GenAI ❤ f-string

Developing with Generative AI without black boxes

Alejandro Vidal
alex@mindmake.rs

Who am I?

Alejandro Vidal

TLDR: Founder of mindmake.rs (GenAI edtech) by day. Generative AI consultant by night.

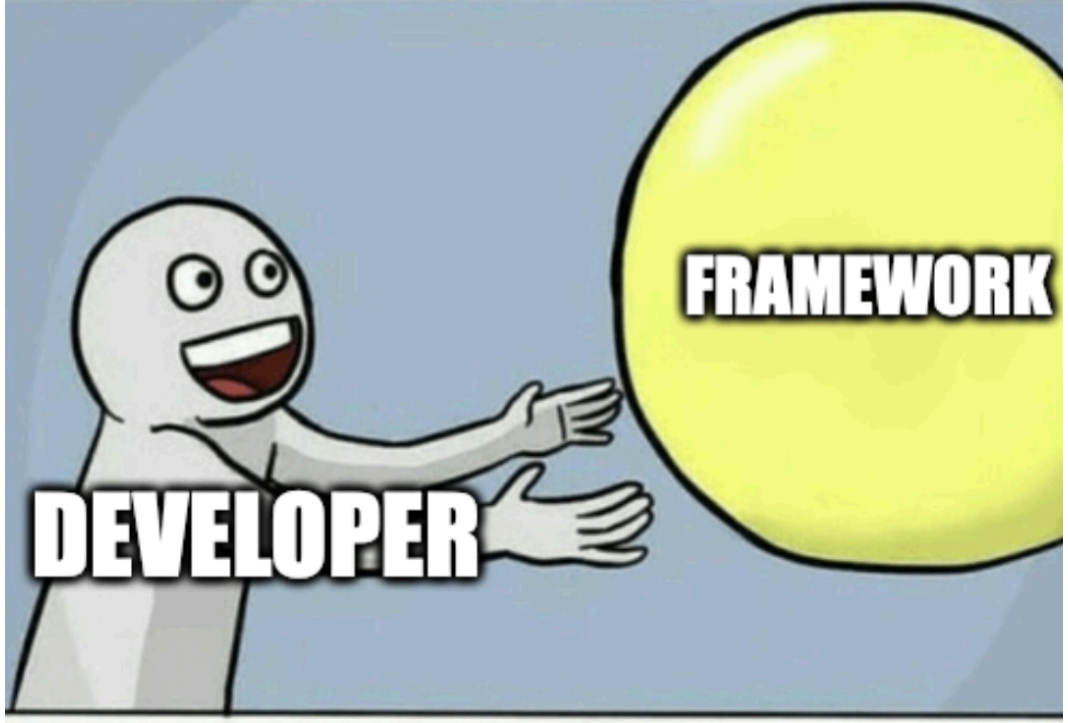
Why should you watch this talk?

About... 40% of my consulting work is migrating Generative AI pipelines from frameworks to pure Python due to their lack of flexibility and "vendor lock-in".

Today I'll try to tell you how you can avoid many problems.

Contact

Simple operations like text interpolations (an `f-string`, that is) are called `StringConcatenationManagerDeluxe`.



All frameworks generate technical debt and "framework lock". But...

how much? how fast?

