Smart Document: Understanding & AI-Ready API Design

Table of Contents

[Introduction 1](#_Toc201621131)

[Design Decisions 1](#_Toc201621132)

[Backend 1](#_Toc201621133)

[Analyze the Document 2](#_Toc201621134)

[API 4](#_Toc201621135)

[2 AI Power Features 5](#_Toc201621136)

[Feature 1 5](#_Toc201621137)

[Feature 2 5](#_Toc201621138)

[Production Considerations 5](#_Toc201621139)

[Handling LLM API Failures 5](#_Toc201621140)

[Caching Strategy 5](#_Toc201621141)

[Estimated Cost per Document 6](#_Toc201621142)

# Introduction

This document is part of a 3-part project available [in this link](https://github.com/SamGuercho/smart_document).

Below, we will come back on the overall design of the project, the AI-features that could be forked to its objects, and the considerations to make it production grade.

Note that as we had no time for benchmarking, I decided to use an LLM I m used to work with and I have subscription to: OpenAI models.

I did see though that Gemini is giving very good results overall too and is worth testing, and Sonnet 3.7 is an excellent candidate for agentic frameworks.

# Design Decisions

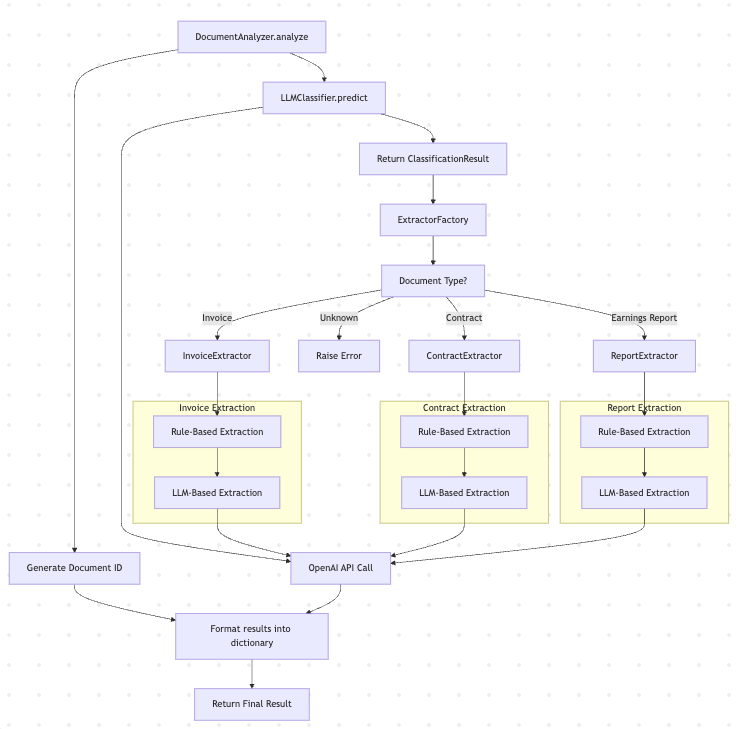
## Backend

2 main goals were achieved in the project:

|  |  |  |
| --- | --- | --- |
| Goal | Description | Components |
| Analyze a document | 1. Classify the document 2. Depending on its classification, extract several fields | DocumentAnalyzer  LLMClassifier  BaseExtractor sub-classes  Dataclasses (ClassificationResult, others) |
| Save the document into a storage | Persist data:   * into a .json format * with a uuid, * in a “data/document\_store” folder | DocumentStore |
| [Mock] Define possible actions to activate in an agentic manner later | Depending on the document\_type, propose some ideas of actions that could be triggered by an LLM | Mock |

### Analyze the Document

Below is the structure of the “Analyze” workflow:



We see here the succession of components being called in a pseudo-workflow.

**Type-based Extraction**

It has been decided to use a factory design on the Extractor to gain in flexibility with the extraction, and better scalability.

**Two-Pronged Extraction Logic**

* Rule-Based Extraction for known, structured patterns (e.g. invoice totals, contract dates).
* LLM-Based Extraction for unstructured data or when layout varies.

Note: I have left the Rule-based part mostly in TODOs, as despite having some results, it was too fuzzy and not scalable to bring anything. For the sake of a clean API, I decided to leave one example commented out and remove the rest.

