



```
f"Hola  
{user}"
```

```
StringConcatenationManagerDeluxe  
    (template="prompt.md",  
     contexts={user: "Alex"})
```

GenAI ❤️ f-string

Desarrollando con IA Generativa
sin cajas negras

Alejandro Vidal
alex@mindmake.rs

¿Quién soy?

Alejandro Vidal

TLDR: Fundador de mindmake.rs (GenAI edtech) de día. Consultor en IA Generativa por la noche.

¿Por qué deberías ver esta charla?

El... 40% de mi consultoría es migrar pipelines de IA Generativa de frameworks a Python puro debido a su falta de flexibilidad y el "vendor lock-in".

Hoy intentaré contarte cómo puedes evitar muchos problemas.

 Contacto



Operaciones simples como interpolaciones de texto (un `f-string`, vaya) se llaman `StringConcatenationManagerDeluxe`.



Todos los frameworks generan deuda técnica y "framework lock". Pero...

¿cuánta? ¿cómo de rápido?

¿sabes reducirla?*

*sin pagar

Esta no es una charla para que:

Dejes de usar frameworks

Despliegues toda la infra de tu startup en una raspberry en tu sotano

Reinventes la rueda

Es una charla para:

Pienses críticamente qué herramientas debes usar, qué necesitas y qué puedes hacer por ti mismo.

Entiendas la deuda técnica **antes** de que se convierta en un problema.

"Estás exagerando, ¿no?" - dijo un amable asistente al taller



Hamel Husain ✅
@HamelHusain ·



Programming abstraction -> a human-like language you can use to translate your task into machine code

LLM abstraction -> an unintelligible framework you can use to translate your task into human language

2:25 AM · Feb 5, 2024



48 Copy link

[Read 2 replies](#)

Hamel Husain escribe a inicios de 2024 un artículo en el que se queja de esto mismo.

La idea que tiene es registrar las llamadas a OpenAI para tener un historial y poder "ver" que prompts se han usado.

LLM DEVELOPER



mitmproxy

IS THIS OBSERVABILITY?

¿Ha mejorado la situación? (8 meses después)

Buscando una solución de observabilidad sencilla*

sencilla = sin servidores, proxies, contenedores. Autocontenido en python

Running locally

1. Clone the repository
2. Setup a PostgreSQL instance (version 15 minimum)
3. Copy the content of `packages/backend/.env.example`
4. Copy the content of `packages/frontend/.env.example`
5. Run `npm install`
6. Run `npm run migrate:db`
7. Run `npm run dev`

Deploying the Application

Deploy the application container to your infrastructure. You can use managed services like AWS ECS, Azure Container Instances, or GCP Cloud Run, or host it yourself.

During the container startup, all database migrations will be applied automatically. This can be optionally disabled via environment variables.

Pin major version **Use latest version**

```
docker pull langfuse/langfuse:2
```

Self-Hosting Open Source LLM Observability with XXXX

Architecture

XXXXXX is comprised of five services:

Web: Frontend Platform (NextJS)

Worker: Proxy Logging (Cloudflare Workers)

Jawn: Dedicated Server for serving collecting logs (Express + Tsoa)

Supabase: Application Database and Auth

ClickHouse: Analytics Database

Minio: Object Storage for logs.

¿Por qué?

El mercado tiene muchas soluciones tienen un modelo de core open-source + SaaS.

Esto hace difícil la portabilidad entre ellas.

Tienden a no ser interoperables y a "recomendar" sus propias soluciones.

Debido al rápido desarrollo + fondeo de VCs ("break thing fast", "winner takes all")
algunos frameworks arrastran deuda técnica o se "hipertrofian" rápidamente.

¿Por qué? II

¡El "framework/vendor lock-in" está incentivado!

No encontré una solución que fuera simple y autocontenido.

En muchos casos no es necesaria tanta infraestructura y abstracción.

Parte 0: terapia grupal para afectados por los frameworks

 Done

"Framework-lock": ¿Merece la pena? ¿Cómo migrar?

Preparando el entorno

Repo: https://github.com/double-thinker/genai_loves_fstrings

1. Usar [devcontainers](#) o pulsar  en la página del repo.

o clonar el repo y ejecutar:

```
uv sync  
source .venv/bin/activate
```

2. Configurar la API key de OpenAI en el fichero  (ver )

"Hello World" de LLMs: RAG

Vamos a migrar un pipeline básico de RAG de `lanchain`

Para reflexionar al acabar:

¿Cuándo merece la pena usar un framework? ¿Cuándo hay que migrar?

