Learning to Make Documents Barrier-free and Accessible: From Prototype to Testing-Framework (Practical Work: PDFstral/Hoffman Projects)

Jack Heseltine *
for: Institut für anwendungsorientierte Wissensverarbeitung (FAW),
Institut Integriert Studieren
and Institute of Machine Learning (IML)
Johannes Kepler University (JKU)
Altenberger Str. 69, 4040 Linz
jack.heseltine+academic@gmail.com

Abstract

This Practical Work Report details the selection and configuration of a modern GenAI (Generative AI)/ECM (Enterprise Content Management) framework for optimizing the PDF document processing required to annotate and tag content to make it readable for screenreader and related software. The goal is to create a usable, portable module, on the one hand, where this Practical Work effort was also supported - by way of introduction the Alfresco ECM system and practical experience with the technology - by FAW GmbH, a company specializing in ECM product customizations in Hagenberg, Austria, and interested in applications of this type of system, and to provide a setup for efficient document processing and evaluation at a more involved level, leading to a Masters Thesis at JKU in Linz. Test runs on an initial dataset demonstrating the applicability of LLMs (Large Language Models) and particularly a self-hosted setup in addition to an API-based one complete the proof of concept, for using LLMs in either the few-shot in-context learning or the fine-tuning regime, both with and without RAG (Retrieval Augmented Generation), to make documents barrier-free and accessible, round out the report, with a technical focus on document handling at the Java ECM level and LLM connectivity and Python environment interoperability, for establishing a platform for experiments in this emerging document transformation regime. Current enterprise-level tools (mainly: Hyland Alfresco, Adobe PDF Services and IBM Docling, all in various flavors of open source software) are considered. An unexpected question encountered in the course of this work is the use of LLMs for previously manual scoring tasks, like Logical Reading Order and Color Contrast (given recent model capabilities), at a preliminary phase to the main work.

Research Question (Story) Summary (Practical Work Report Requirement)

1. **What is the central question?**
   The central question addressed in this report is: How can Generative AI (GenAI) methods be configured and deployed within an Enterprise Content Management (ECM) system to make PDF documents barrier-free and accessible for users relying on assistive technology, and specifically are the Alfresco system, compatible enterprise language software modules (in Java) and an API-based LLM workflow feasible for doing document transformation of this kind? (This is the relevant industry partner aspect.) This question is tightly connected with

*with special thanks to Mag. Daniel Jabornig (FAW GmbH) and Angel Borroy López (Hyland)

Preprint. Under review.

efficient and effective PDF ingestion, i.e., loading the relevant PDF structures into memory for further processing.

2. **Why is this question important?**
   This question is important because the challenge of making complex documents accessible is critical for inclusivity, especially for individuals using screen readers or assistive technology. Given the vast amounts of unstructured information managed by enterprises, a GenAI approach for accessibility could transform document processing workflows on a global scale.
3. **What evidence/data (variables) are needed to answer this question?**
   The evidence/data needed includes:
   - Document structure and metadata extracted from PDF files.
   - Accessibility features such as image descriptions, text-to-speech outputs, and structural representations inside PDF document code.
   - Performance metrics from initial test runs using GenAI models like Mistral's Pixtral 12B, or posing the question about how to evaluate quality for free form outputs, potentially in combination with label data. (Subject focus for the Masters Thesis.)
4. **What methods are used to get this evidence/data?**
   The methods used include:
   - PDF structure visualization using libraries such as PyMuPDF [24] and pikepdf [11] at the prototype stage, for example, and PDF-analysis and accessibility checking tools.
   - API-based interactions with GenAI models, like Mistral, for image description and structure generation.
   - Use of the Alfresco ECM system to integrate and test GenAI functionalities within an ECM context: any logging data available.
5. **What analyses must be applied for the data to answer the central question?**
   The analyses applied include:
   - Verification of structure and metadata extraction for consistency and completeness.
   - Quality analysis beyond baseline requirement fulfillment.
   - Comparative analysis of API-based and self-hosted GenAI setups for scalability and integration within ECM systems, time and space permitting (practical question for applications).
6. **What evidence/data (values for the variables) were obtained?**
   Initial test data included:
   - High-level ASCII representations of document structures generated through Mistral's API, individual image descriptions from API calls for testing the Python-language prototype.
   - Test outputs showing a cohesive GenAI and ECM integration for accessible document processing, at a larger scale.
   - The basic enterprise language (Java) code base for implementing such a pipeline.
7. **What were the results of the analyses?**
   The analyses showed that the GenAI models, particularly when deployed with the Alfresco ECM integration, provided robust document structure representations, high-quality image descriptions, and general performance, affirming the feasibility of GenAI-driven document accessibility in general, and specifically, that this setup can (1) collect training documents, (2) build a solid software framework foundation for advanced techniques like RAG, and (3), provide a simple UI for handling document transformations at various scales.
8. **How did the analyses answer the central question?**
   The analyses confirmed that integrating GenAI into an ECM system could make documents more accessible by providing structure visualization, image descriptions, and the like, all in a modular, scalable way suitable for enterprise deployment. It was also discovered that newer tools like Docling allow for sufficient low-level document manipulation capabilities, suggestion that this is a viable level in the program stack to this part of the work (rather than, say, at the Java level). This lead to a general overview of what the architecture of the software required to start testing GenAI accessible document generation performance should look like.

