Websites & PDF Extractor Application

Websites & PDF Extractor Application

Team Members	Contributions
Vedant Mane	33%
Abhinav Gangurde	33%
Yohan Markose	33%

Introduction

The project provides a centralized viewpoint on different varied sophisticated tools ranging from Open source to paid enterprise solutions which would be generally utilized to scrape data for data extractions from different PDF documents and websites.The project helps:

- Examine different results for the same requirement of data extraction,
- Understand the advantages and disadvantages of the tools used from a centralized application point of view
- Access the extracted results in a configured storage of S3 buckets

Users can observe the data extraction results and capabilities of different tools to make an informed decision on which tool to use based on personalized use case requirements.

Technologies Used

- **Streamlit :** Frontend Framework
- **FastAPI :** Backend API Framework
- **Google Cloud Run :** API Deployment
- **AWS S3 :** External Cloud Storage
- **Scrapy :** Website Data Extraction Open Source Tool
- **PymuPDF :** PDF Data Extraction Open Source Tool
- **Diffbot :** Website Data Extraction Enterprise Tool
- **Microsoft Document Intelligence :** PDF Data Extraction Enterprise Tool
- **Docling :** Document Data Extraction/Conversion Tool

Goal: A fully-fledged application that helps the user extract website and pdf data into a structured markdown file and compare the different outputs by accessing the source and output files stored appropriately in the cloud

Problem Statement

The major challenge lies in extracting meaningful information from unstructured sources of PDFs and websites, majorly due to their varied inconsistencies for their data structures and data layouts.

While the PDF documents often save data in visual layouts or images rather than a specific format they require specialized tools for parsing text, tables, and images from these documents. Similarly for web url page scraping requires analyzing varied dynamic contents, images, tables and complex HTML code analysis to extract the required data.

Solutions to extract data from these datasets range from open-source toolkits to paid enterprise level options the user can test these tools and on the metrics of - accuracy, capabilities, performance, ease of use and scalability and make an informed decision based on the appropriate tool sets and their requirements.

Proof of Concept

With the problem dividing the requirement in two major sections of data scraping -

With the tools provided, users have access to a centralized application to test and validate the tools and their individual capabilities to make an informed decision of each individual tools capabilities and compare them against each other in-terms of their accuracy, capabilities, performance, ease of use and scalability.

The Tools We used and Why

Beautiful Soup (Open Source web scraping) : An open source python library that makes it easy to extract data from websites by parsing the HTML or XML document even if it is poorly formatted. A variety of inbuilt functionalities such as extracting specific elements, navigating through the document helps us give granular as well as a general control over what is being scraped and how it scrapes the data. All this and the simplicity in its usage factored into our decision to select this tool for open source web scraping

Diffbot (Enterprise web scraping) : A web scraping tool capable of categorizing websites to classify a type to the website and extract more specific details from them. Their claim is that websites' content are interpreted by a machine learning model trained to identify the key attributes on a page based on its type. And further specific APIs have been developed for these different types.

As requirement doesn't specifically mention the type of urls diffbots categorization provides a good scalability space to further update the data extraction to a more specific level using its classifications.

For testing out the diffbot Extract APIs documentation has been provided for samples utilizing different classifiers to test and showcase their functionalities and data extraction.

PyMuPDF (Open Source pdf extraction) : An open source tool that gives us very granular control over the pdf we want to extract. It enables page py page rendering letting the user decide how to extract data from each page. Additionally, the speed of rendering and extracting images and contents is much higher compared to other open source tools available and also one of the major reasons in selecting this tool.

Microsoft Document Intelligence (Enterprise Tool) : A cloud based Azure AI service provides users with an enterprise level solution for analysis documents, with varied layouts and data structures.

Using their pre- trained models like read and layout enables users to capture data text, layout information for the document.

Docling (Open Source) : An open-source toolkit that is able to transform complex documents like pdf and Doc files, as well as parse diverse formats like HTML. The one main benefit is that it can be implemented in the data extracting and processing pipelines to extract and convert data into a standard format like a markdown file. The tools ability to convert even complex pdf and HTML data even for pdf with diverse formats led us to select this tool for the purpose of conversion to markdown files.

Basic Tests

- **Selecting the open source web scraping tool:** Examined various open source tools including “newspaper”, “scrapy”, and “beautiful soup”. Tested the features, scalability, ease of use and finalised beautiful soup for web scraping
- **Selecting the enterprise web scraping tool:** Examined various enterprise tools including “Microsoft Document Intelligence”, “”, and “Diffbot”. Tested the features, scalability, ease of use and finalised Diffbot for web scraping
- **Selecting the open source pdf extraction tool:** Examined various open source tools including “pypdf”, “pdfplumber”, “tabula”, and “Pymypdf”. Tested the features, scalability, ease of use and finalised Pymypdf for pdf scraping for its speed and granular control
- **Selecting the enterprise pdf extraction tool:** Evaluate the capabilities of Amazon Textract and Azure Intelligent Document. Tested API features, usability, and capabilities showed that Azure Intelligent Document provided better extraction options and scalability.
- **Comparing Docling and Markitdown:** Tested the various features and practicality of the tools and understood what works and what does not
- **Setting up AWS S3 Buckets and testing boto3:** Created amazon S3 buckets and tested the manipulation of the buckets through code using boto3
- **Creating the flow between S3, FastAPI (Backend) and Streamlit (Frontend):** Established a flow between the three components and made sure the code intractability between them is working

Challenges

- Following tables is consolidated challenges faced for each tools set:

Tools	Category	Challenges
Beautiful Soup	Open-source Web Scraping	Limited scalability for large-scale scraping due to lack of built-in crawling or concurrency mechanisms Difficulty in handling dynamic contents such as animated web images
Diffbot	Enterprise Web Scraping	Limitation due to free-tiers access and

		limited control over customization of scraping logic compared to open-source alternatives
PyMuPDF	Open-source PDF Extraction	Although a table extraction feature is available it does not reliably identify tables in a pdf and considered them as lines of text
Azure Document Intelligence	Enterprise PDF Extraction	- Handling complex table structures and nested layouts in PDFs - Limitations due to free-tier (Number of documents and pages scanned limit to 2) - Rigid response structure and PDF structures - Only extracts images metadata for the document
Docling	Open-source Web Scraping Open-source PDF Extraction	- When converting large or complex PDFs to Markdown can take significant time. - Image extraction does not work for Websites, need to use open source toolkit for the same purpose. Docking is able to identify the images but returns a None type object. -

Addressed Challenges

- **Open-source Web Scraping:** Used the HTML structure to maintain order and extracted only the essential contents
- **Open-source PDF Extraction:** Used Pymupdf for block by block extraction of text, links and images
- **For Enterprise PDF Extraction Tools :** Amazon Textract uses synchronous *AnalyzeDocument* or *DetectDocumentText* are limited to processing one page at a time they can be used and processed parallel using asynchronous by adding the documents onto buckets but it adds additional layer of accessibility and implementation issue as compared to Azure AI Document API enables us to use models and scan pages individually.

