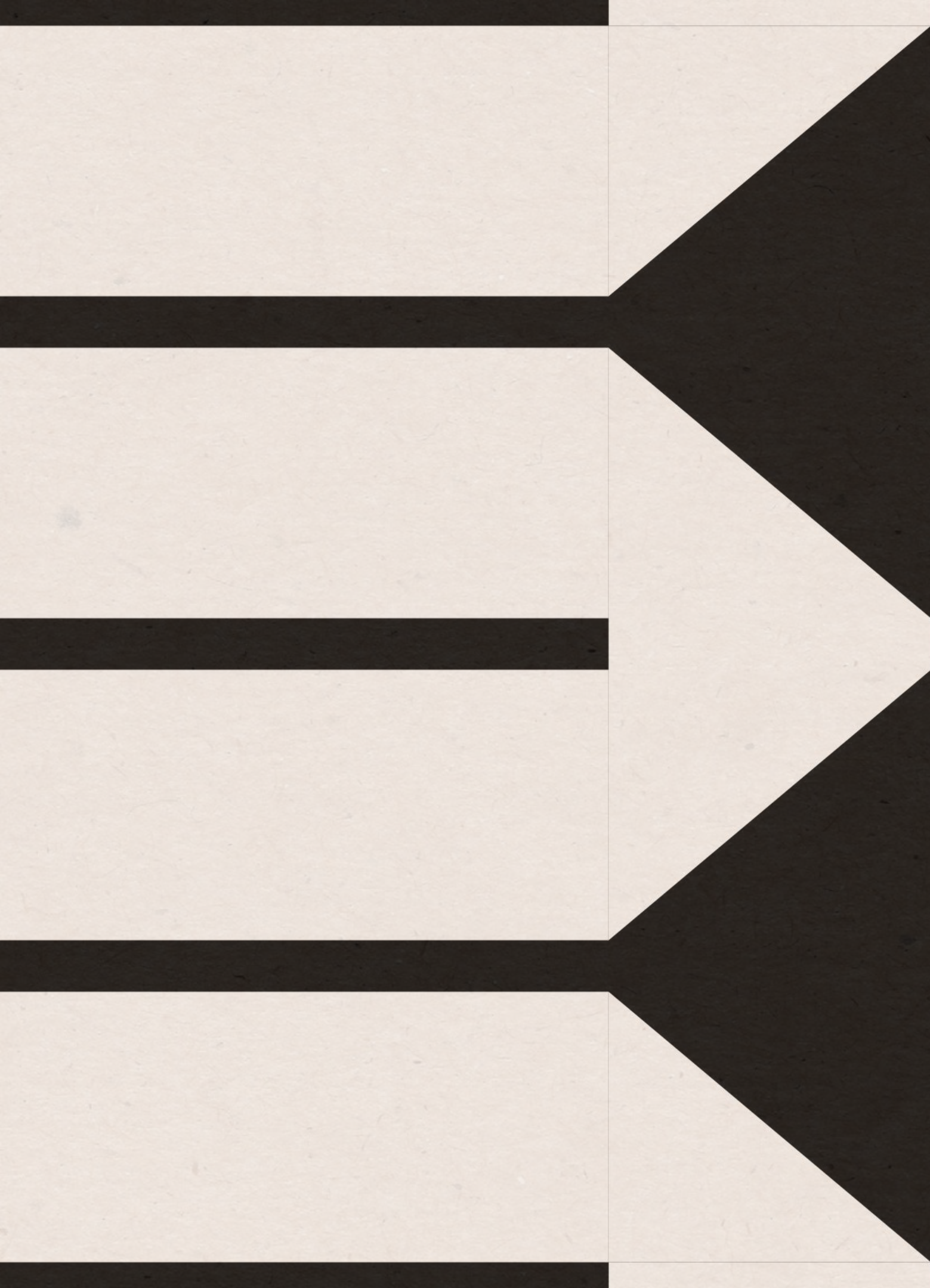




Codebase RAG

Deep Learning
2Q 2025 - ITBA

- 
- 1. Problema**
 - 2. Desafíos**
 - 3. Implementación**
 - 4. Resultados**
 - 5. Conclusiones**

