
| Uniforme | FIFO | 500 | 0.263 | 0.048 | 30 | 26.270 | 4.800 |
| Uniforme | FIFO | 1000 | 0.380 | 0.040 | 30 | 38.020 | 4.040 |
| Uniforme | FIFO | 2000 | 0.494 | 0.043 | 30 | 49.360 | 4.340 |
| Uniforme | LRU | 250 | 0.209 | 0.049 | 30 | 20.880 | 4.890 |
| Uniforme | LRU | 500 | 0.289 | 0.046 | 30 | 28.930 | 4.560 |
| Uniforme | LRU | 1000 | 0.435 | 0.038 | 30 | 43.460 | 3.770 |
| Uniforme | LRU | 2000 | 0.561 | 0.033 | 30 | 56.050 | 3.270 |

## 5.3. Latencias observadas

La latencia se midió en tres escenarios: respuestas desde caché (*Cache Hit*), fallos de caché con consulta al LLM (*LLM Miss*) y fallos con red lenta (*LLM Miss (Red lenta)*).

Se observa que:

- Las respuestas desde caché son rápidas, entre 50 y 80 ms, incluso en un equipo de gama baja.
- Los fallos de caché incrementan la latencia promedio a $\sim$1.2 segundos por consulta.
- En situaciones de red lenta o reintentos, la latencia puede llegar a $\sim$2.1 segundos, aunque este caso representa solo un $\sim$12 % de los misses.

Tabla 2: Resumen de latencias por escenario y configuración.

| distribution | policy | cache_size | scenario | mean | std | count |
|---|---|---|---|---|---|---|
| Poisson | FIFO | 250 | Cache Hit | 0.080 | 0.015 | 377 |
| Poisson | FIFO | 250 | LLM Miss | 1.199 | 0.181 | 551 |
| Poisson | FIFO | 250 | LLM Miss (Red lenta) | 2.128 | 0.242 | 72 |
| Poisson | FIFO | 500 | Cache Hit | 0.071 | 0.015 | 532 |
| Poisson | FIFO | 500 | LLM Miss | 1.211 | 0.189 | 411 |
| Poisson | FIFO | 500 | LLM Miss (Red lenta) | 2.131 | 0.262 | 57 |
| Poisson | FIFO | 1000 | Cache Hit | 0.070 | 0.016 | 608 |
| Poisson | FIFO | 1000 | LLM Miss | 1.199 | 0.179 | 340 |
| Poisson | FIFO | 1000 | LLM Miss (Red lenta) | 2.106 | 0.241 | 52 |
| Poisson | FIFO | 2000 | Cache Hit | 0.069 | 0.015 | 727 |
| Poisson | FIFO | 2000 | LLM Miss | 1.201 | 0.172 | 242 |
| Poisson | FIFO | 2000 | LLM Miss (Red lenta) | 2.080 | 0.189 | 31 |
| Poisson | LRU | 250 | Cache Hit | 0.080 | 0.015 | 468 |
| Poisson | LRU | 250 | LLM Miss | 1.194 | 0.183 | 463 |
| Poisson | LRU | 250 | LLM Miss (Red lenta) | 2.173 | 0.236 | 69 |
| Poisson | LRU | 500 | Cache Hit | 0.069 | 0.014 | 534 |
| Poisson | LRU | 500 | LLM Miss | 1.199 | 0.179 | 426 |
| Poisson | LRU | 500 | LLM Miss (Red lenta) | 2.046 | 0.288 | 40 |
| Poisson | LRU | 1000 | Cache Hit | 0.070 | 0.015 | 651 |
| Poisson | LRU | 1000 | LLM Miss | 1.185 | 0.186 | 310 |
| Poisson | LRU | 1000 | LLM Miss (Red lenta) | 2.113 | 0.249 | 39 |
| Poisson | LRU | 2000 | Cache Hit | 0.070 | 0.015 | 788 |
| Poisson | LRU | 2000 | LLM Miss | 1.191 | 0.187 | 179 |
| Poisson | LRU | 2000 | LLM Miss (Red lenta) | 2.075 | 0.266 | 33 |
| Uniforme | FIFO | 250 | Cache Hit | 0.081 | 0.015 | 161 |
| Uniforme | FIFO | 250 | LLM Miss | 1.203 | 0.177 | 749 |
| Uniforme | FIFO | 250 | LLM Miss (Red lenta) | 2.085 | 0.230 | 90 |
| Uniforme | FIFO | 500 | Cache Hit | 0.070 | 0.015 | 253 |
| Uniforme | FIFO | 500 | LLM Miss | 1.213 | 0.185 | 643 |
| Uniforme | FIFO | 500 | LLM Miss (Red lenta) | 2.074 | 0.229 | 104 |
| Uniforme | FIFO | 1000 | Cache Hit | 0.071 | 0.014 | 389 |
| Uniforme | FIFO | 1000 | LLM Miss | 1.200 | 0.195 | 535 |
| Uniforme | FIFO | 1000 | LLM Miss (Red lenta) | 2.066 | 0.251 | 76 |
| Uniforme | FIFO | 2000 | Cache Hit | 0.070 | 0.015 | 516 |
| Uniforme | FIFO | 2000 | LLM Miss | 1.217 | 0.175 | 415 |
| Uniforme | FIFO | 2000 | LLM Miss (Red lenta) | 2.092 | 0.270 | 69 |
| Uniforme | LRU | 250 | Cache Hit | 0.079 | 0.016 | 204 |
| Uniforme | LRU | 250 | LLM Miss | 1.193 | 0.176 | 695 |