¿Qué partes debería implementar y cuáles externalizar en un framework?

Un RAG básico tiene las siguientes partes:

1. Loader: obtiene los textos que vamos a ingestar, puede incluir tareas como scrapping, descarga, limpieza de PDFs, etc.
2. Chunker: divide los textos en chunks de un tamaño razonable para el consumo de la LLM.
3. Retriever: Indexación y Vectorización de los chunks. En este caso usaremos la BBDD vectorial `chroma`.
4. Prompting: composición de un prompt (aka. texto) con los chunks relevantes para la respuesta.

Si vas a migrar tu proyecto en producción ;No todas las partes deben ser migradas a la vez!

```
python -m rag "What is Task Decomposition?"
```

Veamos el pipeline en `rag.py`

```
rag_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)
```

"Overhead cognitivo del framework"/Preguntas que se hacen un principiante

¿Qué es el diccionario {"context": ...} ?

¿Qué es | ?

¿Para qué sirve RunnablePassthrough() ?

¿Qué es StrOutputParser() ?

¿Qué es el diccionario `{"context": ...}` ?

Contexto global para el pipeline. Similar al contexto de un `jinja2`.

¿Qué es `|` ?

Operador de composición de funciones / operador pipeline.

¿Para qué sirve `RunnablePassthrough()` ?

Indica que el valor `question` será un parámetro del pipeline.

¿Qué es `StringOutputParser()` ?

No hace nada: "parsea"/obtiene el resultado más probable del LLM. Hay otros más útiles (p.e.: parsear JSON).

Mi hipótesis (contrastada múltiples veces en proyectos reales). El beneficio en velocidad de estos frameworks se ve anulado por el coste de aprendizaje, la sintaxis no estándar y el "framework lock-in".

Intentaremos conseguir algo como esto:

```
def pipeline(question: str):
    db = fill_db(...) # 1. Loader/Chunking
    context = db.query(...) # 2. Retriever
    return llm(prompt(context, question)) # 3. Prompting
```

Ventajas:

Sintaxis estándar. Fácil de entender por personas no expertas.

Fácil de extender (decoradores, magic `__call__`, etc.)

No hay que aprender un nuevo "jargon".

Todo el código (incluidos prompts) están versionados en nuestro repositorio.

Prompts.

```
prompt = hub.pull("rlm/rag-prompt")
```

vs

```
def prompt(docs: list[str], question: str) -> str:  
    return f"""You are an assistant for question-answering tasks  
[...] keep the answer concise.
```

Question: {question}

Context: {'\n\n'.join(docs)}

Answer:"""

```
prompt = hub.pull("rlm/rag-prompt")
```

vs

```
def prompt(docs: list[str], question: str) -> str:  
    return f" [...] {question} [...] {'\n\n'.join(docs)}"
```

Fácil de entender.

Fácil de debuguear (¡F11 works!).

Fácil de tipar.

Fácil de extender (decoración, magic `__call__`, etc.)

Fácil de empaquetar (crear módulos de prompts no es distinto a crear cualquier otro módulo Python)

Extremadamente portable -Hi WebAssembly ;)- e incluso a otros lenguajes.

Mi recomendación: Lo primero a migrar son los prompts. Es fácil, mejora el código rápidamente y es un buen ejercicio para entender el problema.

¡Versionado!: Cuidado al usar `hub.pull()`, no podremos observar los cambios en el prompt.

`hub.pull()` genera una falsa sensación de seguridad ("el prompt oficial")

Tip: Si necesitas un repositorio de prompts... usa GitHub, tiene URLs versionadas accesibles por HTTP.

Probablemente no lo necesites (aún).

