# PHASE 3 FEATURES LIST

## 1. PDF Loading and Extraction

For loading and extracting content from PDFs, we used PyMuPDF (fitz).
Before finalizing this, we evaluated other libraries such as pypdf, pdfminer.six, and pdfplumber.

We chose PyMuPDF mainly because of its performance and versatility. It is built on top of MuPDF, which is implemented in C, making it significantly faster than pure Python alternatives. This speed advantage becomes very important when working with large or complex PDF documents.

Another key reason was its all-in-one capability. PyMuPDF allows us to extract text, images, and tables using a single library, including features like find_tables. Additionally, it provides highly accurate bounding box coordinates, which are crucial for our linker logic that associates images with the correct text sections.

The other options were rejected for specific reasons.

- pypdf, although popular, often struggles with complex layouts and does not provide reliable positional information for text blocks.
- pdfminer.six is very precise but is extremely slow, making it impractical for large-scale or real-time processing.
- pdfplumber works well for table extraction but internally depends on pdfminer, inheriting its performance issues.

---

## 2. Text Chunking and Tokenization

For splitting text into meaningful chunks and estimating token counts, we used spaCy (en_core_web_sm). We also considered NLTK, LangChain's RecursiveCharacterTextSplitter, and simple string-based splitting methods.

spaCy was chosen because it is linguistically aware. It understands sentence boundaries, which helps ensure that text chunks do not break in the middle of sentences. This preserves semantic meaning and improves downstream tasks like embedding and retrieval.

It also allows for accurate token estimation, which is far more reliable than counting characters when trying to fit content into model context windows.

The alternatives had clear drawbacks.

- NLTK is more academic in nature and tends to be slower and heavier for production use cases.
- Simple string splitting (by newline or fixed character count) frequently breaks context and leads to poor semantic quality.

- LangChain, while powerful, would introduce unnecessary dependencies and overhead since we only needed a small part of its functionality for text splitting.

---

## 3. Image Processing

For handling images extracted from PDFs, we used Pillow (PIL). We also considered OpenCV (cv2).

Pillow was selected because it is lightweight, simple, and well-suited for basic image operations such as resizing, checking dimensions, and saving files. It integrates smoothly with the image byte streams provided by PyMuPDF, making the workflow clean and efficient.

OpenCV was not used because it is a much heavier library, primarily designed for advanced computer vision tasks like object detection and image filtering. Using it just for basic image handling would have added unnecessary complexity and increased environment setup overhead.

---

## 4. Data Schema and Serialization

For structuring data and serializing outputs, we used Python dataclasses along with native JSON. Alternatives like Pydantic and Marshmallow were also evaluated.

Dataclasses were chosen because they are built into Python, requiring no additional dependencies. They are lightweight and introduce minimal runtime overhead, which fits well with our controlled data generation pipeline.

Libraries like Pydantic were not used because, although they provide excellent runtime validation, they add external dependencies and additional processing overhead. Since all data is generated internally and not coming from untrusted user input, we prioritized simplicity and performance over strict validation.