9. **What does this answer tell us about the broader field?**
   This answer suggests that GenAI has a strong potential to improve document accessibility in enterprise settings, offering a framework that could be applied to a wide range of unstructured document types, and contributing to a new paradigm of accessibility-focused content management, but requires strong validation as well, with a focus on minimal requirements as well as quality metrics. This background is outside of the scope of this practical work, however, focusing on the technical foundation instead, and developing it all the way up to a working version for each of a pure in-context, a fine-tuning, and a RAG prototype with live PDF data (direct structural access to the document in the appropriate software) and touching on the topic of on-premises vs. remote deployment (feasibility and data security here, mainly).
10. **Did the paper answer the question satisfactorily? Why (not)?**
    N.a. for the Practical Work Report/no paper, see Seminar Report submitted in Winter Semester 2023/24.

1 Introduction

The challenge of making complex documents accessible to a diverse audience, including those using screen readers, has become increasingly pertinent as enterprises manage vast amounts of unstructured information. Inspired by Angela Carter's narrative in The Infernal Desire Machines of Doctor Hoffman, where reality is reshaped by the projection of desires, this project seeks to employ modern Generative AI (GenAI) techniques within an enterprise content management (ECM) framework to transform documents for accessibility.

Unlike these Hoffman machines, the present machine wants to develop GenAI for the good, in this case making documents accessible in a fully automated manner: this concern is international, and so the UK as well as the Austrian context is considered from the perspective of a London Hackathon project on a Python stack, informing the first part of this work, the prototype, leading to the main work and setup in enterprise languages and Docker, as a testing framework for future work.

The results from initial test-runs provide a proof of concept for using advanced AI techniques to fulfill the promise of reshaping the way knowledge is made accessible, echoing the transformative power illustrated in Carter's fictional world. The focus will be to demonstrate extensibility and solidity of the chosen approach and system, on top of demonstrating that this domain even works as an LLM application.

1.1 PDFstral.london

This hackathon project forms a valuable introduction to the topic of API-based LLM-service interfacing with PDF-data, in a Python stack in this case: the packages used were pikepdf for metadataextraction, pymupdf and pymupdf4llm for PDF document-manipulation, as well as the technologies poetry and streamlit for managing dependencies and staging a python-driven website, respectively.

The result was a Single Page Application (SPA) easily shown in one screenshot (Figure 1) to cover the functionality of both structural analysis of the PDF document dropped into the app and a markdown and speech audio file conversion, to simulate an end-to-end, screen reader oriented workflow: however, provisioning of the barrier-free PDF documents remains the focus of this work, not screen reader (AI).

The LLM-service used is Mitral AI, and specifically Pixtral 12B, a multimodal model. The use of the helper functions abstracting the Mistral AI API call shows how both text and images can be elaborated by the LLMs in the new regime. For text which is then put into further processing, the approach seen in the repo source code is this:

```
extract_code_between_triple_backticks(query_pixtral(f"Provide just the
text of an ascii tree representation of the following document structure:\n
{ascii_structure}"))
```

`traverse_structure_ascii` is a recursive function to tackle such a text processing task in the conventional way, and the LLM layer is added on top, demonstrating how these paradigms work together.