Lo básico:

```
def llm(prompt: str, model: str = "gpt-4o-mini") -> str:  
    """Función que llama a la LLM. Hay otras librerías que lo hacen (ver comentarios)"""  
    #...  
  
def prompt(docs: list[Document], question: str) -> str:  
    """Usa una f-string para formatear el prompt"""  
    #...
```

En la v1: Mantenemos el splitter, el loader y el tooling de Chroma (es más difícil de migrar)

```
loader = WebBaseLoader(...)  
docs = loader.load()  
text_splitter = RecursiveCharacterTextSplitter(...)  
splits = text_splitter.split_documents(docs)  
vectorstore = Chroma.from_documents(documents=splits,  
                                      embedding=OpenAIEmbeddings())  
retriever = vectorstore.as_retriever()
```

No intentéis migrar todo de una vez. Estas tres partes suelo mantenerlas, al menos en las primeras versiones.

En mi día a día tengo pequeñas librerías que reutilizo y implementan cosas parecidas sin tanto overhead. (p.e.: chunkers)

Depende de cada proyecto: tamaño, deuda técnica, etc.

Para acabar...

```
def chatbot(question: str):
    # Obtenemos los documentos relevantes
    docs = retriever.invoke(question)

    # Generamos una respuesta
    return llm(prompt(docs, question))

if __name__ == "__main__":
    #...
```

Solución:

```
python -m solved.rag.v1 "What is Task Decomposition?"
```

rag v2: migrando todo

No es siempre necesario y no lo recomiendo en todos los proyectos.

Lo muestro para ilustrar las posibilidades.

Muchas partes del código de v2 deberían ser externalizadas a librerías para su reutilización o pueden ser sustituidas por librerías especializadas.

```
def scrappe_web(url: str) -> Generator[str, None, None]:  
    # ...  
    yield ...  
  
def text_splitter(doc: str, chunk_size: int = 1000, chunk_overlap: int = 200):  
    # ...  
    yield ...  
  
def fill_db(docs: Generator[str, None, None]):  
    """Crea y rellena la base de datos con los documentos aportados."""  
    # ...
```

Solución en `solved/rag/v2.py`

El chunker es fácilmente reutilizable (se puede usar como una función normal), no siempre merece la pena migrarlo

Lo mismo con el loader, aunque hay muchas otras librerías que lo hacen mejor (p.e.: [unstructured](#))

El scrapper en este caso está implementado con bs4 al ser muy sencillo pero hay herramientas como `scrapy` para casos más complejos.

(Mis) Conclusiones

Muchos pipelines son fácilmente reimplementables en Python puro con una cantidad de código similar (¡o menor!)

Puedes seguir usando piezas relevantes del framework (p.e.: `WebBaseLoader` o `RecursiveCharacterTextSplitter`).

No debes eliminar todo el framework de una vez: no suele merecer la pena.

No es necesario usar todo el framework: puedes mantener algunas piezas

En mi experiencia todo el pipeline salvo los retrievers es más sencillo en Python puro que en el framework.

(Mis) Conclusiones II

Perks "gratuitos" al escribir en Python puro: observabilidad, cacheo, timing, debugging se vuelve trivial vs docker, servidores, proxies, etc.

```
from cachier import cachier

def observe(fn): ...# logging logic

@observe
@cachier(...)
def llm(prompt: str, model: str = "gpt-4o-mini") -> str:
    #...
```

Comparado con algunas de las soluciones del mercado, creo que merece mucho la pena.

Show me the prompt

Tras mi frustración por tener una solución de observabilidad **sencilla** para proyectos escritos por terceros, decidí escribir una utilidad muy sencilla (~400 locs) que parchea `openai` para registrar las llamadas.

En el repositorio tenéis una versión simplificada

**Cuando sea más estable la publicaré como una librería independiente
(escribidme si estáis interesados: alex@mindmake.rs)**

Estado del arte en observabilidad de LLMs

levanta *varios* dockers con *varias* configuraciones para tener una interfaz web con un formato *no estándar*.

Usando monkeypatching

```
import observability.openai
```

Ya está :)

SmartLLMChain

SmartLLMChain usa varias técnicas para mejorar el resultado (self-critique, etc.). Pero leyendo la documentación no queda claro como funciona.

Esto es muy común en los frameworks actuales debido a la velocidad de desarrollo y al crecimiento del ecosistema.

SmartLLMChain

```
python -m smartllm "¿Cuál es el sentido de la vida?"
```

Tarea: Añade observabilidad para entender cómo funciona SmartLLMChain para poder reimplementarlo posteriormente.