do you know how to reduce it?*

*without paying

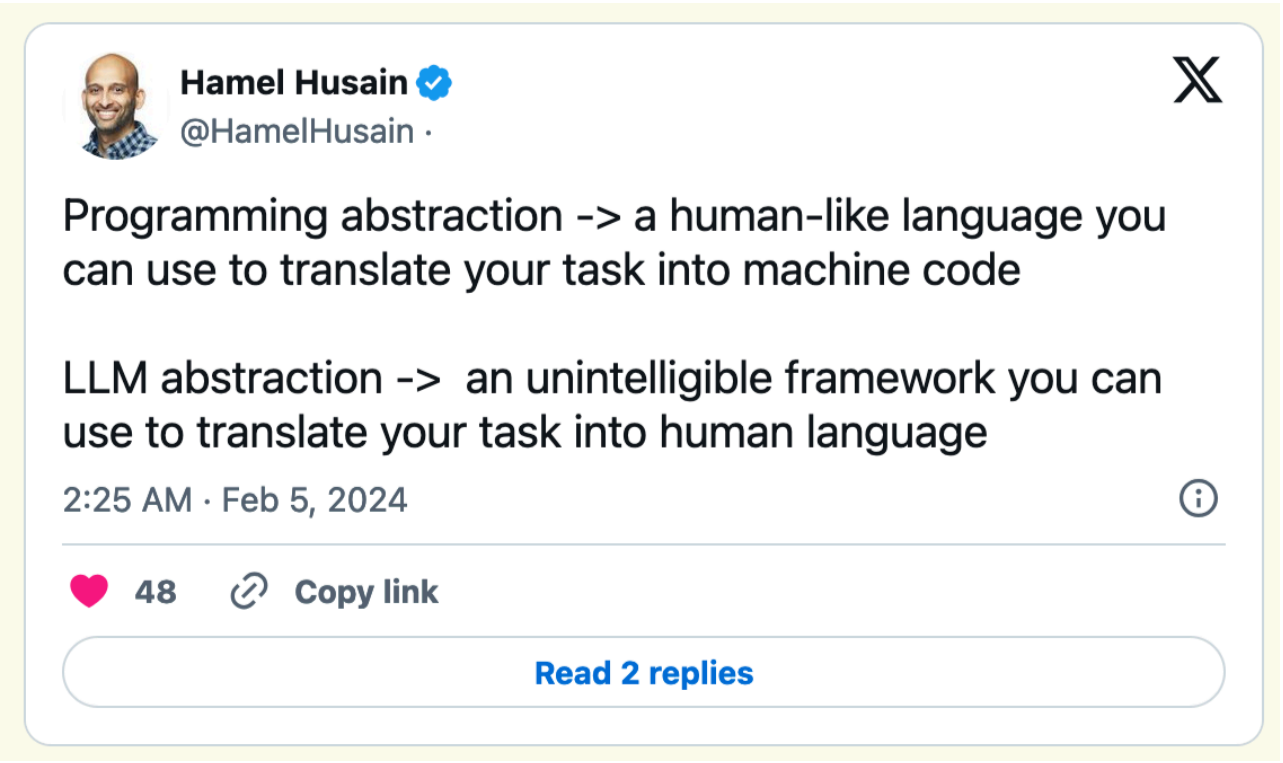
This is not a talk for you to:

- Stop using frameworks
- Deploy all your startup's infrastructure on a raspberry pi in your basement
- Reinvent the wheel

It's a talk for you to:

- Think critically about what tools you should use, what you need, and what you can do for yourself.
- Understand technical debt **before** it becomes a problem.

> "You're exaggerating, aren't you?" - said a kind workshop attendee



Hamel Husain writes an article in early 2024 complaining about this very thing.

His idea is to log OpenAI calls to have a history and be able to "see" what prompts have been used.


LLM DEVELOPER
mitmproxy
IS THIS OBSERVABILITY?

Has the situation improved? (8 months later)

Looking for a simple* observability solution

simple = without servers, proxies, containers. Self-contained in python

Running locally

1. Clone the repository
2. Setup a PostgreSQL instance (version 15 minimu
3. Copy the content of `packages/backend/.env.ex`
4. Copy the content of `packages/frontend/.env.e`
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
```