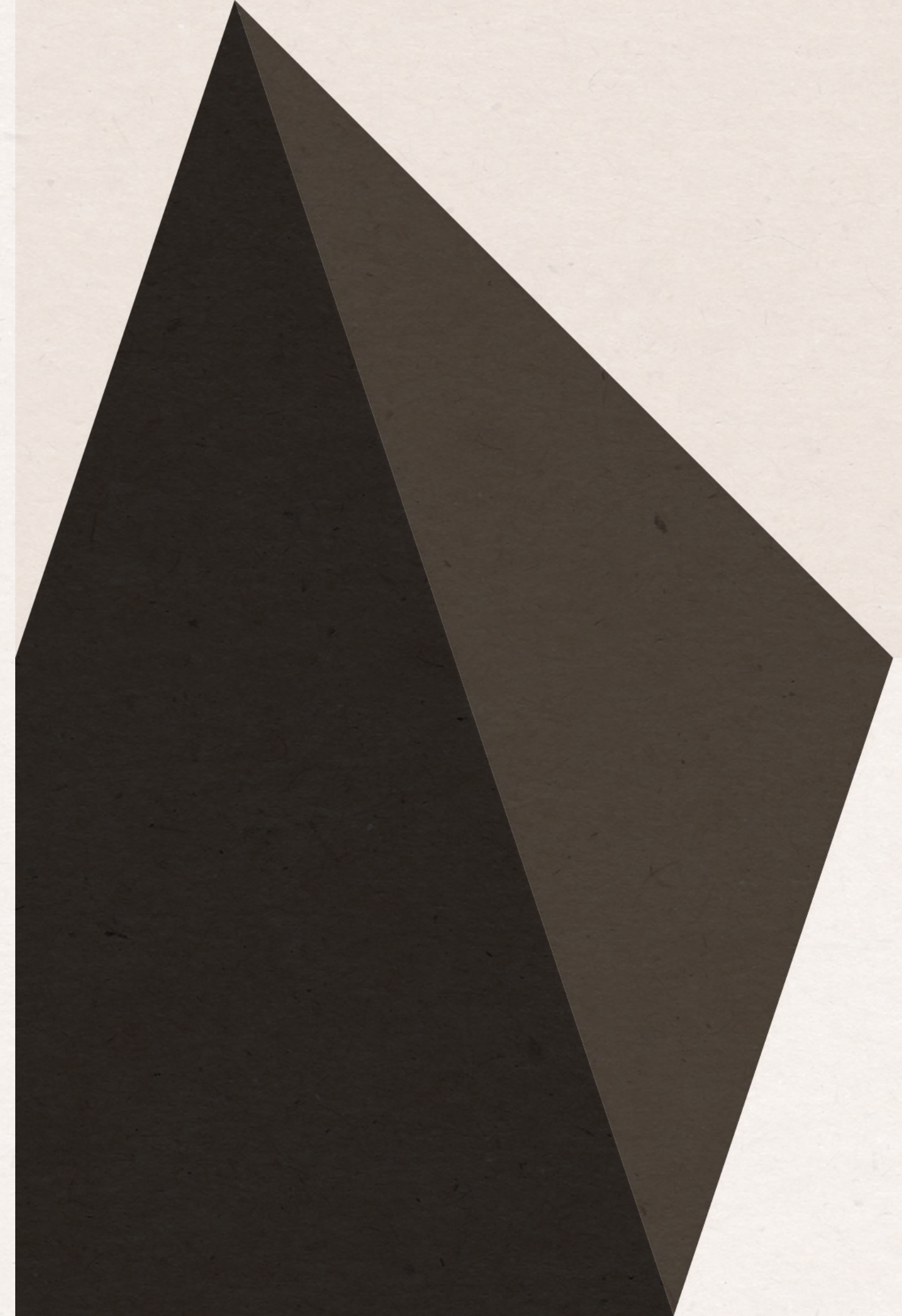
Problema

001

El Problema

Los LLMs como ChatGPT son útiles para generar y explicar código, pero tienen limitaciones críticas:

- No pueden acceder a repositorios fácilmente
- **Context window limitado** : imposible cargar codebases completos
- Sin conocimiento actualizado de proyectos en desarrollo
- **Copiado y pegado manual:** workflow ineficiente y propenso a errores



Solución: Codebase RAG

Sistema de Retrieval-Augmented Generation especializado en código fuente que:

- Indexa repositorios completos (ZIP o GitHub)
- Búsqueda semántica inteligente de snippets relevantes
- Respuestas contextualizadas usando LLMs con retrieval
- Soporte multilenguaje: 30+ lenguajes de programación
- Ejemplos de uso:
 - "¿Cómo funciona la autenticación en este proyecto?"
 - "Explica la arquitectura del módulo de pagos"
 - "¿Dónde se maneja el error 404?"
 - "Generá tests para el modulo X"





Desafíos

002

Desafío I: Parsing Inteligente

Problema: el código no es texto plano, tiene una estructura sintáctica (funciones, clases, bloques) que debe preservarse.

Desafíos específicos:

- **Split ingenuo rompe contexto:** dividir por caracteres "destruye" funciones completas
- Cada lenguaje tiene sintaxis diferente: Python usa indentación, C++ usa llaves, etc...
- **Archivos grandes** : ¿cómo dividir sin perder coherencia semántica?

Impacto:

Chunks mal formados → retrieval ineficaz → respuestas incorrectas

Desafío 2: Búsqueda Semántica

Problema: relacionar la query en lenguaje natural con snippets técnicos de código.