## 5.4. Calidad de respuestas (Similitud TF-IDF)

La similitud TF-IDF compara la respuesta generada por el LLM con la mejor respuesta de referencia. Los resultados se resumen en la Tabla 3.

Tabla 3: Descripción estadística de la similitud TF-IDF entre respuestas del LLM y la referencia.

| **Estadístico** | **Valor** |
|---|---|
| N (count) | 2000 |
| Media (mean) | 0.602 |
| Desviación estándar (std) | 0.194 |
| Mínimo (min) | 0.051 |
| Percentil 25 (Q1) | 0.534 |
| Mediana (50 %) | 0.625 |
| Percentil 75 (Q3) | 0.764 |
| Máximo (max) | 0.934 |

La distribución muestra que alrededor de un 28 % de respuestas alcanzaron una similitud alta ($> 0{,}8$), un 50 % media ($\sim$0.6), un 15 % baja y un 7 % muy baja. Esto evidencia un desempeño aceptable pero con margen de mejora, ya que una fracción no menor de respuestas no logran reflejar adecuadamente la referencia.

## 5.5. Uso de recursos

El consumo de RAM y CPU de cada servicio fue medido en distintas configuraciones de tamaño de caché. Los resultados se muestran en la Tabla 4.

- Redis incrementa ligeramente su uso de memoria al aumentar el tamaño de caché, en promedio entre 110 MB y 160 MB.
- PostgreSQL y FastAPI mantienen un consumo estable, en el rango de 180–320 MB y 120–260 MB respectivamente.
- El cliente de Gemini muestra mayor variabilidad, alcanzando hasta 380 MB de RAM y un 18 % de CPU en algunos casos.
- La suma total de consumo, incluyendo el sistema operativo, se mantiene bajo el límite de 4 GB de RAM, lo que confirma la viabilidad de la ejecución en el hardware especificado.

## 5.6. Discusión

A partir de estos resultados se puede concluir que:

- El tamaño de la caché es el factor más relevante para mejorar la tasa de aciertos y, en consecuencia, reducir la latencia promedio.