```
docker run --name langfuse \
-e DATABASE_URL=postgresql://hello \
-e NEXTAUTH_URL=http://localhost:3000 \
-e NEXTAUTH_SECRET=mysecret \
-e SALT=mysalt \
-e ENCRYPTION_KEY=0000000000000000000000000000000000000000000000000000000000000000 \ # generate v
-p 3000:3000 \
-a STDOUT \
langfuse/langfuse
```

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.

Why?

The market has many solutions with a core open-source + SaaS model.

This makes portability between them difficult.

They tend not to be interoperable and "recommend" their own solutions.

Due to rapid development + VC funding ("break things fast", "winner takes all") some frameworks carry technical debt or quickly "hypertrophy".

Why? II

Framework/vendor lock-in is incentivized!

I couldn't find a solution that was simple and self-contained.

In many cases, so much infrastructure and abstraction is not necessary.

Part 0: group therapy for those affected by frameworks

✅ Done

"Framework-lock": Is it worth it? How to migrate?

Preparing the environment

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

1. Use devcontainers or press `.` on the repo page.

   or clone the repo and run:

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

2. Configure the OpenAI API key in the `.env` file (see `.env.example`)

"Hello World" of LLMs: RAG

We're going to migrate a basic RAG pipeline from `langchain`

To reflect on at the end:

- When is it worth using a framework? When should we migrate?
- Which parts should I implement and which should I outsource to a framework?

A basic RAG has the following parts:

1. Loader: obtains the texts we're going to ingest, may include tasks like scraping, downloading, cleaning PDFs, etc.
2. Chunker: divides the texts into chunks of a reasonable size for LLM consumption.
3. Retriever: Indexing and Vectorization of chunks. In this case, we'll use the `chroma` vector database.
4. Prompting: composition of a prompt (aka. text) with the relevant chunks for the answer.

If you're going to migrate your production project, not all parts need to be migrated at once!

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

Let's look at the pipeline in `rag.py`

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

"Cognitive overhead of the framework"/Questions a beginner might ask

- What is the `{"context": ...}` dictionary?
- What is `|`?
- What is `RunnablePassthrough()` for?
- What is `StrOutputParser()`?

What is the `{"context": ...}` dictionary?

Global context for the pipeline. Similar to a `jinja2` context.

What is `|`?

Function composition operator / pipeline operator.

What is `RunnablePassthrough()` for?

Indicates that the `question` value will be a pipeline parameter.

What is `StrOutputParser()`?

Does nothing: "parses"/gets the most likely result from the LLM. There are other more useful ones (e.g.: parsing JSON).

My hypothesis (tested multiple times in real projects). The speed benefit of these frameworks is nullified by the learning cost, non-standard syntax, and "framework lock-in".

We'll try to achieve something like this:

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

Advantages:

- Standard syntax. Easy to understand by non-experts.
- Easy to extend (decorators, magic `__call__`, etc.)
- No need to learn new "jargon".
- All code (including prompts) is versioned in our repository.

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)}"
```

> Easy to understand.
>
> Easy to debug (F11 works!).
>
> Easy to type.
>
> Easy to extend (decoration, magic `__call__`, etc.)
>
> Easy to package (creating prompt modules is no different from creating any other Python module)
>
> Extremely portable -Hi WebAssembly ;)- and even to other languages.

My recommendation: The first thing to migrate are the prompts. It's easy, quickly improves the code, and is a good exercise to understand the problem.

Versioning!: Be careful when using `hub.pull()`, we won't be able to observe changes in the prompt.

`hub.pull()` generates a false sense of security ("the official prompt")

Tip: If you need a prompt repository... use GitHub, it has versioned URLs accessible via HTTP.

You probably don't need it (yet).

The basics:

```
def llm(prompt: str, model: str = "gpt-4o-mini") -> str:
  """Function that calls the LLM. There are other libraries that do this (see comments)"""
  #...

def prompt(docs: list[Document], question: str) -> str:
  """Uses an f-string to format the prompt"""
  #...
```

In v1: We keep the splitter, loader, and Chroma tooling (it's harder to migrate)

```
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()
```

> Don't try to migrate everything at once. I usually keep these three parts, at least in the first versions.
>
> In my day-to-day, I have small libraries that I reuse and implement similar things without so much overhead. (e.g.: chunkers)
>
> It depends on each project: size, technical debt, etc.

To finish...

```
def chatbot(question: str):
    # We get the relevant documents
    docs = retriever.invoke(question)

    # We generate an answer
    return llm(prompt(docs, question))

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

Solution:

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

rag v2: migrating *everything*

It's not always necessary and I don't recommend it in all projects.

I show it to illustrate the possibilities.

Many parts of v2 code should be externalized to libraries for reuse or can be replaced by specialized libraries.

```python
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]):
  """Creates and fills the database with the provided documents."""
  # ...
```

Solution in `solved/rag/v2.py`

The chunker is easily reusable (it can be used as a normal function),

it's not always worth migrating

The same with the loader, although there are many other libraries that do it better (e.g.: `unstructured`)

The scraper in this case is implemented with bs4 as it's very simple but there are tools like `scrapy` for more complex cases.

(My) Conclusions

Many pipelines are easily reimplementable in pure Python with a similar (or smaller!) amount of code

You can still use relevant pieces of the framework (e.g.: `WebBaseLoader` or `RecursiveCharacterTextSplitter`).

You shouldn't remove the entire framework at once: it's usually not worth it.

It's not necessary to use the entire framework: you can keep some pieces

In my experience, the entire pipeline except for retrievers is simpler in pure Python than in the framework.

(My) Conclusions II

"Free" perks when writing in pure Python: observability, caching, timing, debugging becomes trivial vs docker, servers, proxies, etc.

```
from cachier import cachier

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

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

Compared to some of the market solutions, I think it's very worth it.