Desafíos específicos:

- **Vocabulario diferente:** usuario dice "validar", código dice "authenticate"
- **Conceptos abstractos:** "flujo de autenticación" vs. código específico
- **Selección de k :** ¿cuántos snippets recuperar?
 - Muy pocos → falta contexto
 - Muchos → ruido



Q: "¿Cómo se validan usuarios?"

Código relevante: `def authenticate_user(username, pwd)`

Desafío 3: Integración Multi-Proveedor

Problema: diferentes LLMs tienen APIs, formatos y capacidades distintas.

Desafíos específicos:

- **APIs incompatibles:** OpenAI usa formato de mensajes, Gemini usa strings, etc...
- **Modelos diferentes:** GPT-5 vs. Gemini 2.5 Flash tienen fortalezas distintas
- **Configuración** : temperatura, max_tokens, system prompts varían

Requisito:

Arquitectura flexible que permita cambiar de proveedor sin modificar la lógica del RAG.



Desafío 4: Evaluación de Calidad

Problema: a diferencia de clasificación (accuracy), evaluar generación de texto es subjetivo.

Preguntas clave:

- **Faithfulness:** ¿la respuesta es fiel al código recuperado?
- **Relevancy:** ¿responde lo que preguntó el usuario?
- **Context quality:** ¿recuperamos los snippets correctos?

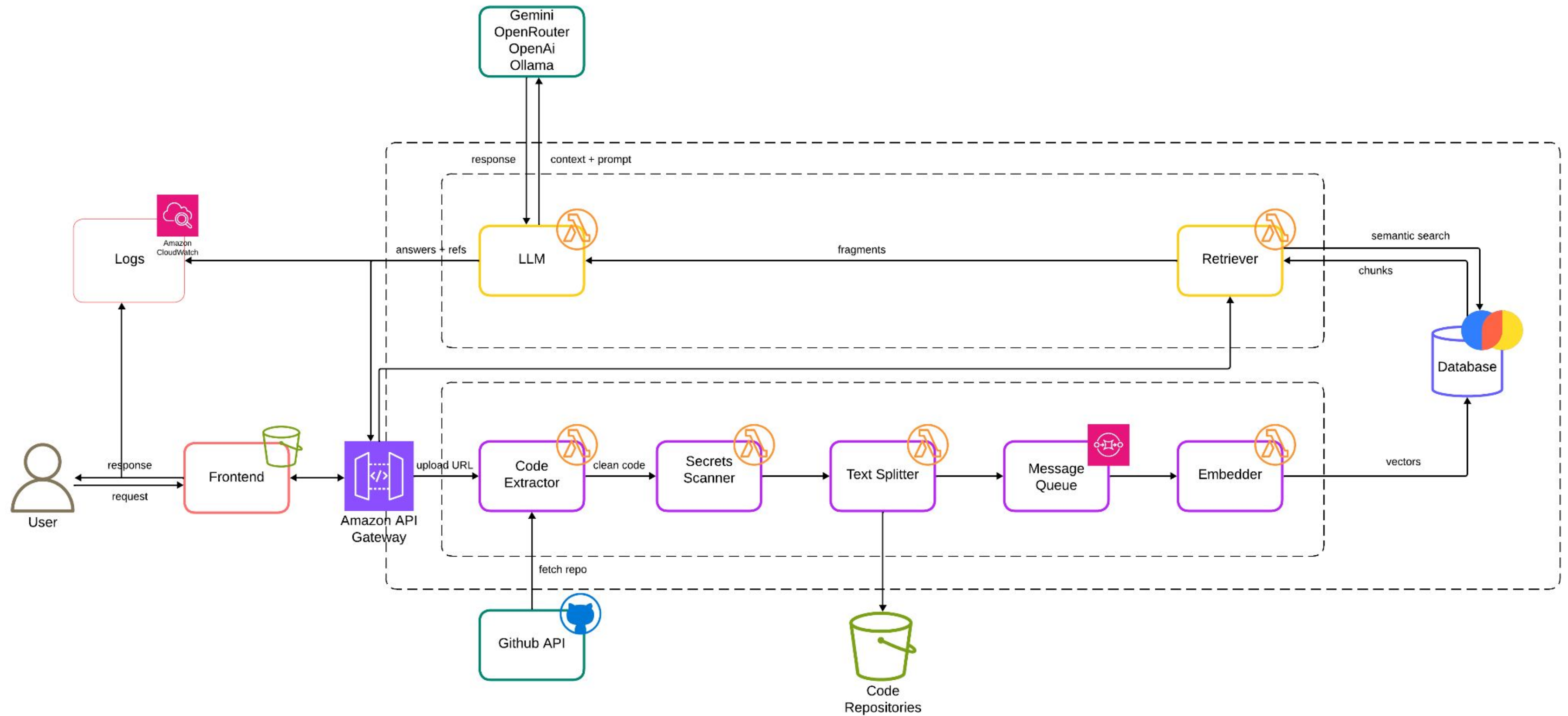
Complejidad:

- Ground truth manual es costoso de generar
- Métricas tradicionales (BLEU, ROUGE) no capturan calidad semántica
- Necesitamos evaluar retrieval + generación por separado

Implementación

003

Arquitectura



Tree-sitter + LangChain

La solución al desafío I (parsing) es usar tree-sitter, un parser universal que entiende la sintaxis de múltiples lenguajes

Ventajas:

- Preserva estructura sintáctica (funciones completas, clases)
- 30+ lenguajes soportados con parsers específicos
- Integración con LangChain para fácil uso
- Parser threshold configurable (tamaño mínimo de chunks)

```
# src/text_splitter.py
from langchain_text_splitters import LanguageParser

parser = LanguageParser(language=detected_lang)
chunks = parser.split_documents(document)
```


ChromaDB para Retrieval

La solución al desafío 2 (búsqueda semántica) es la combinación de ChromaDB con SentenceTransformers.

Ventajas:

- **ChromaDB:** persistente, rápida, fácil de usar
- **SentenceTransformers** : embeddings BERT especializados en similitud semántica
- **k = 5 snippets:** balance entre contexto y ruido
- **Metadata filtering** : por lenguaje, archivo, etc...

```
# src/inserter.py
from chromadb.utils.embedding_functions import
    SentenceTransformerEmbeddingFunction

embedding_function = SentenceTransformerEmbeddingFunction()
collection = client.create_collection(
    name="codeRAG",
    embedding_function=embedding_function
)
```


Strategy Pattern + Factory Pattern

Para abstracción de proveedores.

Características:

- **Interfaz unificada** : generate(messages) para todos
- Auto-detección de API keys desde .env
- **Conversión de formatos:**
OpenAI messages ↔ Gemini strings
- **Extensible:** agregar Claude, local models, etc...

```
BaseLLMProvider (Abstract)
├── GoogleProvider (Gemini)
└── OpenAIProvider (GPT)

LLMFactory (Registry + Cache)
└── create_llm(provider, model)
```


Evaluación con RAGAS

La solución encontrada para el cuarto desafío (evaluación de calidad) fue el uso del Framework RAGAS (Retrieval-Augmented Generation Assessment).

- **Características** : LLM evaluador aislado (no contamina el cache del generador)
- Validación de dependencias de métricas (ground truth)
- Dataset generator para crear casos de prueba
- Versión tracking para reproducibilidad
- Export múltiple: CSV, JSON, Excel

```
# src/evaluation/ragas_evaluator.py
from ragas.metrics import (
    faithfulness,           # ¿Es fiel al contexto?
    answer_relevancy,       # ¿Responde la pregunta?
    context_precision,      # ¿Contexto relevante arriba?
    context_recall          # ¿Todo el contexto necesario?
)
```


Stack Tecnológico

Parsing & Loading

- tree-sitter + LangChain LanguageParser
- Request a la api de Github para clonado de repos
- zipfile para extracción

Retrieval

- ChromaDB (base vectorial persistente)
- SentenceTransformers (embeddings)

Generación

- Modelos disponibles en la API de OpenAI
- Modelos disponibles en la API de Gemini

Stack Tecnológico

Evaluación

- RAGAS (metrics + datasets)
- Pandas para análisis de resultados



CLI & Utils

- argparse para interfaz de línea de comandos
- python-dotenv para configuración
- logging para debugging

Configuración JSON

- **Problema:** diferentes casos de uso requieren diferentes parámetros, se busca no modificar el código
- **Solución:** configuración mediante extensiva a archivos JSON
- **Ventajas:**
 - Configuraciones predefinidas (default, optimal, fast, chat, reranking...)
 - Experimentación fácil (cambiar modelo, k docs, estrategia de reranking)
 - Sin modificar código (todo en JSON)

```
// configs/chat_openai.json
{
  "model": {
    "provider": "openai",
    "name": "gpt-4o-mini",
    "temperature": 0.7
  },

  "retrieval": {
    "k_documents": 5
  },

  "rerank": {
    "enabled": true,
    "strategy": "cross_encoder",
    "retrieve_k": 15,
    "top_n": 5
  },

  "embeddings": {
    "model_name": "all-MiniLM-L6-v2"
  }
}
```

```
# Rápida (GPT-3.5, sin reranking)
python main.py -z code.zip -p "query" --config configs/fast.json

# Máxima calidad (GPT-4o, reranking)
python main.py -z code.zip -p "query" --config configs/optimal.json
```