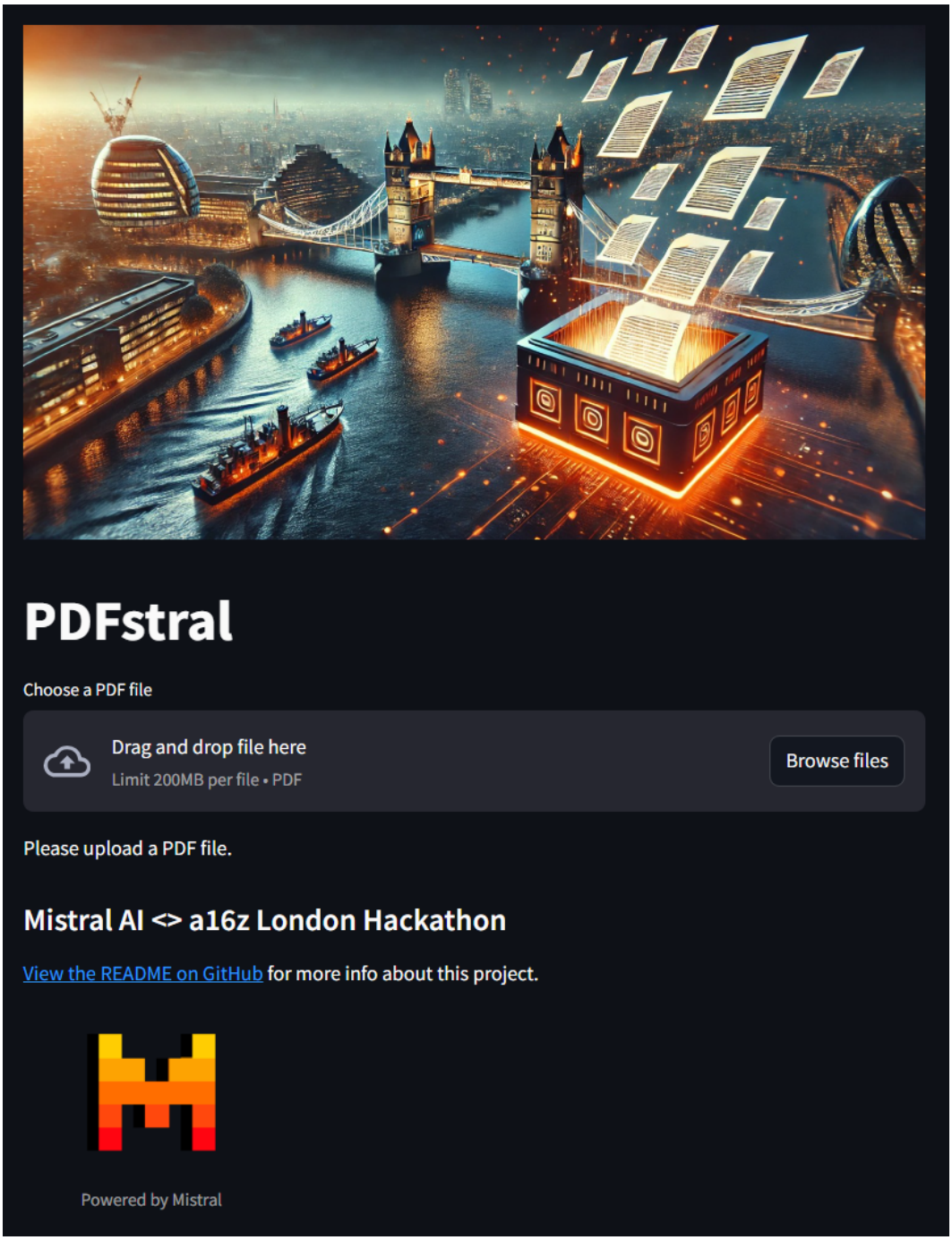

Figure 1: Mistral AI x a16z London Hackathon 2024 "PDFstral.london" project, available online at PDFstral.london

Multimodal image interaction is implemented by a simple `descriptions = query_pixtral(f"Describe these images:", encoded_images)`. Especially this latest capability is of interest to this accessibility project due to the image-content relevance to accessibility, considering the WCAG (Web Content Accessibility Guidelines) 2.0 level AA requirement for contrast ratios of at least 4.5:1 for normal text and 3:1 for large text, for example, and a related technical standard. A fully automated system on the basis of multimodal systems is conceivable, along with context aware image annotation, of course.