Show me the prompt

After my frustration with having a **simple** observability solution for projects written by third parties, I decided to write a very simple utility (~400 locs) that patches `openai` to log calls.

In the repository, you have a simplified version

When it's more stable, I'll publish it as an independent library (write to me if you're interested: alex@mindmake.rs)

State of the art in LLM observability

sets up *several* dockers with *several* configurations to have a web interface with a *non-standard* format.

Using monkeypatching

```
import observability.openai
```

That's it :)

SmartLLMChain

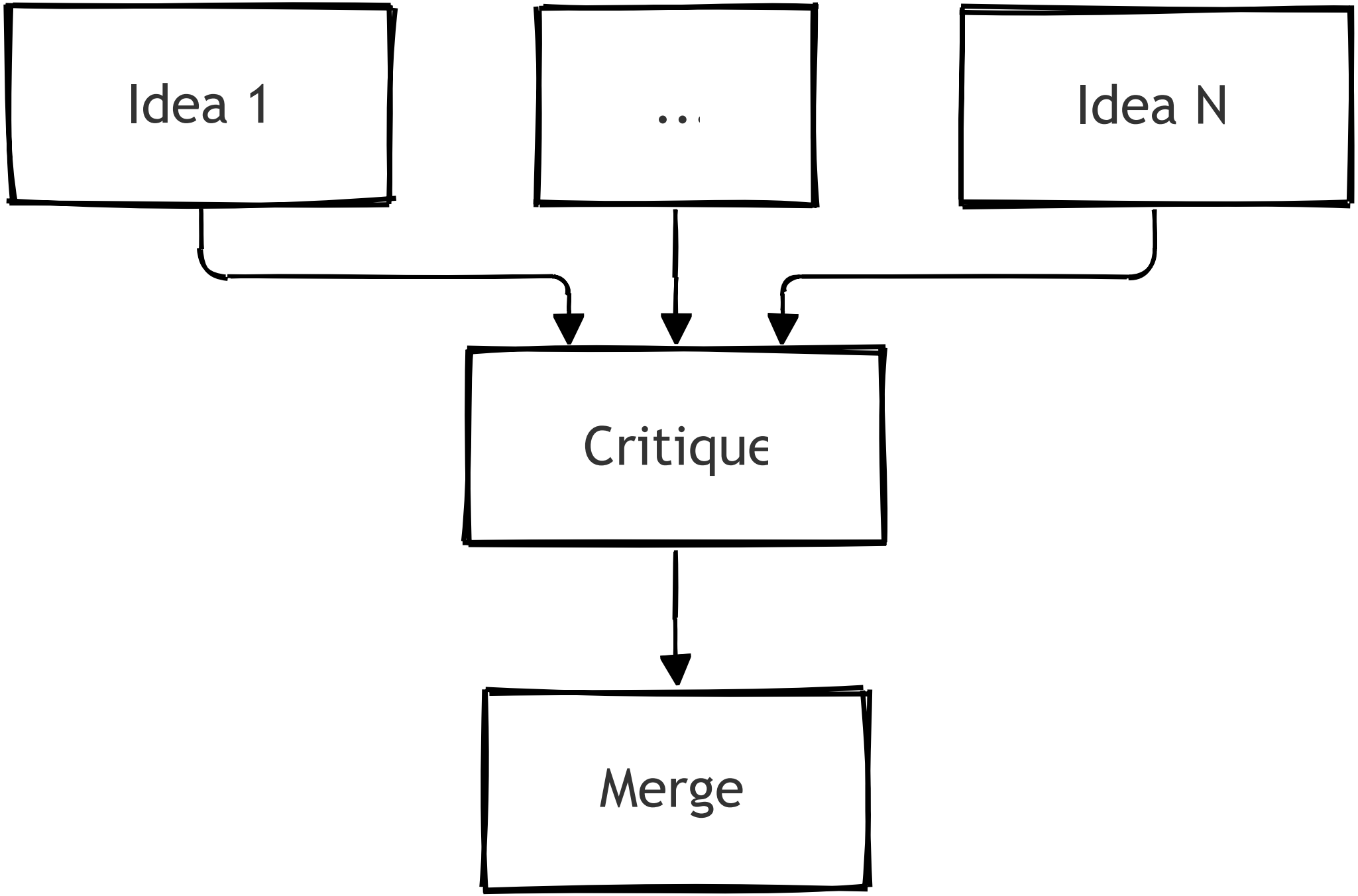
SmartLLMChain uses several techniques to improve the result (self-critique, etc.). But reading the documentation, it's not clear how it works.