Reranking de Documentos

```
"rerank": {  
  "enabled": true,  
  "strategy": "cross_encoder",  
  "retrieve_k": 15,           // Candidatos iniciales  
  "top_n": 5                  // Documentos finales  
}
```

```
"rerank": {  
  "enabled": true,  
  "strategy": "mmr",  
  "retrieve_k": 15,           // Candidatos iniciales  
  "top_n": 5                  // Documentos finales  
}
```

- **Problema:** la búsqueda vectorial puede recuperar documentos redundantes o imprecisos
- Implementamos dos estrategias:
 - **Cross-Encoder (precisión):**
 - Mejora precisión
 - Menor velocidad
 - Ideal para queries técnicas
 - **MMR - Maximal Marginal Relevance (diversidad):**
 - $MMR = \lambda \times \text{Relevancia} - (1 - \lambda) \times \text{Similaridad}$
 - Reduce redundancia entre documentos
 - Ideal para queries exploratorias

Chat Interactivo

- **Motivación:** explorar codebases requiere múltiples preguntas con contexto acumulado
- **Características:**
 - Indexación única al inicio
 - Contexto conversacional
 - RAG completo por query (retrieval + reranking + generation)
 - Comandos:
 - /help
 - /clear
 - /exit
 - Modo single-shot también disponible (-p)

```
$ python main.py -z repo.zip --config chat_openai.json

=====
      MODO CHAT INTERACTIVO
=====
Comandos: /help, /clear, /exit

Q: ¿Cómo se crea un usuario?
Buscando...
A: Para crear un usuario, instancia User(name, email)...
   📄 models.py, db.py

Q: ¿Y cómo le asigno roles?
Buscando...
A: (Recuerda que hablabas de usuarios)
   Los roles están en permissions.py...
   📄 permissions.py, models.py

Q: /clear
✓ Conversación reiniciada

Q: /exit
Hasta pronto!
```


Dataset

Codebases reales para testing:

1. SMTP-Protos (C)

- Servidor de email non-blocking con I/O multiplexing
- ~ 3 000 líneas de código

2. [Jam.py](#) (Python)

- Framework web low-code para desarrollo rápido
- Aplicaciones database-driven
- ~ 17 500 líneas de código

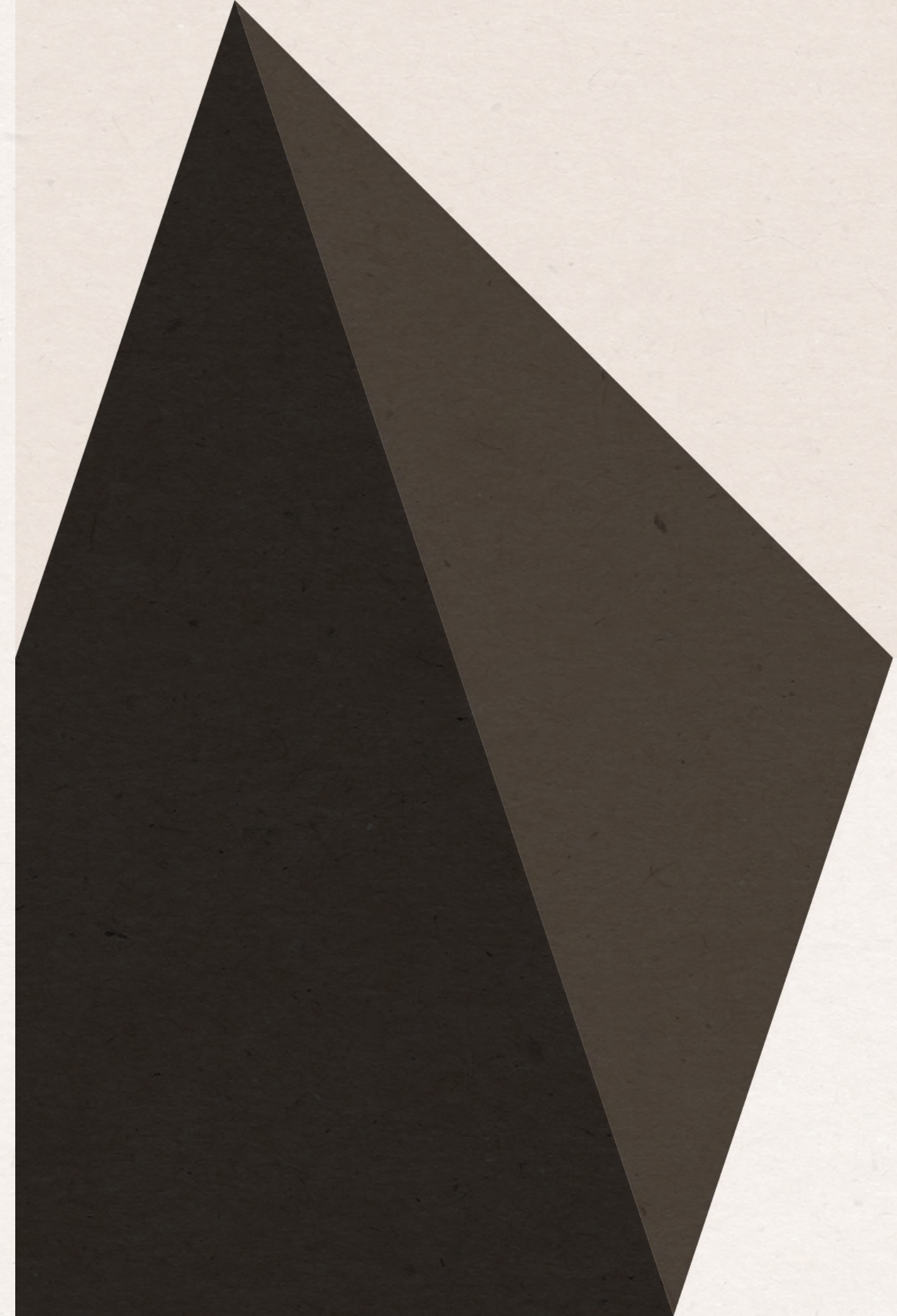
3. NumPy (Python/C)