- **For Diffbot :**

  They provide a standard Analyze API, which categorizes the page into an appropriate type. The classification are article, image, video, discussion, event, or list - using these specific classes further detailed datasets can be configured to extract. They also offer a custom API where users can create entirely new custom extractions by defining rules, all provided in their documentation : https://docs.diffbot.com/reference/extract-introduction

They also provide bulk extraction using Extract API which can be scheduled or automated. Each url that needs to be scraped needs to be provided in a job. Thus, if the use case requirement specifies scraping data from thousands of websites rather than creating a thousand API calls for varied HTML structures a bulk job can be configured to collect data.

Product Type\Plan	Free	Startup	Plus
All APIs	5 Calls Per Minute	5 Calls Per Second	25 Calls Per Second
Extract API	10,000 Calls Per Month	None	None
Crawl / Bulk Extract API	n/a	n/a	25 Active Jobs, 1000 Total Jobs

Pricing analysis :

The free tier account for a new user is provided with 10,000 credits per month at no cost. A small scale requirement can easily be handled by the same.

Additionally previewing the provided costing plans gives more leverage for bulk extraction which is accessible in the paid plans only with increased credits and rate limits.

Thus for reference if we wanna take 10000 webpages (1 credit per webpage) which is also the limit for a free account one would :

- For a free account -

10000 webpages / 5 calls per min = 2000 minutes which is around 33.33 hours at 0$

- For a Startup Plan -

10000 webpages / (5 * 60) calls per min = 33.33 minutes costing around $299

Additional with no limitations on further calls but the same also doesn't include bulk extraction feature.

- For a Plus Plan -

10000 webpages / (25 * 60) calls per min = 6.67 mins at $899 costing plan,

With no limitations on further calls and bulk bulk extraction feature jobs to configure larger jobs.

For Azure Intelligent Document : We provide the option of 2 pre-handled models - read and layout to handle text extraction along with the potential images, table and layout information if the document has the same or needs to be collected.
Additional to the Document Intelligent azure service we also explored on Azure AI Vision for image-to-text capabilities by converting PDF to images, observed results are similar to that of read or layout models from Intelligent Document.

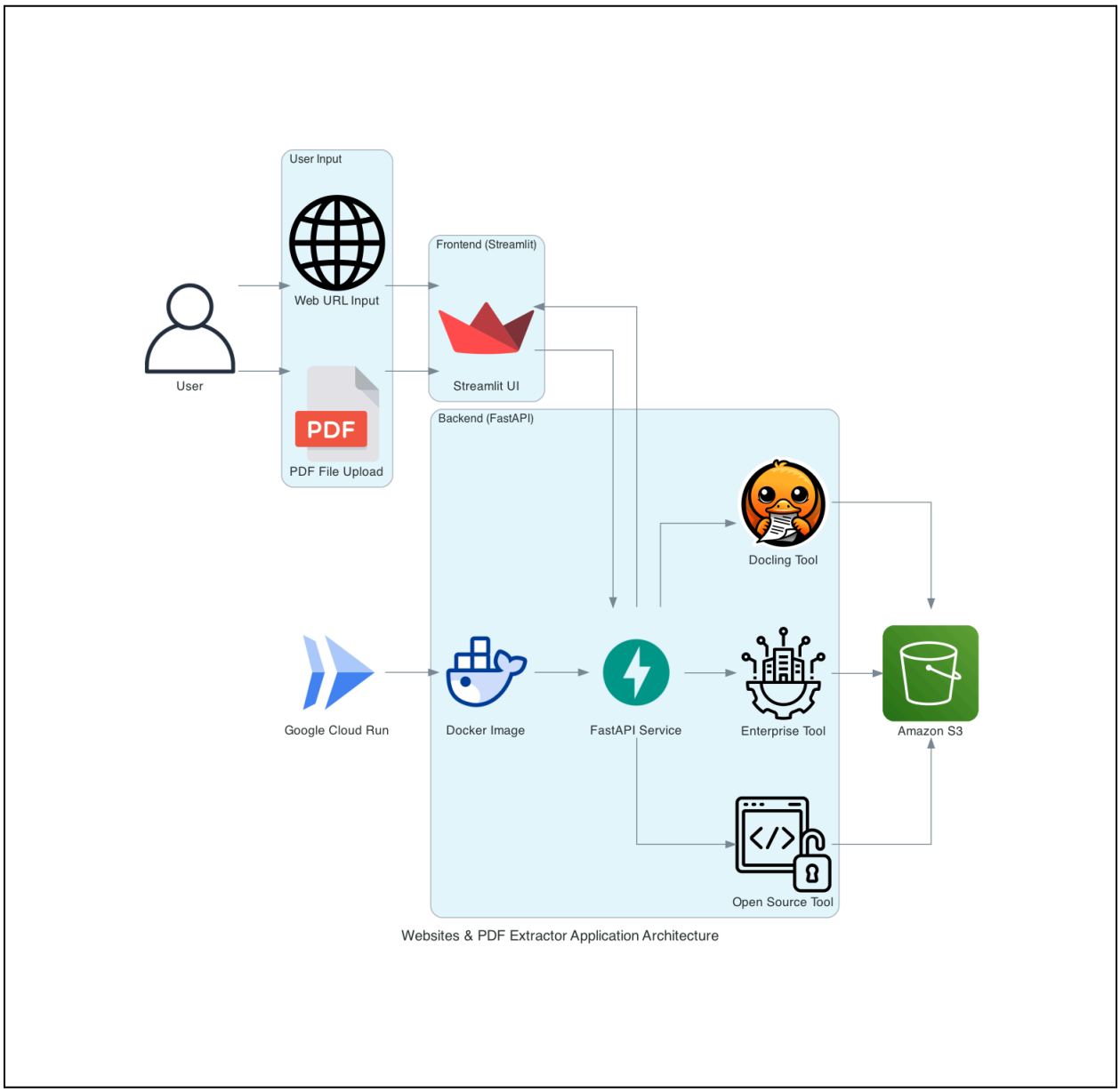
Limitation for Azure AI :