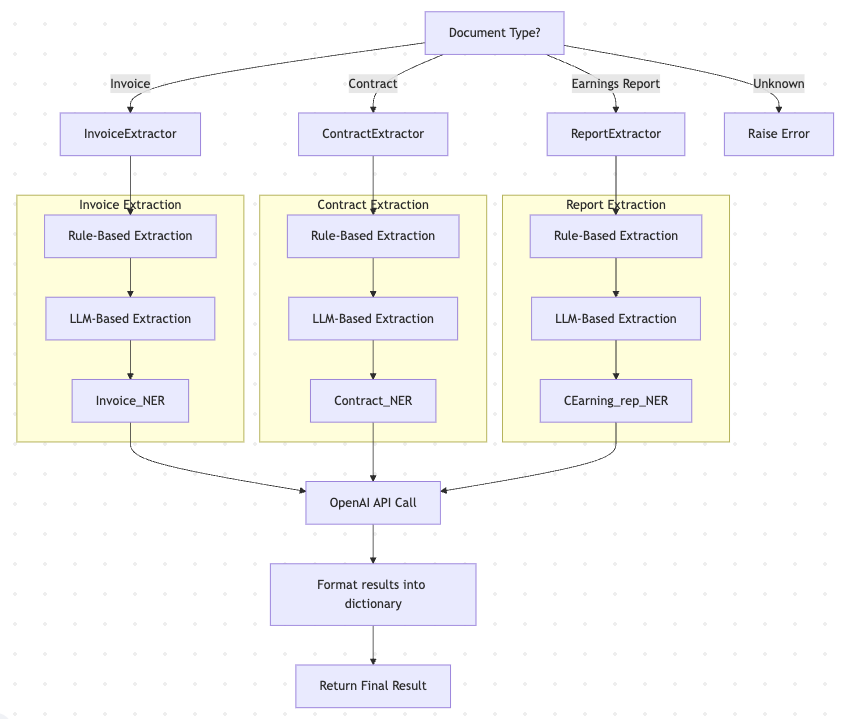
With more time than the scope of this HA, and in real world condition with enough data, I would have envisaged to improve the rule-based method in first for 2 main considerations:

1. Rule-based system is much faster than calling any model, even locally stored on the pod
2. Reducing the number of fields a AI-based NER has to handle increases the quality by reducing the effort of reasoning: less fields to handle for the model means less risk of hallucinating, less tokens to pay attention to and less token to consider while generating.

As for the LLM-based extraction Logic, as we can see on the chart I used OpenAI LLM for time concerns, but in terms of pricing, latency, and computation, it performs, but looks overkill.

Here is what I would have tried with more time and data:

1. **Use a spacy/Llama pure NER model:** Use an encoder model specialized in entity extraction. With enough data to fine-tune, it is possible to teach SOTA models new fields specific to each document type. But as we said, it presents the disadvantage of requiring a fine-tuning, hence data, and usually handles bad different languages, as the models are language specific. It means extra handling of language detection to use a NER specialized in the language detected.
2. **Use a distilled version of a decoder model:** not a real solution, but an in-between in terms of costs and latency. Not ideal solution.

In the case of a fine-tuned, personal NER, the extraction phase becomes the following:  


We can now separate and specialize each NER to its specific task.

**OpenAI GPT-4**

In many ways, GPT-4 simplified a lot the whole process. Among the things that it simplified is the non-requirement of few shot inference. The model understood properly the exercise, and even by trying other templates of pdf found on the internet, generalizes well. It was then more interesting for me to not add few shots for now, to keep the instructions as simple as possible.

An example of few-shot inference would have been to provide a full text of an invoice for instance (they seem to be the lowest text possible) and show the required output.

But in our case, it was more a waste of input token for a pretty standard, JSON serializable output format, and a clear enough instruction to not need it.

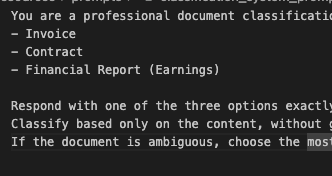
One challenge occurred during the classification process:

The OpenAI model NEVER presented the token “Earning” when it needed to classify an Earning report. Instead, the token “E” was always favored, and the top 20 most probable tokens were all broken tokens (not proper words, spaces, letters, lemmatized parts of words).

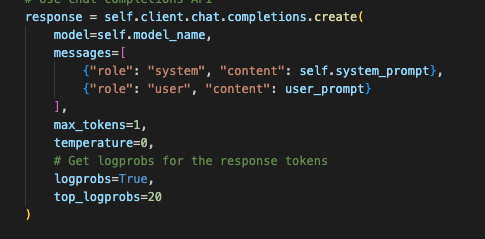
I can’t believe that that OpenAI does not have a token “Earning” when it even built a token “ earnings” (the space is not a typo). Not only should it exist, but its token key should be lower than “ earnings”.

To cope with it in the limited amount of time, I did the following:

I changed the name of the classification to “Financial Report (Earnings):



And I limited the output of the model to only one token:



It then returns only the word “financial” which I can associate to Earnings as the model intended to write Financial Report.