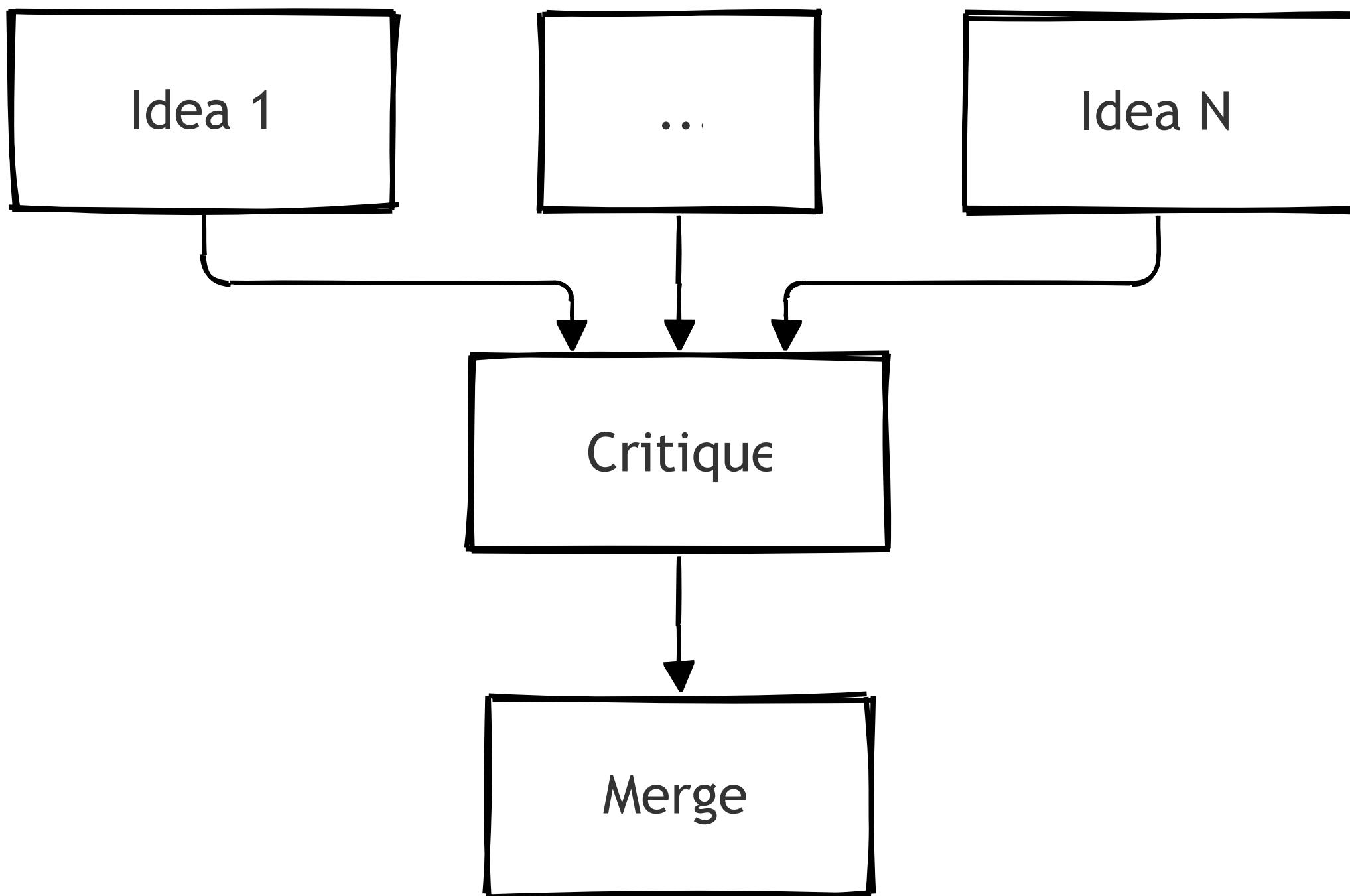
Idea 1

...