This is very common in current frameworks due to the speed of development and ecosystem growth.

SmartLLMChain

```
python -m smartllm "What is the meaning of life?"
```

Task: Add observability to understand how `SmartLLMChain` works to be able to reimplement it later.


Idea 1
...
Idea N
Critique
Merge

SmartLLMChain in a nutshell

```
def idea_prompt(question: str) -> str:
    """Idea generation. Runs N times."""
    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:
    """Critique with the generated ideas."""
    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:
    """We get the final result by combining the ideas and the critique."""
    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))
```

(My) Conclusions III

Practically all "advanced" prompting techniques can be implemented with flow control, functions, coroutines, and a couple of auxiliary functions.

Including agents (we won't see it in this talk).

The code is much more readable, which is critical when working with LLMs that are **very** sensitive to small changes.

From scratch: best practices for developing complex applications

Extractor

We'll reimplement a metadata/field extractor from a document. It's a very useful case for LLMs.

1. We give the LLM a document (e.g.: web page of this talk) and the schema of the data we want to extract: author, title, date, etc.
2. We go through several validation processes to ensure the reliability of the data.
3. In case of error, we show the LLM the error and ask it to correct it.
4. We repeat the process until all validations pass.

Note: I currently recommend using function calling or constrained generation for this case if the provider supports it.

Final result:

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

```
{'links': [{'description': 'Article about problems in Generative AI.',
            'url': 'https://hamel.dev/blog/posts/prompt/'},
           {'description': 'GitHub discussion about frameworks.',
            'url': 'https://github.com/langchain-ai/langchain/discussions/18876'},
           {'description': 'Introductory tutorial using langchain.',
            'url': 'https://python.langchain.com/v0.1/docs/use_cases/question_answering/'}],
 'speaker': 'Alejandro Vidal',
 'technologies': ['OpenAI', 'LLMs', 'langchain'],
 'title': 'GenAI❤️f-string. Developing with Generative AI without black boxes.'}
```

Start with `v1.py` and add:

A function that extracts a JSON block from the LLM output:

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

Use this URL to test: https://pretalx.com/pycones-2024/talk/SKZFHY.ics

Extractor v4 and v5

Taking `v3.py` as a base, add:

- A validator for links that ensures it has this format: `{'description': str, 'url': str}` (solution in `v4.py`)
- A validator that filters technologies with the following criteria:
  - Acronyms are described in parentheses, for example: `IPv6 (Internet Protocol Version 6)`
  - Technologies are written in English

(solution in `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"""From the following list of tags: {techs} verify that
the following conditions are met:
- Acronyms are described in parentheses, for example:
  `IPv6 (Internet Protocol Version 6)`
- They are written in English

Show the errors in a list formatted as a JSON array
with the following format:

``json
[
    "Not in English: Programación en Python, ...",
    ...
]
``

If there are no errors return an empty list.

# Errors

```json
["""
    )

    if output:
        raise ValueError(output)
```
```

Best practices

Prioritize migrating prompts and pipelines. RAGs, loaders, etc. are secondary.

Use specialized libraries for each of these parts (chroma, jinja2, scrapy, unstructured, llamaindex, etc.)

Use frameworks for the first demos/MVPs, but save time to migrate it (it's not as much as it seems).

Best practices II

Keep your APIs simple, typed, and documented. We're doing string concatenation, not rocket science.

- `def llm(prompt: str, model: str, ...) -> str` and `def xxx_prompt(variable: str) -> str` are the most basic.
  - You can have a separate module with the prompts or use a git repository with **text files** or templates if you want something more advanced.
- `def my_pipeline(variable: str) -> str` if you don't need multi-step.
- `async def my_pipeline(variable: str) -> ...` if you want to use coroutines and make it asynchronous.

Best practices III

Don't obsess: you can use frameworks, they're useful, don't reinvent the wheel.

But don't copy and paste the tutorial example without understanding how you can migrate it.

Be critical about which parts are necessary and understand the debt and lock-in you're acquiring.

In LLM code **Readability >>> LOCs**: don't try to make it brief, make it simple.

Complex pipelines are 90% flow control and basic conditionals. Don't be carried away by industry inertia.

Thank you very much!

If you're in a similar situation or are about to start creating your own pipeline, don't hesitate to contact.

Or if you're a company that wants to learn how to use AI ⬇️

💬 Contact

PyCones 24 VIGO