1. The documentation service for the pre-trained models expects a specific structure/layout for the document.
2. Free-tier accounts are limited with the number of pages that can be scanned for data as well as the number of pages in a single document are limited to only 2.
3. Supported file size: the file size must be less than 50 MB and dimensions at least 50 x 50 pixels and at most 10,000 x 10,000 pixels
4. The AI Documentation provides the option in the “prebuild-layout” model to extract the images from the in cropped png but the render images are almost indecipherable - thus we are only collecting the metadata from documents containing images.

   https://azuresdkdocs.z19.web.core.windows.net/python/azure-ai-documentintelligence/latest/index.html#extract-figures-from-documents

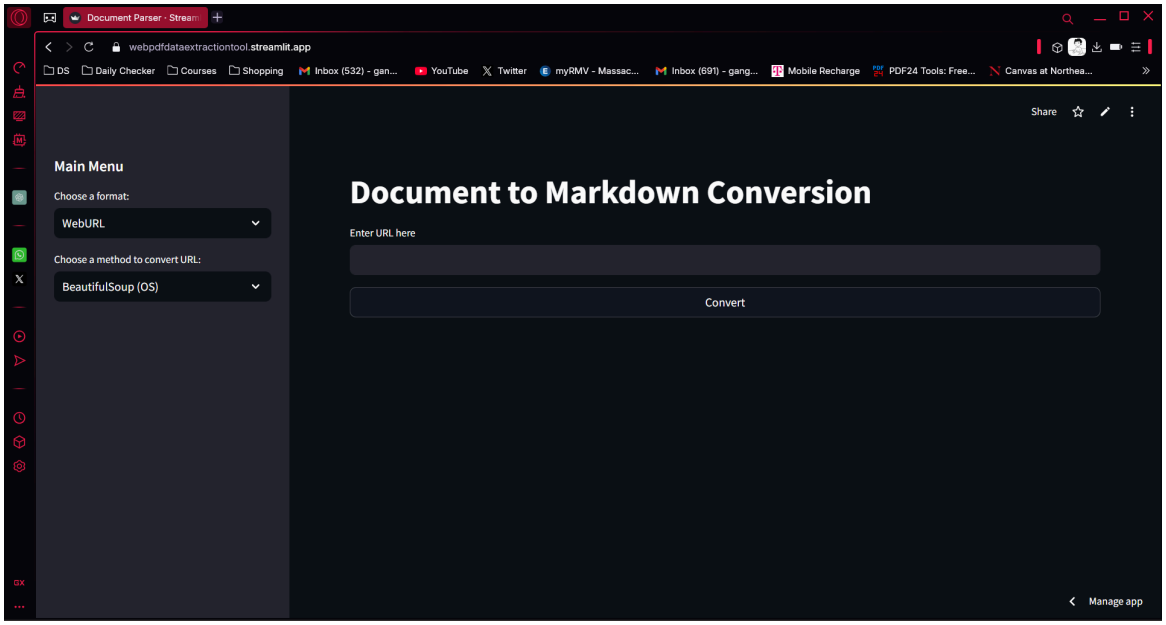
   https://learn.microsoft.com/en-us/answers/questions/2116806/can-i-extract-images-form-documents-using-azure-ai

Architecture Diagram



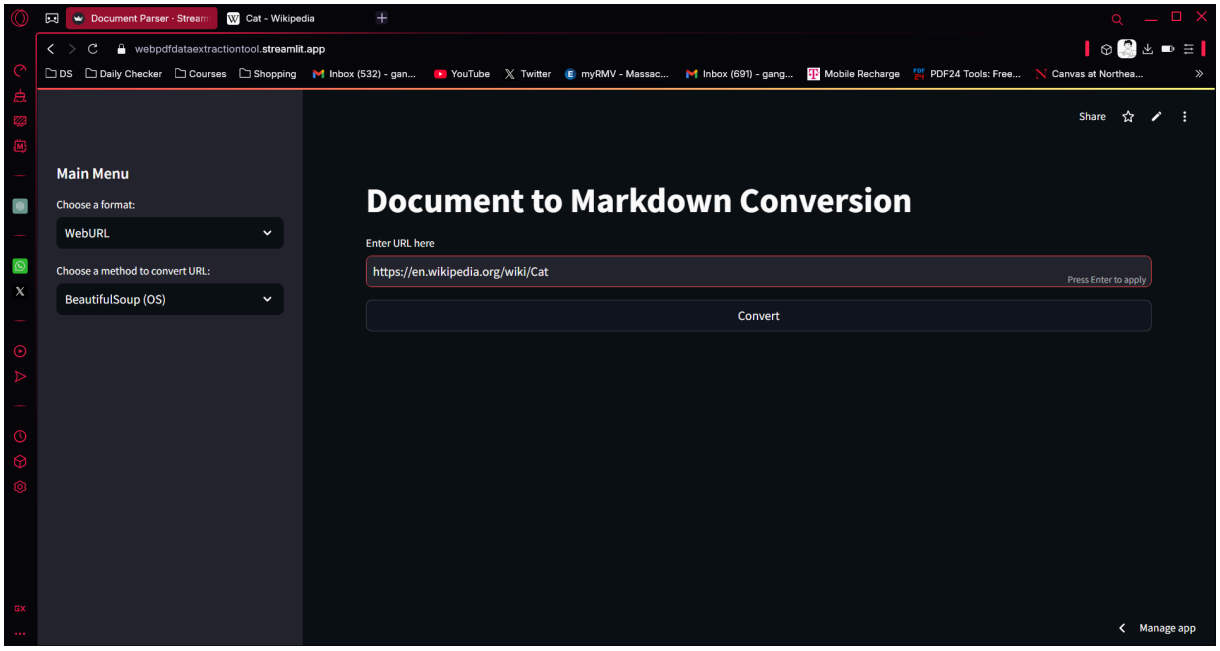
Walkthrough of the Application

Landing Page for the application - Provides an entry point for the user to access the tools and services offered with the option to select a PDF document or scrape through a webpage url.