Idea N

Critique

Merge



SmartLLMChain in a nutshell

```
def idea_prompt(question: str) -> str:  
    """Idea generation. Se ejecuta N veces."""  
    return f"""Question: {question}  
Answer: Let's work this out in a step by step way to be sure we have the right answer:  
"""  
  
def critique_prompt(question: str, ideas: list[str]) -> str:  
    """Crítica con las ideas generadas."""  
    return f"""{idea_prompt(question)}  
{'\n'.join(f"> Idea {i+1}: {idea}" for i, idea in enumerate(ideas))}  
You are a researcher [...]:  
"""  
  
def merge_prompt(question: str, ideas: list[str], critique: str) -> str:  
    """Obtenemos resultado final combinando las ideas y la crítica."""  
    return f"""{critique_prompt(question, ideas)}  
{critique}  
You are a resolver tasked with [...]  
"""
```

SmartLLMChain in a nutshell

```
def smartllm(question: str = DEFAULT, n_ideas: int = 2):
    ideas = [llm(idea_prompt(question)) for _ in range(n_ideas)]
    critique = llm(critique_prompt(question, ideas))
    return llm(merge_prompt(question, ideas, critique))
```

(Mis) Conclusiones III

Prácticamente todas las técnicas "avanzadas" de prompting se pueden implementar con control de flujo, funciones, corutinas y un par de funciones auxiliares.

Incluído agentes (no lo veremos en esta charla).

El código es mucho más legible, lo cual es crítico cuando trabajamos con LLMs que son **muy** sensibles a cambios pequeños.

Desde cero: mejores prácticas para desarrollar aplicaciones complejas

Extractor

Reimplementaremos un extractor de metadatos/campos de un documento. Es un caso muy útil de LLMs.

1. Damos a la LLM un documento (p.e.: web de esta charla) y el esquema de los datos que queremos extraer: autor, título, fecha, etc.
2. Pasamos varios procesos de validación para asegurar la fiabilidad de los datos.
3. En caso de error, mostramos a la LLM el error y pedimos que lo corrija.
4. Repetimos el proceso hasta que todas las validaciones pasen.

Nota: actualmente recomiendo usar function calling o generación restringida para este caso si el proveedor lo soporta.

Resultado final:

```
python -m solved.extractor.v5
```

```
{'links': [{  
    'description': 'Artículo sobre problemas en IA Generativa.',  
    'url': 'https://hamel.dev/blog/posts/prompt/'},  
    {'description': 'Discusión en GitHub sobre frameworks.',  
    'url': 'https://github.com/langchain-ai/langchain/discussions/18876'},  
    {'description': 'Tutorial introductorio usando langchain.',  
    'url': 'https://python.langchain.com/v0.1/docs/use_cases/question_answering/'}],  
'speaker': 'Alejandro Vidal',  
'technologies': ['OpenAI', 'LLMs', 'langchain'],  
'title': 'GenAI ❤️ f-string. Desarrollando con IA Generativa sin cajas negras.'}
```

Comenzad con el `v1.py` y añadid:

Una función que extraiga un bloque de JSON de la salida de la LLM:

```
```json
{...}
```
```

Usa esta URL para probar: <https://pretalx.com/pycones-2024/talk/SKZFHY.ics>

Extractor v4 y v5

Tomando el `v3.py` como base, añade:

Un validador para links que asegure que tiene este formato: `{ 'description': str, 'url': str}` (solución en `v4.py`)

Un validador que filtre las tecnologías con los siguientes criterios:

Las siglas están descritas entre paréntesis, por ejemplo: `IPv6 (Internet Protocol Version 6)`

Las tecnologías están escritas en inglés

(solución en `v5.py`)

```
def extractor(model: Model, doc: str, max_retries: int = 3) -> dict[str, any] | None:
    parsed, validation_errors = None, []
    for _ in range(max_retries):
        if validation_errors:
            prompt = fix_fields_prompt(model, doc, parsed, validation_errors)
        else:
            prompt = extract_fields_prompt(model, doc)

        output = llm(prompt)
        parsed, validation_errors = validate_output(output, model)
        if not validation_errors:
            return parsed
    return None
```

```
def validate_techs(techs: list[str]) -> None:
    output = parse_json_block(
        llm(f"""De la siguiente lista de etiquetas: {techs} verifica que
se cumplen las siguientes condiciones:
- Las siglas están descritas entre paréntesis, por ejemplo:
`IPv6 (Internet Protocol Version 6)`
- Están escritas en inglés
```

Muestra los errores en una lista formateada como un array JSON con el siguiente formato:

```
```json
[
 "No está en inglés: Programación en Python, ...",
 ...
]
```

Si no hay errores devuelve una lista vacía.