- Librería científica fundamental de Python
- Código mixto Python + C con optimizaciones SIMD
- ~ 100 000+ líneas

4. Codebase RAG (Python)

- La implementación de este proyecto
- ~3200 líneas de código

Objetivo: evaluar generalización del sistema RAG en proyectos reales de diferentes dominios, lenguajes y complejidades.



Demo!



1. Indexación del Código
2. Modo Single-Shot
3. Modo Chat Interactivo
4. Comparación de Configuraciones



Resultados

004

Metodología de Evaluación

Métrica	Componente	¿Qué mide?	GT
Context Precision	Retrieval	¿Documentos bien rankeados?	✓
Context Recall	Retrieval	¿Info completa recuperada?	✓
Faithfulness	Generation	¿Se basa en contexto?	x
Answer Relevancy	Generation	¿Respuesta relevante?	x

Marco de Evaluación: RAGAS + Diseño Experimental

- **Framework** : RAGAS con LLM-as-Judge
- 4 métricas complementarias
- **Ground Truth**
 - 60 queries curadas (20 técnicas, 20 arquitectónicas, 20 semánticas)
 - Referencias manuales validadas + archivos esperados por query
 - Evaluador LLM aislado (gpt-4.1-mini, T=0.3) para fairness
- **Diseño Experimental**
 - A/B Testing con variable única
 - Variables controladas (modelo, temperature, embeddings)
 - Configuraciones versionadas para reproducibilidad
- **Score Range:** 0.0 (peor) → 1.0 (perfecto)



Templates de Prompts

Hipótesis: templates con instrucciones explícitas mejorarán faithfulness en un 10% al reducir alucinaciones.

Fundamentos:

- Prompts detallados fuerzan al LLM a citar contexto explícitamente
- Estructura forzada reduce improvisación y deriva del contexto
- Prompts con ejemplos y constraints mejoran adherencia

Análisis y Resultados

Hipótesis parcialmente validada, aumento de faithfulness leve

Insights:

- Template detailed mejora faithfulness pero menos de lo esperado
- Menor Answer Relevancy para el detailed prompt.
- Puede deberse a que está realizando respuestas innecesariamente más largas o detalladas de lo que deberían ser.

Config	Faithfulness	Answer Relevancy	Context Precision	Context Recall
detailed	0.986	0.69	0.599	0.76
default	0.975	0.83	0.557	0.66

k_documents

Hipótesis: existe trade-off Precision \leftrightarrow Recall.
El punto óptimo se encuentra entre $k=5$ y $k=8$.

$k < 5 \rightarrow$ alta precisión/bajo recall

$k > 10 \rightarrow$ degradación por "lost in the middle"

Fundamentos:

- Más documentos llevan a mayor cobertura pero mayor ruido
- LLMs degradan con contextos largos
- Teoría de IR: Precision decrece con k , Recall crece

Análisis y Resultados

Hipótesis parcialmente validada: "Sweet spot" entre Precision y Context Retrieval.

Insights:

- A mayor K. mas documentos por los que basarse. mayor faithfulness.
- Retrieve de mayor cantidad de documentos ayuda al modelo a encontrar su "ground truth", aumentando el context recall.
- Pico de Context Precision en k=8. Mayor K implica que se pueden extraen documentos menos relevantes.

Config	Faithfulness	Answer Relevancy	Context Precision	Context Recall
k=3	0.690	0.576	0.400	0.327
k=8	0.819	0.809	0.732	0.670
k=15	0.9972	0.8322	0.5075	0.8000



Modelos de Embeddings

Hipótesis: all-mpnet-base-v2 (MPNet) mejorará retrieval en +5-10% vs all-MiniLM-L6-v2 (MiniLM) debido a mayor dimensionalidad semántica.