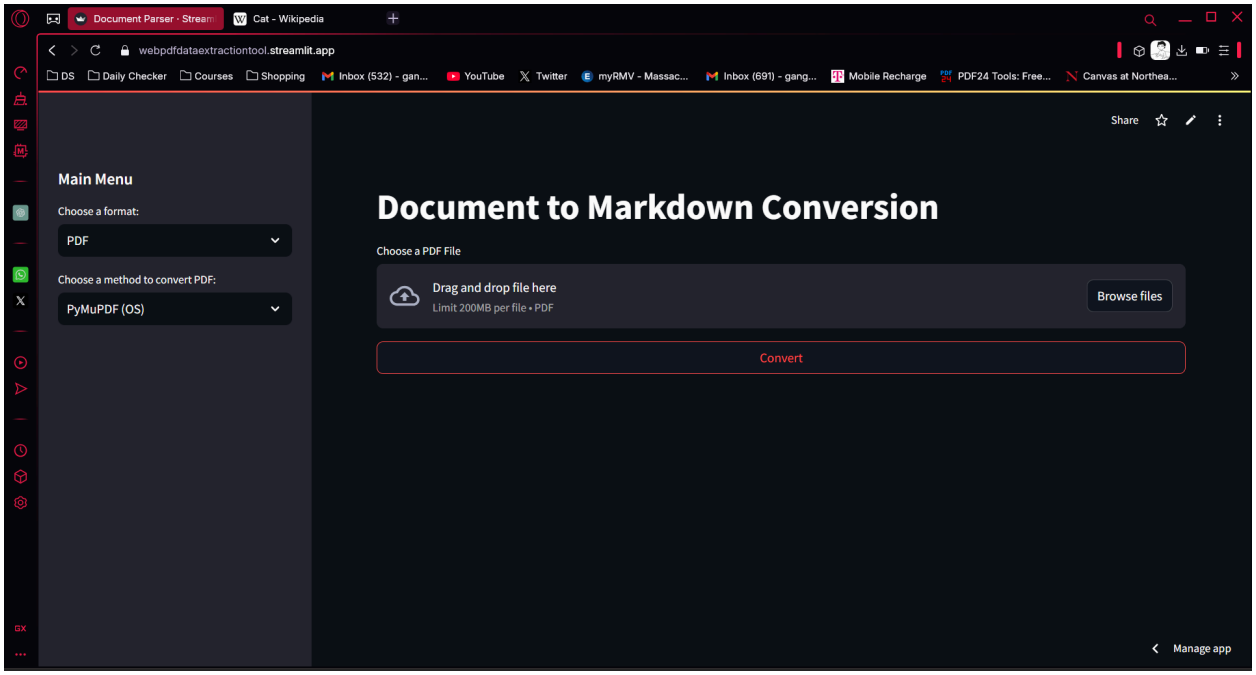
Option 1 : WebURL - Selecting the option of web urls user can parse through web scraping tools like : BeautifulSoup (Open Source), Diffbot (Enterprise Tool) and Docling (Open Source)

And the user needs to paste a web url that he wants to scrape data from and click on the “Convert” Button.
eg : https://en.wikipedia.org/wiki/Cat

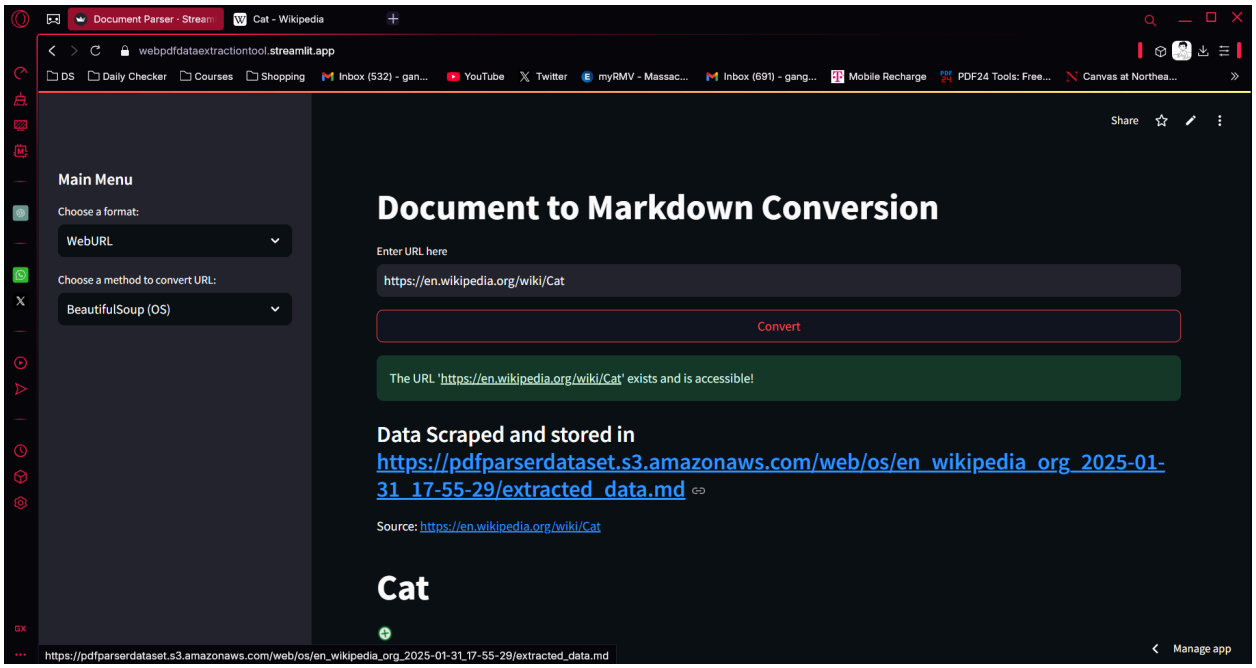


Option 2 : PDF - Other option from the drop down provides the user to select a PDF file to upload for data parsing with the tool options of - PyMuPDF (Open Source), Azure Intelligent Document (Enterprise Tool) and Docling (Open Source).

User needs to upload a pdf format file using the “Browse file” option and click on the “Convert” button.



After Data Scraping : Post the conversion process is successful for the selected tool the user can access the converted markdown .md file via the provided link.
Additionally the same file can be accessed in the configured S3 buckets storage space with the file identifiable using the directory path.



Users can select and try all the options and compare the results to make an educated decision on the tools capabilities and use case requirements for the tools.

Application Workflow (Data Engineering Work + Code Explanation)