```
Errores

```json
["""]
    )
)

if output:
    raise ValueError(output)
```

Mejores prácticas

Prioriza la migración de prompts y los pipelines. RAGs, loaders, etc son secundarios.

Usa librerías especializadas en cada una de esas partes (chroma, jinja2, scrapy, unstructured, llamaindex, etc.)

Usa frameworks para las primeras demos/MVPs, pero guarda tiempo para migrarlo (no es tanto como parece).

Mejores prácticas II

Mantén tus APIs simples, tipadas y documentadas. Estamos haciendo concatenación de cadenas, no cohete espaciales.

```
def llm(prompt: str, model: str, ...) -> str y def  
xxx_prompt(variable: str) -> str son los más básicos.
```

Puedes tener un módulo separado con los prompts o usar un repositorio de git con **ficheros de texto** o templates si quieres algo más avanzado.

```
def my_pipeline(variable: str) -> str si no necesitas multi-step.
```

```
async def my_pipeline(variable: str) -> ... siquieres usar corrutinas y  
hacerlo asíncrono.
```

Mejores prácticas III

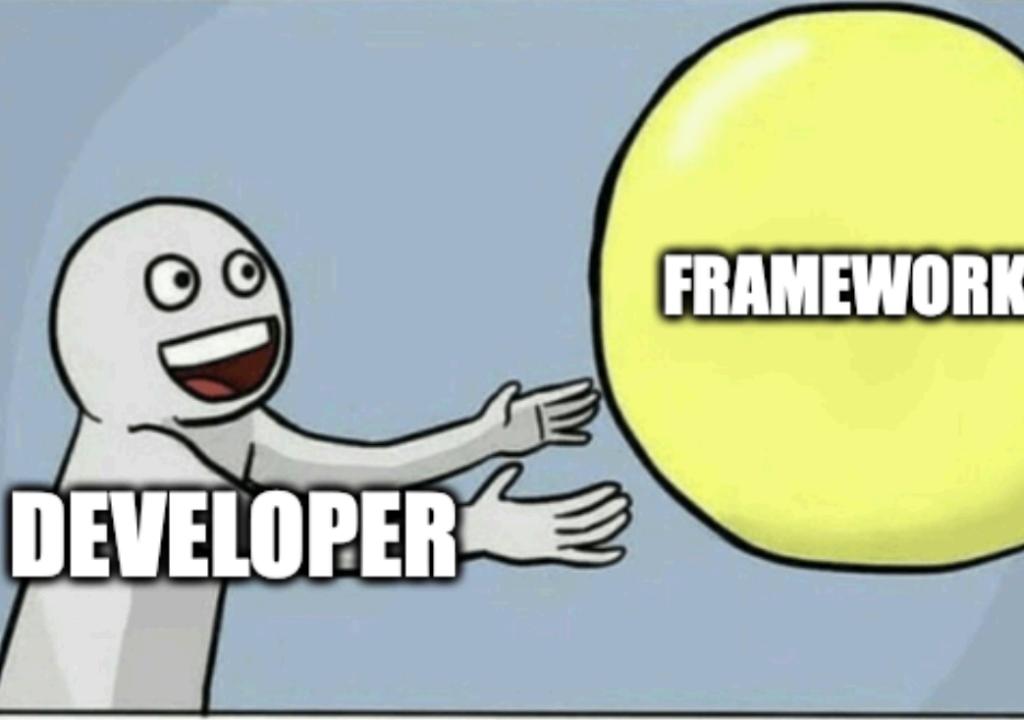
No te obsesiones: puedes usar frameworks, son útiles, no reinventes la rueda.

Pero no copies y pegues el ejemplo del tutorial sin entender cómo puedes migrarlo.

Se crítico con qué partes son necesarias y entiende la deuda y el lock-in que adquieres.

En código de LLMs **Legibilidad >>> LOCs**: no intentes hacerlo breve, hazlo sencillo.

Pipelines complejos son en un 90% control de flujo y condicionales básicos. No te dejes llevar por la inercia de la industria.



¡Muchas gracias!

Si estáis en una situación similar o vais a comenzar a crear vuestro propio pipeline, no dudéis en contactar.

O si sois una empresa que quiere aprender a usar IA



... [Contacto](#)