The helper function syntax for working with API calls will be covered in more detail in Section 2 to form the technical introduction to the main practical work detailed here as well. One central implementation idea is to not just provide transformation endpoints, but also checking capabilities.

1.2 Context and Thanks

The context of this Practical Work is provided by FAW GmbH, specializing in ECM system customizations, and specifically Mag. Daniel Jabornig, whom I would like to thank for the suggestion to use the GenAI Stack approach to an Alfresco ECM integration and the exposure to practical work with Alfresco [1] - this fits well with the need to organize and process documents at scale for a Masters Thesis project on the topic of "Learning to Make Documents Barrier-free and Accessible," since this environment may serve as a testing-platform for the various LLM approaches currently being researched globally.

The practical upshot of this work is a GenAI module ready to work on this kind of task in the Alfresco-PDF context, since this work will demonstrate loose coupling of the LLM functionality, among other things. I would also like to thank Angel Borroy López (Hyland) for practical guidance in realizing the module. In his role as Technical Evangelist he demonstrated to me how document aspects ([5]), transaction objects and the Python LLM-caller level all connect in Alfresco, but more importantly, how a living developer community around a product category can help adapt to technical advancements affecting the whole ecosystem.

2 LLM-API Workflows: Two-Path-Idea and PDFstral as a Simple Example

This section details the core workflow of the PDFstral application, which leverages multiple APIs and libraries to process PDF documents and extract accessible information. The primary functions are organized around structure visualization, content extraction, and multimodal interactions for making documents more accessible.

2.1 The Initial Idea: Two Paths

Before jumping into it, the seed idea of sending PDF-sourcecode to an LLM for annotation should be documented: In the following section it will be demonstrated that Docling, a PDF-processing python package, is well suited for both OCR (Optical Character Recognition) and native PDF tasks. The output is either markdown or text based on PDF. This output presents the possibility of forming a new PDF document, fully accessible, that has little to do in format and appearance with the original PDF document - call this **Path 1, or new-document-generation**.

In contrast a **Path 2, or document-enrichment**, would be the one where the original PDF document is manipulated. Before tackling this problem a more naive approach demonstrating current Python package capabilities for various text representations, neither a new PDF document or an enriched version, is now described in this section.

2.2 Structure Visualization with ASCII Representation

In PDFstral, the sample toy project exploring a practical component on purely Python package level, the function `visualize_pdf_structure_ascii` (`Listing 2`) uses `pikepdf` to read the PDF structure. If a Structure Tree Root is present, the code recursively traverses the structure using the function `traverse_structure_ascii`, which translates the structural elements into an ASCII representation. This is then sent to the Mistral API for further processing: together this demonstrates that a structural representation of the document can already be generated with minimal overhead, suggesting structure tree tagging with reliable annotations may be within reach.

```
ascii_structure = traverse_structure_ascii(struct_tree_root, 0)
mistral_response = extract_code_between_triple_backticks(
    query_pixtral(f"Provide just the text of an ASCII tree representation
        of the following document structure:\n{ascii_structure}")
)
```