Frontend - Streamlit

```python
import time
import streamlit as st
import requests, os, base64

API_URL = "https://fastapi-service-rhtrkfwlfq-uc.a.run.app"

def main():
    # Set the title of the app
    st.title("Document to Markdown Conversion")
    # Add a sidebar
    st.sidebar.header("Main Menu")
    input_format = st.sidebar.selectbox("Choose a format:", ["WebURL", "PDF"])

    if "text_url" not in st.session_state:
        st.session_state.text_url = ""
    if "file_upload" not in st.session_state:
        st.session_state.file_upload = None

    if input_format == "WebURL":
        st.session_state.file_upload = None
        tool = st.sidebar.selectbox("Choose a method to convert URL:",
                                    ["BeautifulSoup (OS)", "Diffbot (Enterprise)", "Docling"])
        st.session_state.text_url = st.text_input("Enter URL here")
        convert = st.button("Convert", use_container_width=True)
    elif input_format == "PDF":
        st.session_state.text_url = ""
        tool = st.sidebar.selectbox("Choose a method to convert PDF:",
                                    ["PyMuPDF (OS)", "Azure Document Intelligence (Enterprise)", "Docling"])
        if tool == "Azure Document Intelligence (Enterprise)":
            radio = st.radio("Choose a model :", ["Read", "Layout"])
        else:
            radio = None
        st.session_state.file_upload = st.file_uploader("Choose a PDF File", type="pdf", accept_multiple_files=False)
        convert = st.button("Convert", use_container_width=True)

    # Define what happens on each page
    if convert:
        if input_format == "WebURL":
            if st.session_state.text_url:
                if check_url(st.session_state.text_url):
                    st.success(f"The URL '{st.session_state.text_url}' exists and is accessible!")
                    convert_web_to_markdown(tool, st.session_state.text_url)
                else:
                    st.error(f"The URL '{st.session_state.text_url}' does not exist or is not accessible.")
            else:
                st.info("Please enter a URL.")

        elif input_format == "PDF":
            if st.session_state.file_upload:
                st.success(f"File '{st.session_state.file_upload.name}' uploaded successfully!")
                convert_PDF_to_markdown(tool, st.session_state.file_upload, radio)
            else:
                st.info("Please upload a PDF file.")
```

- The user inputs the source from which he wants to extract the data. The selection (Url/pdf) is stored in the secession state of streamlit, which will have a string or fileupload option accordingly
- This session state is then used to send the respective source, along with the tool being used (Chosen by the user) and send it for extraction by calling the respective function
- Separate functions have been created for pdf and web conversion each of which will use the required api calls (with respective endpoints) depending on the chosen tool

Backend FastAPI

```python
@app.post("/scrape_url_os_bs")
def process_url(url_input: URLInput):
    md_result = os_url_extractor_bs.scrape_to_markdown(url_input.url)
    timestamp = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
    folder_name = url_to_folder_name(url_input.url)
    base_path = f"web/os/{folder_name}_{timestamp}/"
    s3_obj = S3FileManager(AWS_BUCKET_NAME, base_path)
    s3_obj.upload_file(AWS_BUCKET_NAME, f"{s3_obj.base_path}/web_url.txt", str(url_input.url))

    # md_result = convert_json_to_markdown(json_result) # Scrape the URL
    s3_obj.upload_file(AWS_BUCKET_NAME, f"{s3_obj.base_path}/extracted_data.md", str(md_result))

    return {
        "message": f"Data Scraped and stored in https://{s3_obj.bucket_name}.s3.amazonaws.com/{s3_obj.base_path}/extracted_data.md",
        "scraped_content": md_result
    }
    You, 4 days ago • Added the initial code for backend - fast api
# OS PDF - Pymupdf
@app.post("/scrape_pdf_os")
def process_pdf_os(uploaded_pdf: PdfInput):
    pdf_content = base64.b64decode(uploaded_pdf.file)
    # Convert pdf_content to a BytesIO stream for pymupdf
    pdf_stream = BytesIO(pdf_content)
    timestamp = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
    base_path = f"pdf/os/{uploaded_pdf.file_name.replace('.','').replace(' ','')}_{timestamp}/"
    s3_obj = S3FileManager(AWS_BUCKET_NAME, base_path)
    s3_obj.upload_file(AWS_BUCKET_NAME, f"{s3_obj.base_path}/{uploaded_pdf.file_name}", pdf_content)
    file_name, result = pdf_os_converter(pdf_stream, base_path, s3_obj)
    return {
        "message": f"Data Scraped and stored in https://{s3_obj.bucket_name}.s3.amazonaws.com/{file_name}",
        "scraped_content": result
    }
```

,

- The API (Backend) contains specific end points for each type of data extraction that needs to be done.
- Depending on the the user selection, the appropriate endpoint is called and the extraction logic for that particular endpoint is performed
- Each time a user needs an extraction, the source for the same is being uploaded to the s3 bucket in their respective folders for future reference
- Each endpoint logic contains the respective process functions thats being called to process the source given by the user, process it and uploaded the final markdown file to S3 along with its source.
- The extracted data is now being send back to streamlit to display it in markdown format for the user

Storage S3 Buckets

```python
class S3FileManager:
    def __init__(self, bucket_name, base_path=''):
        self.s3 = boto3.client('s3', aws_access_key_id=AWS_ACCESS_KEY_ID,
                               aws_secret_access_key=AWS_SECRET_ACCESS_KEY,)
        self.bucket_name = bucket_name
        self.base_path = base_path.strip('/')

    def list_files(self, prefix=''):
        full_prefix = f'{self.base_path}/{prefix}'.strip('/')
        response = self.s3.list_objects_v2(
            Bucket=self.bucket_name,
            Prefix=full_prefix
        )
        return [obj['Key'] for obj in response.get('Contents', [])]

    def upload_file(self, bucket_name, file_name, content):
        self.s3.put_object(Bucket=bucket_name, Key=file_name, Body=content)

    def get_presigned_url(self, object_name, expiration=3600):
        full_path = f'{self.base_path}/{object_name}'.strip('/')
        try:
            url = self.s3.generate_presigned_url(
                'get_object',
                Params={
                    'Bucket': self.bucket_name,
                    'Key': full_path
                },
                ExpiresIn=expiration
            )
            return url
        except Exception as e:
            print(f"Error generating presigned URL: {e}")
            return None

    def upload_with_retry(self, file_path, bucket_name, object_name=None, max_attempts=3):
        config = transfer.TransferConfig(
            multipart_threshold=1024 * 25,  # 25MB
            max_concurrency=10,
            multipart_chunksize=1024 * 25,
            use_threads=True
        )
```

- The output markdown files, images and source files are all being stored on s3 once the documents are processes
- The functions defined in our s3 class has multiple features for bucket manipulation using boto3 including upload feature, which takes in the bucket name, the base path and actual content as argument an uploads them into the required path

Features (Extraction Functions)