Fundamentos:

- mpnet: 768 dims vs MiniLM: 384 dims → mayor capacidad semántica
- Arquitectura MPNet: pre-training superior con permuted language modeling
- Trade-off 2x más lento en indexing, 2x storage

Análisis y Resultados

Hipótesis validada: MPNet supera consistentemente a MiniLM en todas las métricas, con una mejora promedio del 40.8% en *retrieval*, mayor a la esperada.

Insights:

- *Faithfulness* aumenta considerablemente (+0.24) indicando reducción en alucinaciones
- MiniLM presenta rendimiento aceptable considerando su dimensionalidad, pero pierde semántica notable en queries complejas
- *Ceiling effect*: MPNet se acerca al límite de precisión contextual posible sin sobreajuste.
- Trade-off costo-calidad:
 - MPNet: Mejores respuesta a un costo aproximadamente doble
 - MiniLM: menor costo computacional, adecuado para pipelines ligeros.
- MiniLM sorprendentemente robusto para prototipado rápido o entornos con recursos limitados.

Config	Faithfulness	Answer Relevancy	Context Precision	Context Recall
mpnet	0.97	0.67	0.73	0.67
MiniLM	0.78	0.66	0.52	0.45



Estrategias de Reranking

Hipótesis: Two-stage retrieval con Cross-Encoder mejorará la precisión en +8-12% vs. retrieval directo. MMR tendrá mejora intermedia (+3-5%).

Fundamentos:

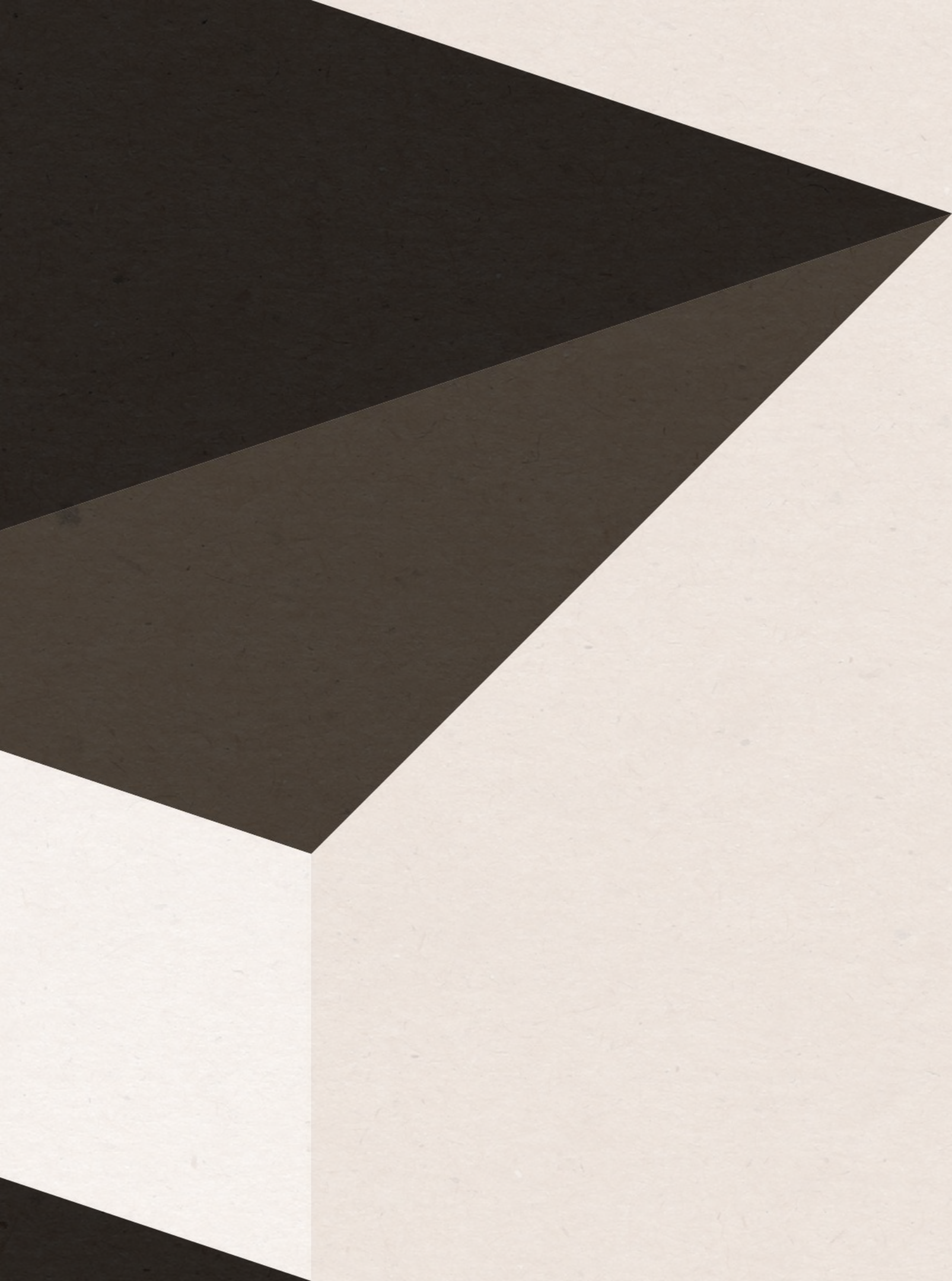
- Cross-Encoder: re-evaluá relevancia con model más potente (bi-encoder inicial es rápido pero menos preciso)
- MMR: optimiza diversidad (útil en NLP general, menos en código)