## API

**FastAPI Framework**

I used FastAPI to design the API and expose the endpoints. It was simpler, easy to handle and the direct access to the Swagger allowed me to make sure quickly that all the services exposed were operational. It also combines very well with Pydantic, which is the structure I chose to go with as I extracted the results into dataclasses.

As of today, all the methods are synchronous, but in a real world we would have expected the analyze one to be asynchronous. FastAPI does support it, so it’s not an issue.

If we were to connect with other services in a full-stack way (as I assume it would be), it would be more interesting to move it on to a Django framework.

**Store Data After Analyzing**

I have decided that by design, every analyzed document should be saved, but we can completely add a flag to store, or even separate the service into 2:

1. A service only to analyze
2. A service to store

And expose a new endpoint analyze\_and\_store to do both.

For the sake of the HA, I store the data in .json file, as it was fast and easy to use in our dictionary format. It does present some issues for a production-grade code:

* We can’t index, nor query
* It is not scalable
* It has no protection in terms of neither integrity nor confidentiality.

Depending on where and how much we plan to save, we can consider a simple SQL, a NoSQL or even a vectorDB solution to replace it.

# 2 AI Power Features

## Feature 1

**Handling contract expiration and renewal**

The first thing that comes to mind is helping to handle contract expiration. A more granular system to focus on expired contracts, or on the contrary to ignore them is a powerful feature.

Depending on the conditions of the contract, if it’s a renewable one we can couple the search to actions such as “renew the contract with the same terms”, that would provide a pre-filled contract, ready-to-be-sent to the counter-part (whose information are also in the metadata) to automatically renew it.

## Feature 2

**Semantic search**

Instead of opening every document manually, we could query for instance “How many invoice remain to be paid for [X period]”. Not only could it search properly all the relevant invoices to look up for (status unpaid or partially paid, sorted by amount or by due date), but it could also display meaningful metrics such as the stats of the numerical fields (i.e “Sum: 450,00$”) for all the searched invoices. For this end, We would need to have either a local, or a server-based vector store where the document would have been saved.

Note: Without deduplication, the above feature becomes non valuable.

As the volume of data can be big depending of the environment, I recommend to not only embed the data in a vector store if required, but also to add a min-hashing with fixed hashing functions such as in [my github project here](https://github.com/SamGuercho/Paraphrasing).

With min-hashing, we can approximate the Jaccard similarity of an extensive number combination of texts with a relatively low trade-off. The advantage of a project like in my github is that I fixed the hashing keys, allowing us to embed the text in its min hash and reuse the same hashing to compare with new embedded documents later.

# Production Considerations

## Handling LLM API Failures

To ensure robustness, the 2 following strategies should be considered:

1. Retry and time out system
2. Fallback modes  
   We should be able for example to skip the LLM step in the extraction if the model is unavailable. This would increase the interest of converting as many entities into rule-based extraction as possible  
   It also make the status field valuable: if not all the components have provided their service, we can still return the result, but with a status: partial or error

## Caching Strategy

The things to consider for caching are the following:

1. Which hashing and metadata to store
2. How to store
3. For how much time

For the hashing, we can have 2 different modes:

1. Strict hashing: should be the most common use case, we check that the SHA256 signature of the text we are about to process is already present in our cache.
2. Loose hashing: we do not want to redo an action on a too similar document, use min-hashing.

We can store both with the text and use either of them according to our need.

To store the cache, Redis sounds best for fast consultation.

The TTL really depends on our business requirements, I assume that 7 days represents a whole week of work, it should be enough to keep in cache.

We can also run periodically, let’s say every month/quarter, a cleaning process that would collect the expired documents in cache.

## Estimated Cost per Document

By using GPT-4o, we have the following costs

* Input Tokens: $0.005 per 1000 tokens
* Output Tokens: $0.015 per 1000 tokens

Here is the cost to analyze a document:

|  |  |  |
| --- | --- | --- |
| Step | Token Usage estimation (tkn.) | Estimated Cost |
| Classification | 500 | $0.0035 |
| Metadata Extraction | 1800 | $0.012 |
| TOTAL | **2300** | **$0.0155** |

The cost seems high when you consider the sheer expected volume of analyzed documents. (more than 15k$ for 1M documents analyzed).

We can reduce the cost by:

* Downgrading to GPT-3.5 if the results are comparable
* Implement the fine-tuned open-source NER alternative mentioned earlier
* Put in place a system of routing, for instance by using a cheaper (open-source, rule-based) extraction, check the output, and use GPT-4o ONLY when the result seems ambiguous
* The caching helps reducing the redundant model calls. We can make the caching strategy more aggressive (we could cache for 1 month instead of one week)