Open Source Url Extraction

```python
def scrape_to_markdown(url):
    md_content = []

    response = requests.get(url)
    response.raise_for_status()
    soup = BeautifulSoup(response.text, 'html.parser')

    # Removing unwanted elements
    for tag in soup.find_all(['script', 'style']):
        tag.decompose()

    md_content.append(f"Source: {url}\n\n")

    main_content = soup.find('main') or soup.find('body')

    # Finding the necessary data (headings, images content and tables)
    for element in main_content.find_all(['h1', 'h2', 'p', 'img', 'table'], recursive=True):
        if element.name == 'h1':
            md_content.append(f"# {clean_text(element.text)}\n\n")

        elif element.name == 'h2':
            md_content.append(f"## {clean_text(element.text)}\n\n")

        elif element.name == 'p':
            text = clean_text(element.text)
            if text:
                md_content.append(f"{text}\n\n")

        elif element.name == 'img':
            src = element.get('src', '')
            if src:
                # Handling relative URLs
                if not src.startswith(('http://', 'https://')):
                    src = urljoin(url, src)
                alt = element.get('alt', 'image')
                md_content.append(f"![{alt}]({src})\n\n")
                    You, 18 hours ago • Updated the UI and Open source scraper function…
        elif element.name == 'table':
            # Process table headers
            headers = []
            for th in element.find_all('th'):
                headers.append(clean_text(th.text))

            if headers:
                md_content.append('| ' + ' | '.join(headers) + ' |\n')
                md_content.append('| ' + ' | '.join(['---'] * len(headers)) + ' |\n')

            # Process table rows
            for row in element.find_all('tr'):
                cols = []
                for td in row.find_all('td'):
                    cols.append(clean_text(td.text))
                if cols:  # Only write non-empty rows
                    md_content.append('| ' + ' | '.join(cols) + ' |\n')
            md_content.append('\n')

    return "".join(md_content)
```

- The function takes in a url (source provided by the user) and parses the html content using Beautiful Soup, html parser.
- Since the html content is structured, we loop through the structure and append the necessary element contents in a a list (md_content) formatting it accordingly
- While looping through the structure, we have cleaned some data in the text as well as handled relative urls for proper embedding to markdown
- This then joins the all the elements in the appended list and return one full string that is in markdown format

Open Source PDF Extraction

```python
def pdf_os_converter(pdf_stream, base_path, s3_obj):
    doc = pymupdf.open(stream=pdf_stream, filetype="pdf")
    md_content = []

    for page_num in range(len(doc)):
        page = doc[page_num]
        # Extract text and headings
        text = page.get_text("dict")
        for block in text["blocks"]:
            if "lines" in block:
                for line in block["lines"]:
                    for span in line["spans"]:
                        if span["size"] > 12:  # Assuming headings have larger font size
                            md_content.append(f"## {span['text']}\n\n")
                        else:
                            md_content.append(f"{span['text']}\n\n")

        # Extract images
        for img_index, img in enumerate(page.get_images(full=True), start=1):
            xref = img[0]
            base_image = doc.extract_image(xref)
            image_bytes = base_image["image"]
            image_ext = base_image["ext"]
            image_s3_path = f"images/page{page_num+1}_img{img_index}.{image_ext}"

            s3_obj.upload_file(s3_obj.bucket_name, f"{s3_obj.base_path}/{image_s3_path}", image_bytes)

            md_content.append(f"![Image](https://{s3_obj.bucket_name}.s3.amazonaws.com/{s3_obj.base_path}/{image_s3_path})\n\n")

        # Extract links
        links = page.get_links()
        for link in links:
            if "uri" in link:
                md_content.append(f"[Link]({link['uri']})\n\n")

        # Extract tables
        tables = page.find_tables()
        try:
            for table in tables:
                for row in table.extract():
                    md_content.append(" | ".join(row) + "\n")
                md_content.append("\n")
        except Exception as e:
            print(f"Error processing Tables: {e}")
            You, 3 days ago • Created pdf open source endpoint
        # Join all markdown content
    final_md_content = "".join(md_content)

    timestamp = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
    md_file_name = f"{s3_obj.base_path}/extracted_{timestamp}.md"

    s3_obj.upload_file(s3_obj.bucket_name, md_file_name ,final_md_content.encode('utf-8'))
    return md_file_name, final_md_content
```

- Pymupdf can read and process a pdf page by page converting it to blocks that contain dictionaries with text, image, link and table elements
- Making use of this feature,we loop through the blocks for each page

- The main challenge was to extract contents from the pdf in an ordered manner. It would have been easy to extract all the text once and then the images but making use of the blocks we were able to extract each content in the appropriate order
- For each type of elements the functions processes it and appends it in the respective format into the md_content list
- Once all pages have been process, the list is joined together to form one string in the markdown format

Enterprise Url Extraction (Diffbot)