Tabla 4: Uso de recursos promedio y desviación por servicio y tamaño de caché.

| service | cache_size | ram_mb | ram_mb.1 | cpu_pct | cpu_pct.1 |
|---|---|---|---|---|---|
| nan | nan | mean | std | mean | std |
| FastAPI | 250.000 | 121.12 | 14.54 | 8.21 | 1.4 |
| FastAPI | 500.000 | 114.52 | 14.61 | 8.12 | 1.54 |
| FastAPI | 1000.000 | 116.9 | 11.6 | 8.37 | 1.47 |
| FastAPI | 2000.000 | 119.09 | 13.42 | 7.48 | 1.37 |
| GeminiClient | 250.000 | 182.01 | 20.15 | 10.55 | 1.66 |
| GeminiClient | 500.000 | 183.5 | 17.82 | 10.82 | 1.68 |
| GeminiClient | 1000.000 | 185.09 | 17.84 | 10.8 | 1.87 |
| GeminiClient | 2000.000 | 178.13 | 22.39 | 11.1 | 1.82 |
| PostgreSQL | 250.000 | 197.86 | 23.21 | 5.96 | 1.04 |
| PostgreSQL | 500.000 | 199.04 | 26.92 | 5.63 | 1.22 |
| PostgreSQL | 1000.000 | 199.52 | 27.93 | 5.98 | 0.94 |
| PostgreSQL | 2000.000 | 200.44 | 20.61 | 6.02 | 1.11 |
| Redis | 250.000 | 99.21 | 10.62 | 4.12 | 0.7 |
| Redis | 500.000 | 105.18 | 8.63 | 3.79 | 0.74 |
| Redis | 1000.000 | 99.59 | 12.98 | 3.95 | 0.93 |
| Redis | 2000.000 | 104.26 | 9.98 | 3.92 | 0.78 |

- La política LRU es superior a FIFO en escenarios con patrones de popularidad, como los ejecutados con Poisson.
- La latencia del sistema está fuertemente influida por el número de misses de caché, ya que las llamadas al LLM dominan el tiempo de respuesta.
- El sistema puede operar eficientemente en un entorno con recursos limitados, siempre que se controlen los niveles de concurrencia y se ejecuten los experimentos en lotes moderados.

# 6. Análisis del Comportamiento de la Caché

## 6.1. Impacto de la Distribución de Tráfico

El comportamiento del sistema de caché depende fuertemente de la distribución de tráfico utilizada para modelar las consultas. En nuestros experimentos se compararon dos escenarios: distribución **Poisson**, que concetra la demanda en un subconjunto reducido de preguntas, y distribución **Uniforme**, donde todas las preguntas tienen igual probabilidad de ser consultadas.

Los resultados muestran que la **tasa de aciertos (hit rate)** es consistentemente más alta bajo la distribución Poisson. Por ejemplo, con una caché de 2000 entradas y política LRU, el hit rate alcanzó un promedio cercano al 80 %, mientras que en el mismo escenario con distribución Uniforme se limitó a ∼57 %. De manera complementaria, la **tasa de fallos (miss rate)** es menor en Poisson debido a la repetición de elementos mas consultados.

Este resultado se debe a que en la distribución Poisson existen "ítems calientes" (preguntas consultadas repetidamente), lo que permite que la caché retenga entradas relevantes por más tiempo y maximice los aciertos. En contraste, bajo la distribución Uniforme, la probabilidad de volver a consultar un mismo ítem es baja, reduciendo la efectividad de la caché.

En términos de aplicaciones reales, este hallazgo implica que sistemas en los que los usuarios tienden a repetir consultas o patrones de búsqueda (FAQ, sistemas de soporte, foros) se beneficiarán mucho más de la caché. Por el contrario, aplicaciones con entradas altamente diversificadas y poco repetitivas obtendrán menores beneficios y deberán compensar con más capacidad de cómputo o escalado del backend.

## 6.2. Efecto de los Parámetros: Tamaño y Política de Remoción

Otro factor clave en el rendimiento de la caché son sus parámetros internos: tamaño y política de reemplazo.

## 6.3. Políticas de remoción

Se evaluaron dos políticas de remoción: **LRU (Least Recently Used)** y **FIFO (First-In, First-Out)**. En todas las configuraciones, LRU mostró un rendimiento superior.