Análisis y Resultados

Insights:

- Sin reranks: Mayor faithfulness y relevancia con la respuesta. Debido a su simplicidad, se pierde documentos importantes en contexto.
- Cross encoder: Mayor answer relevancy. Posiblemente priorizo snippets que fueron más relevantes con el modelo potente, pero se pierde precisión.
- MMR: La diversidad de MMR ayudó en el context recall a costa de la relevancia del resultado final.

Config	Faithfulness	Answer Relevancy	Context Precision	Context Recall
CE (I5→5)	0.925	0.88	0.55	0.7
MMR (I5→5)	0.967	0.804	0.557	0.73
No Reranking	0.976	0.838	0.59	0.66



Heterogeneidad Multi-Language

Hipótesis: Repositorios multi-language mostraran menor precision en *retrieval* comparados con repositorios mono-lenguaje.

Fundamentos:

- Embeddings generales capturan peor la semántica de múltiples lenguajes
- Diferencias de sintaxis y convenciones introducen ruido en el espacio vectorial
- Queries ambiguas pueden recuperar *snippets* de lenguaje incorrecto
- La homogeneidad semántica de un solo lenguaje favorece el *clustering* y la precisión contextual

Análisis y Resultados

Hipótesis rechazada: mejora observada (-10.67%) menor que la estimada (10%)

Insights:

- Configuración Optimal: el *single-language* mejora ligeramente en *faithfulness* (+9.6%) y *answer relevancy* (+4.8%), pero pierde en *context recall* (-23%).
 - Muestra un mejor ajuste cuando la precisión contextual no es prioritaria.
- Predominio multi-language: gana en 9/12 métricas (75%), incluyendo *context precision* y *recall* en configuraciones *default* y *fast*.
 - Indica mejor recuperación y relevancia en entornos heterogéneos.
- Ceiling effect: las configuraciones con *faithfulness* inicial alta (~0.98) dejan poco margen de mejora.

Config	Faithfulness	Answer Relevancy	Context Precision	Context Recall
jam-py-v7	0.90	0.70	0.41	0.61
SMTP - Protocolos de comunicacion	0.99	0.73	0.43	0.47



Tamaño del Repositorio

Hipótesis: Context Precision se degradará un 10-15% al escalar de repos pequeños (1K docs) a grandes (40K+ docs) por aumento de signal-to-noise ratio.

Fundamentos:

- Más documentos → mayor probabilidad de falso positivos semánticos
- Embeddings generales capturan similitud superficial
- Vector search retrieval escala sub-linealmente en calidad

Análisis y Resultados Numpy

Hipótesis validada: Context precision bajo en gran parte.


- Demasiados documentos en el espacio vectorial, retrieval poco preciso.
- Context recall bajo en gran parte.
- Faithfulness y Answer Relevancy sospechosamente altos.
- Posiblemente el modelo responde con lo que "ya sabia" de numpy.

Config	Faithfulness	Answer Relevancy	Context Precision	Context Recall
default	0.917	0.82	0.08	0.175



Conclusiones

005



Este trabajo demuestra que la arquitectura RAG es una solución efectiva y escalable para la exploración inteligente de repositorios de código mediante lenguaje natural. El sistema desarrollado aborda exitosamente las limitaciones de los LLMs tradicionales al integrar parsing inteligente multilenguaje, búsqueda semántica especializada y generación contextualizada.

Hallazgos Clave

1. La configuración del sistema es determinante para el rendimiento
2. El sistema escala, pero con limitaciones predecibles
3. Trade-offs bien definidos permiten optimización por caso de uso
4. La evaluación sistemática con RAGAS es fundamental

Trabajo Futuro

- Embeddings especializados en código
- Soporte para lenguajes de programación adicionales
- Metadata filtering avanzado
- Agentic RAG
- Escalabilidad a repos > 100K docs

Gracias!

Preguntas?