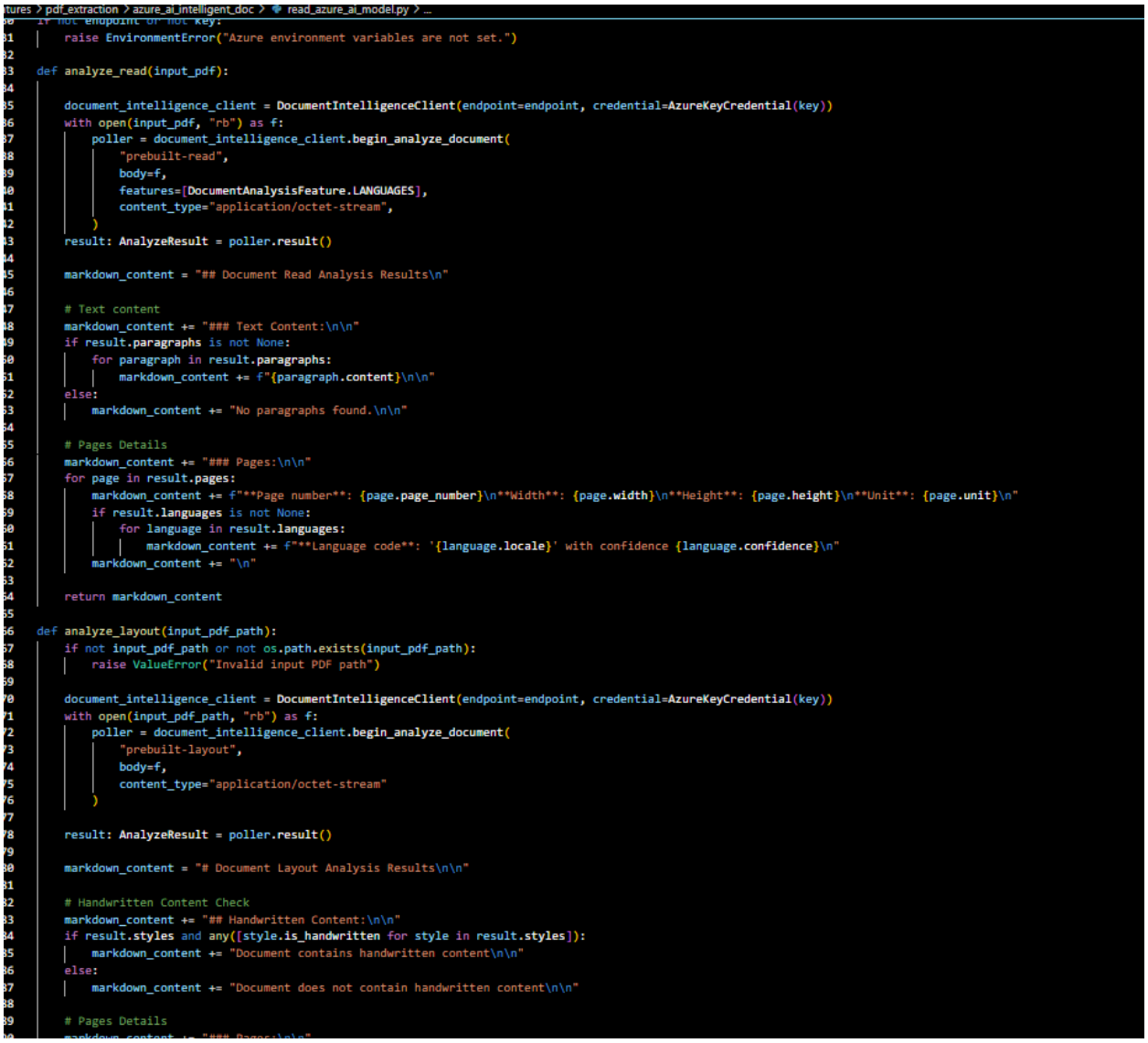
- To utilize the diffbot API services diffbot provides and consolidates reference samples in the link. Using the same base we configure the client and the API response connection.
- The Extract API service for the client using the provided function extracting the data from a generalized standpoint API providing the function
- The response provided contains a defined structured response from the same we have configured to extract only the 'title', 'text', 'images', 'pageUrl', 'type' objects and render the same content onto a .md file to save and display as the response.
- Additionally both the generated .md and provided links are being saved onto a S3 buckets.

```python
@app.post("/scrape_diffbot_en_url")
def diffbot_process_url(url_input: URLInput):
    diffbot = DiffbotClient()
    token = DIFFTBOT_API_TOKEN
    url = url_input.url
    api = "analyze"
    response = diffbot.request(url, token , api, fields=['title', 'text', 'images', 'pageUrl', 'type'])
    if isinstance(response, str):
        try:
            response = ast.literal_eval(response)
        except Exception as e:
            print(f"Error converting response: {e}")
        exit()
    objects = response.get("objects", [])
    extracted_data = extract_for_markdownrender(objects)
    # Generate Markdown content
    markdown_content = f"# Diffbot Extracted Content\n\n"
    markdown_content += f"**Date:** {datetime.now().strftime('%A, %B %d, %Y, %I:%M %p %Z')}\n\n"
    for item in extracted_data:
        markdown_content += f"### Title : \n{item['title']}\n"
        markdown_content += f"### Page URL\n[{item['pageUrl']}]({item['pageUrl']})\n"
        markdown_content += f"### Identified Page Type\n[{item['type']}]({item['type']})\n"
        markdown_content += f"### Text Extracts \n{item['text']}\n"
        markdown_content += f"### Images Extracts \n"
        if 'images' in item and isinstance(item['images'], list):
            for image in item['images']:
                image_url = image['url']
                image_title = image['title']
                markdown_content += f"{image_title} : ![{image_title}]({image_url})\n\n"
        else:
            markdown_content += f"## Images Extracts : \n No Images Found\n"
    file_name = "diffbot_scraped_url.md"

    timestamp = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
    folder_name = url_to_folder_name(url_input.url)
    base_path = f"web/ent/{folder_name}_{timestamp}/"
    s3_obj = S3FileManager(AWS_BUCKET_NAME, base_path)
    s3_obj.upload_file(AWS_BUCKET_NAME, f"{s3_obj.base_path}/web_url.txt", str(url_input.url))
    s3_obj.upload_file(AWS_BUCKET_NAME, f"{s3_obj.base_path}/extracted_data.md", str(file_name))
```

Enterprise PDF Extraction (Azure Intelligent Documentation)

- The Azure Intelligent Documentation services in the application have been configured for 2 pre-trained models which users can select from the application.
- With the limitations mentioned, images or complex data structures like tables are not captured by the read model - primarily configured for data text extraction from documents. Layout model along with the data extraction is able to capture identified images we in our application are collecting only metadata for their location polygons.
- The Response provided from the APIs is captured and manually render into a structured format to display the results from the document scans.
- Additionally the results in generated .md and original pdf is stored in S3 and an accessible link is provided.



Docling APIs:

```python
# Web Docling
@app.post("/scrape-url-docling")
def process_docling_url(url_input: URLInput):
    response = requests.get(url_input.url)
    soup = BeautifulSoup(response.content, "html.parser")
    html_content = soup.encode("utf-8")
    html_stream = BytesIO(html_content)

    # Setting the S3 bucket path and filename
    html_title = f"URL_{soup.title.string}.txt"
    timestamp = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
    print(html_title)
    base_path = f"web/docling/{html_title.replace('.','').replace(' ','').replace(',','').replace("|","").replace('+','')}_{timestamp}/"

    s3_obj = S3FileManager(AWS_BUCKET_NAME, base_path)
    s3_obj.upload_file(AWS_BUCKET_NAME, f"{s3_obj.base_path}/{html_title}", BytesIO(url_input.url.encode('utf-8')))
    file_name, result = url_docling_converter(html_stream, url_input.url, base_path, s3_obj)
    print(base_path)
    print(file_name)
    return {
        "message": f"Data Scraped and stored in https://{s3_obj.bucket_name}.s3.amazonaws.com/{file_name}",
        "scraped_content": result  # Include the original scraped content in the response
    }


# PDF Docling       You, 1 second ago • Uncommitted changes
@app.post("/scrape_pdf_docling")
def process_pdf_docling(uploaded_pdf: PdfInput):
    pdf_content = base64.b64decode(uploaded_pdf.file)
    # Convert pdf_content to a BytesIO stream for pymupdf
    pdf_stream = BytesIO(pdf_content)
    timestamp = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
    base_path = f"pdf/docling/{uploaded_pdf.file_name.replace('.','').replace(' ','')}_{timestamp}/"
    s3_obj = S3FileManager(AWS_BUCKET_NAME, base_path)
    s3_obj.upload_file(AWS_BUCKET_NAME, f"{s3_obj.base_path}/{uploaded_pdf.file_name}", pdf_content)
    file_name, result = pdf_docling_converter(pdf_stream, base_path, s3_obj)
    return {
        "message": f"Data Scraped and stored in https://{s3_obj.bucket_name}.s3.amazonaws.com/{file_name}",
        "scraped_content": result  # Include the original scraped content in the response
    }
```

URL API:

- The API fetches the HTML content from the URL, encodes it, and converts it into a byte stream
- It uploads the HTML URL link to pre-defined path in the S3 bucket
- Then the HTML byte stream is passed to a converter function for further processing
- Returns a success message with the S3 URL and processed content

URL PDF:

- The API decodes the file content and converts it into a byte stream
- It uploads the PDF content to pre-defined path in the S3 bucket
- Then the PDF byte stream is passed to a converter function for further processing
- Returns a success message with the S3 URL and processed content

Docling URL Extraction

```python
def url_docling_converter(web_stream, base_url, base_path, s3_obj):

    # Prepare pipeline options
    pipeline_options = PdfPipelineOptions()
    pipeline_options.do_ocr = True
    pipeline_options.do_table_structure = True
    pipeline_options.images_scale = 2.0
    pipeline_options.generate_page_images = True
    pipeline_options.generate_picture_images = True

    # Initialize the DocumentConverter
    doc_converter = DocumentConverter(
        allowed_formats=[InputFormat.HTML],
        format_options={
            InputFormat.PDF: PdfFormatOption(
                pipeline_options=pipeline_options,
            ),
        },
    )

    web_stream.seek(0)
    with NamedTemporaryFile(suffix=".html", delete=True) as temp_file:
        # Write the PDF bytes to a temporary file
        temp_file.write(web_stream.read())
        temp_file.flush()
        print(Path(temp_file.name))

        # Convert the PDF file to markdown
        timestamp = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
        md_file_name = f"{s3_obj.base_path}/extracted_{timestamp}.md"
        conv_result = doc_converter.convert(temp_file.name)
        final_md_content = conv_result.document.export_to_markdown(image_mode=ImageRefMode.REFERENCED)
        temp_file.seek(0)

        soup = BeautifulSoup(temp_file.read(), "html.parser")
        #print(soup.prettify('utf-8'))
        for img_tag in soup.find_all("img"):
            img_url = img_tag.get("src")
            if img_url:
                # Handle relative URLs
                if not img_url.startswith("http"):
                    img_url = requests.compat.urljoin(base_url, img_url)
                if '<!-- image -->' in final_md_content:
                    final_md_content = final_md_content.replace(
                        '<!-- image -->', f"\n![image]({img_url})\n", 1
                    )
                else:
                    final_md_content = final_md_content + f"\n![image]({img_url})\n"
                #print(img_url)

    # Upload the markdown file to S3
    s3_obj.upload_file(s3_obj.bucket_name, md_file_name ,final_md_content.encode('utf-8'))

    # Return the markdown file name and content
    return md_file_name, final_md_content
```

- Set up options for processing the Website, including OCR, table structure extraction, image scaling, and generating page/picture images.
- Writes the HTML byte stream to a temporary file for processing and ensures it is ready for conversion.
- Converts the temporary HTML file into Markdown format using docling.
- Use BeautifulSoup to extract and replace images in the markdown as docling is unable to extract Image data from the HTML file
- Upload the converted Markdown file to an S3 bucket and returns the file name along with its extracted content

Docling PDF Extraction

```python
def pdf_docling_converter(pdf_stream: io.BytesIO, base_path, s3_obj):

    # Prepare pipeline options
    pipeline_options = PdfPipelineOptions()
    pipeline_options.do_ocr = True
    pipeline_options.do_table_structure = True
    pipeline_options.images_scale = 2.0
    pipeline_options.generate_page_images = True
    pipeline_options.generate_picture_images = True

    # Initialize the DocumentConverter
    doc_converter = DocumentConverter(
        allowed_formats=[InputFormat.PDF],
        format_options={
            InputFormat.PDF: PdfFormatOption(
                pipeline_options=pipeline_options,
            ),
        },
    )

    pdf_stream.seek(0)
    with NamedTemporaryFile(suffix=".pdf", delete=True) as temp_file:
        # Write the PDF bytes to a temporary file
        temp_file.write(pdf_stream.read())
        temp_file.flush()
        print(Path(temp_file.name))
        # Convert the PDF file to markdown
        timestamp = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
        md_file_name = f"{s3_obj.base_path}/extracted_{timestamp}.md"
        # doc_stream = DocumentStream({"name": md_file_name, "stream": pdf_stream})
        conv_result = doc_converter.convert(temp_file.name)

        # timestamp = datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
        # md_file_name = f"{s3_obj.base_path}/extracted_{timestamp}.md"
        final_md_content = conv_result.document.export_to_markdown(image_mode=ImageRefMode.EMBEDDED)

        # Upload the markdown file to S3
        s3_obj.upload_file(s3_obj.bucket_name, md_file_name ,final_md_content.encode('utf-8'))

    # Return the markdown file name and content
    return md_file_name, final_md_content
```

- Set up options for processing the PDF, including OCR, table structure extraction, image scaling, and generating page/picture images.
- Writes the PDF byte stream to a temporary file for processing and ensures it is ready for conversion.
- Converts the temporary PDF file into Markdown format using docling.
- Upload the converted Markdown file to an S3 bucket and returns the file name along with its extracted content.

Directory Structure

```
/DAMG7245_Assignment01/
├── backend
│   └── app
│       └── main.py
├── frontend
│   └── app.py
├── features
│   ├── pdf_extraction
│   │   ├── os_pdf_extraction.py
│   │   ├── azure_ai_intelligent_doc
│   │   │   └── read_azure_ai_model.py
│   │   └── docling_pdf_extractor.py
│   └── web_extraction
│       ├── os_url_extractor_bs.py
│       ├── diffbot_python_client
│       │   └── diffbot_client.py
│       └── docling_url_extractor.py
├── services
│       └── s3.py
├── requirements.txt
├── README.md
├── .dockerignore
├── .env
└── .gitignore
```

References

Streamlit documentation

FastAPI Documentation

Scrapy Documentation

PyMuPDF Documentation

Diffbot Documentation

Microsoft Document Intelligence Documentation

Docling Documentation

Disclosures

WE ATTEST THAT WE HAVEN'T USED ANY OTHER STUDENTS' WORK IN OUR ASSIGNMENT AND ABIDE BY THE POLICIES LISTED IN THE STUDENT HANDBOOK

We acknowledge that all team members contributed equally and worked to present the final project provided in this submission. All participants played a role in crucial ways, and the results reflect our collective efforts.

Additionally we acknowledge we have leveraged use of AI along with the provided references for code updation, generating suggestions and debugging errors for the varied issues we faced through the development process.AI tools like we utilized:

- ChatGPT
- Preplexity
- Github Copilot
- Claud

Team Members	Contributions
Vedant Mane	33%
Abhinav Gangurde	33%
Yohan Markose	33%