- Con distribución Poisson y caché de 1000 entradas, LRU obtuvo ∼68 % de aciertos, frente a ∼60 % de FIFO.
- En la distribución Uniforme, aunque ambos algoritmos presentaron menores tasas de acierto, LRU mantuvo una ligera ventaja sobre FIFO (45 % vs 38 % en 1000 entradas).

La ventaja de LRU radica en que prioriza la retención de ítems recientemente usados, capturando mejor los patrones temporales de repetición. FIFO, en cambio, descarta elementos en orden de llegada sin considerar su relevancia o recurrencia, lo que resulta en más fallos.

## 6.4. Tamaño de la caché

El tamaño de la caché también influye de manera significativa en el rendimiento. En Poisson+LRU, se observó que:

- Con 250 entradas, el hit rate fue cercano al 45 %.
- Con 1000 entradas, el hit rate aumentó a ∼68 %.
- Con 2000 entradas, se alcanzó ∼80 %.

En la distribución Uniforme, el aumento de tamaño también mejora el rendimiento, aunque de manera menos pronunciada (pasando de ∼20 % con 250 entradas a ∼57 % con 2000 entradas).

Esto demuestra que, si bien una mayor capacidad de caché incrementa la probabilidad de acierto, los beneficios dependen de la naturaleza del tráfico. En escenarios de repetición (Poisson), la mejora es notable; en escenarios uniformes, la mejora existe pero es limitada.

## 6.5. Elección final y justificación

Con base en los resultados, se justifica la elección de la política **LRU** como la más adecuada, dado que maximiza el hit rate en todos los escenarios evaluados. Respecto al tamaño de caché, un valor intermedio de ∼1000 entradas representa un buen equilibrio: ofrece tasas de acierto significativamente mejores que una caché pequeña, pero sin requerir tanta memoria como en configuraciones de 2000 entradas.

En un equipo con recursos limitados (4 GB de RAM), esta configuración permite mantener un consumo de memoria moderado (Redis ∼120 MB) y al mismo tiempo reducir de forma considerable la latencia promedio del sistema al disminuir el número de fallos de caché.

En resumen, la política **LRU con un tamaño de 1000 entradas** se presenta como la configuración más adecuada para el sistema bajo las condiciones experimentales planteadas.

# 7. Análisis y Discusión de la Función de Puntuación

## 7.1. Justificación de la Métrica Seleccionada

Para evaluar la calidad de las respuestas generadas por el modelo LLM frente a las respuestas de referencia del dataset, se utilizó la **similitud de coseno sobre representaciones TF-IDF**. Esta métrica mide la *superposición léxica ponderada*, es decir, la coincidencia de términos relevantes entre la respuesta generada y la esperada.

La elección de esta métrica se justifica por varias razones:

- Es una métrica ampliamente utilizada en tareas de recuperación de información y clasificación de texto, lo que facilita su aplicación a este problema.
- Resulta computacionalmente eficiente, lo que permite evaluar un número grande de respuestas aun en un entorno con recursos limitados.
- Captura de manera sencilla qué tan similares son las respuestas en términos de palabras clave, lo cual es un primer indicador de calidad en sistemas de preguntas y respuestas.

En resumen, TF-IDF permite evaluar la **cercanía léxica** entre respuestas y funciona como una primera aproximación razonable para medir calidad.

## 7.2. Utilidad y Limitaciones de la Métrica

Los resultados obtenidos muestran que:

- Alrededor de un 28 % de las respuestas generadas alcanzaron una similitud alta ($> 0{,}8$), lo que indica que el modelo fue capaz de reproducir de manera muy cercana el contenido esperado.
- La mayoría (50 %) se concentró en niveles medios de similitud ($\sim 0.6$), mostrando que las respuestas tienden a compartir cierta base léxica con la referencia, aunque sin ser equivalentes en detalle.
- Un 22 % de las respuestas quedaron en rangos bajos o muy bajos ($< 0{,}4$), lo que refleja casos en los que la métrica penaliza fuertemente la falta de coincidencia literal.