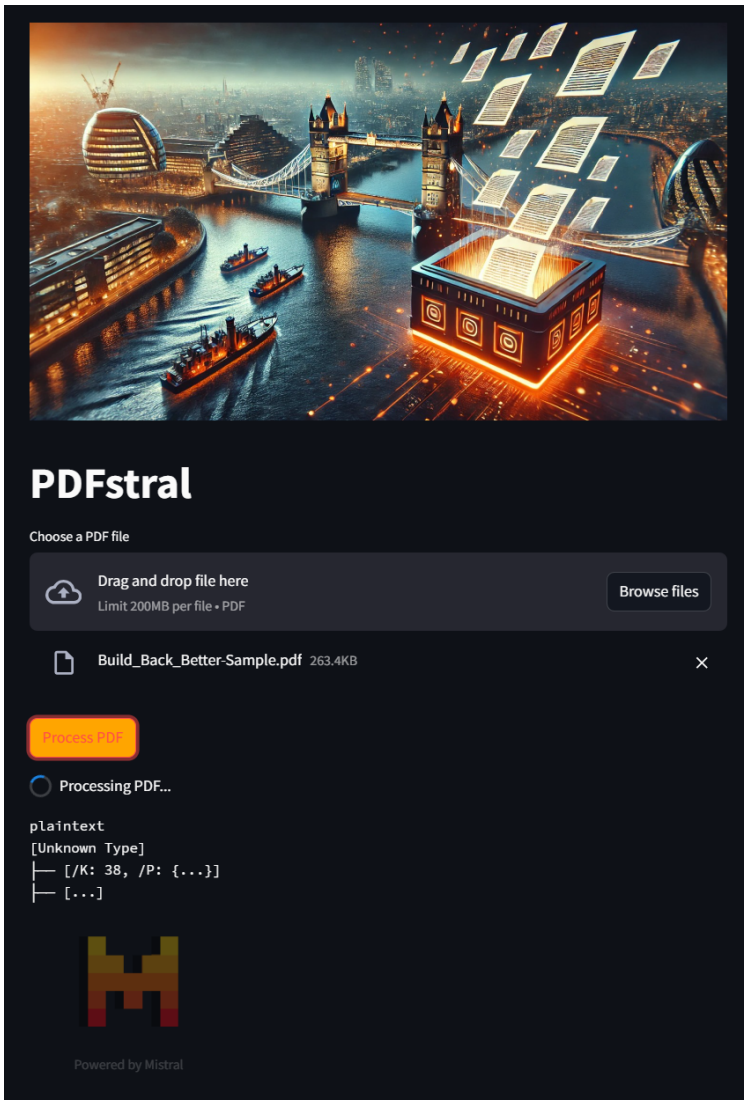

Figure 2: Sample ASCII structure output being loaded and processed by Mistral AI API

2.3 Image Extraction and Multimodal Interaction

Image extraction could turn out to be a key feature in making documents accessible. PDFstral uses `PyMuPDF` to identify and extract images, which are then converted to base64 and sent to the Mistral API for descriptive analysis. This enhances accessibility by providing visual descriptions that aid users reliant on screen readers, already in this little prototype simply producing a markdown file and converting to PDF, with image descriptors in place of the images.

```
# Send images to Mistral API for description
descriptions = query_pixtral(f"Describe these images:", encoded_images)
```

The output includes text descriptions which are promising in initial tests: these could provide an alternative text for the images.

2.4 Demonstration: Text-to-Speech and Audio Generation

For demontration purposes, though out of scope for this practical work and the upcoming thesis, PDFstral also converts extracted text into audio using the gTTS library, enabling screen-reader functionality at a basic, undeveloped level. The concatenated text content of the PDF is processed

and saved as an audio file in MP3 format for easy playback. The file converted in this way is actually the markdown-based conversion product, and is enhanced in an accessibility-minded way even more by the usage of image-to-text conversion.

```
# Convert the concatenated text to speech and save as MP3
full_audio_file = generate_audio(all_text, "full_document")
```

This audio file, enriched with extracted descriptions of images, is accessible to users via a download button (Figure 3).