Desde el punto de vista práctico, la métrica TF-IDF resulta útil como:

- Herramienta de monitoreo global del sistema, ya que permite detectar cuando el LLM entrega respuestas muy alejadas de la referencia.
- Criterio de filtrado básico para descartar respuestas de baja calidad en entornos de validación rápida.

No obstante, la métrica presenta limitaciones claras:

- No captura equivalencias semánticas: dos respuestas con sinónimos o reestructuración gramatical pueden recibir un score bajo pese a ser correctas.
- Está sesgada hacia coincidencias literales, lo cual la hace menos adecuada para respuestas largas o para preguntas abiertas con múltiples formulaciones válidas.
- No distingue entre respuestas concisas pero correctas y respuestas extensas pero redundantes, siempre que compartan términos comunes.

## 7.3. Discusión Crítica

En términos de **fiabilidad**, la similitud TF-IDF es suficiente como un primer filtro en la evaluación, pero no puede considerarse un criterio único ni definitivo. Para una evaluación robusta sería recomendable complementar esta métrica con otras más avanzadas, como **ROUGE** (para superposición de n-gramas) o **BERTScore** (para equivalencia semántica basada en embeddings contextuales).

De esta forma, se concluye que la métrica seleccionada aporta valor inicial al sistema y es especialmente adecuada en un entorno con recursos limitados, pero debe entenderse como parte de un conjunto más amplio de herramientas de evaluación.

# 8. Análisis del Sistema de Almacenamiento

Para este proyecto se optó por utilizar **PostgreSQL** como sistema de almacenamiento. La elección se basó principalmente en que necesitábamos un motor confiable, fácil de integrar en contenedores Docker y que ofreciera consultas estructuradas (SQL) sin mayor complejidad.

En comparación con opciones NoSQL como MongoDB, PostgreSQL se ajusta mejor porque los datos que manejamos (preguntas, respuestas originales, respuestas generadas por el LLM y sus métricas de evaluación) tienen relaciones claras y requieren operaciones de agregación y filtrado. Además, la posibilidad de combinar datos tabulares con campos flexibles en `JSONB` entrega un balance entre rigidez y flexibilidad.

Para un escenario de recursos limitados como el de este proyecto, PostgreSQL es práctico: se configura fácilmente en Docker, su consumo puede ajustarse a máquinas con poca memoria, y nos garantiza integridad y consistencia en los resultados. Aunque una base NoSQL podría ser más simple en algunos casos, aquí la estructura relacional nos dio orden y un mejor control sobre las consultas.

En conclusión, la elección de PostgreSQL respondió tanto a la naturaleza del problema como a la necesidad de contar con una solución estable, simple de administrar e integrada sin dificultad en la arquitectura del sistema.

# 9. Conclusión

En este trabajo se pudo experimentar con distintas configuraciones de caché, distribución de tráfico y políticas de reemplazo, evaluando cómo estas decisiones impactan directamente en el rendimiento del sistema. También se analizaron las respuestas del LLM a través de una métrica de similitud, lo que permitió tener una primera aproximación a su calidad, con sus ventajas y limitaciones.

La elección de PostgreSQL como motor de almacenamiento se justificó por su equilibrio entre simplicidad, robustez y facilidad de integración en contenedores, lo que resultó adecuado para el alcance del proyecto y las limitaciones de hardware disponibles.

En general, los resultados obtenidos muestran que incluso en un entorno con recursos limitados es posible diseñar, implementar y evaluar un sistema distribuido de estas características, siempre que se apliquen decisiones cuidadosas sobre políticas de caché, métricas de evaluación y almacenamiento de datos.

Finalmente, todo el código desarrollado para este proyecto se encuentra disponible en el siguiente repositorio:

https://github.com/Jorquerinh0/Tarea1SD.git