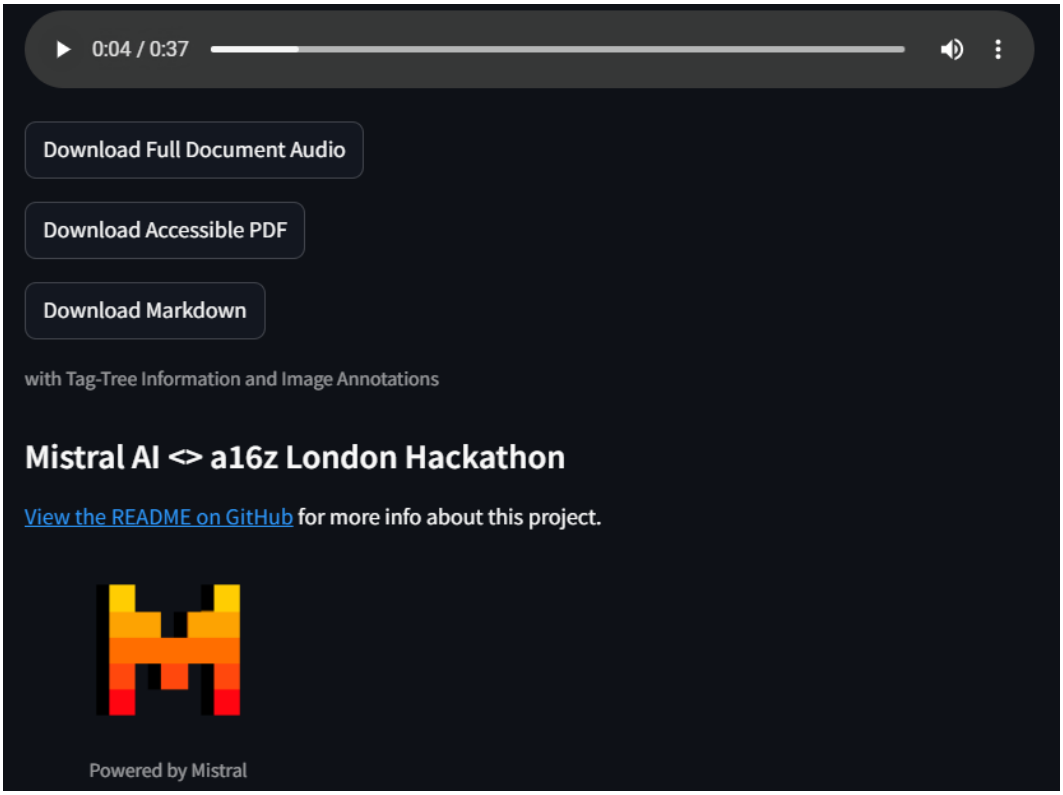

Figure 3: The web app provides an accessibility-oriented markdown-PDF-conversion, and the markdown itself, in addition to an audio transcription, to demonstrate conversion capabilities.

2.5 Streamlining LLM-API Interactions

PDFstral's modular workflow illustrates how GenAI models can enhance document accessibility. This workflow separates ECM concerns and API-based LLM processing, leveraging Python functions to create a seamless interface. Helper functions like `extract_code_between_triple_backticks` streamline text-based responses from the LLM, ensuring clean and actionable data extraction.

The combined use of in-context learning and multimodal API-based interactions illustrates the potential of using Generative AI for automated document processing, creating a scalable, barrier-free approach to content management. The self-imposed limitation to Python packages, rather than enterprise software engineering languages like Java or finished for-purchase modules, demonstrates that using high level programming languages and open source packages is feasible, even for interacting with a complex format like PDF.

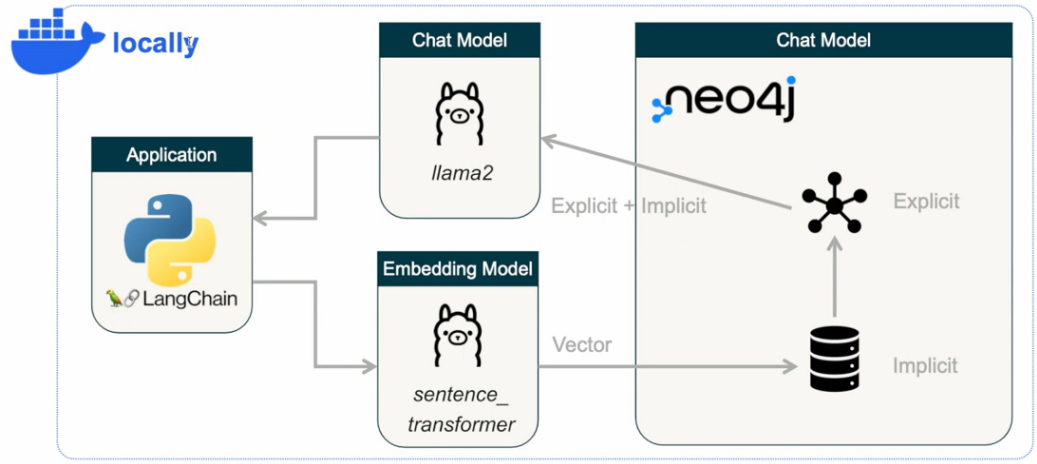

Figure 4: The Gen-AI Stack, from [9] and [15]

3 Setting Up the Document Processing Environment

Switching to the Docker GenAI and Alfresco context, it is quite possible to carry over the separation of ECM and GenAI concerns as well, the REST API acting as the single bridge for consuming and requesting, providing the means of AI enrichment to the Alfresco part of the system: see Figure 6.

The technically more sophisticated add-on will be the neo4j-based RAG (Retrieval-Augmmented Generation), providing a third approach of interest, next to pure in-context- and fine-tuning-based inference and learning, addressed in the outlook in Section 5.

The Alfresco system architecture follows the diagram in in Figure 5.

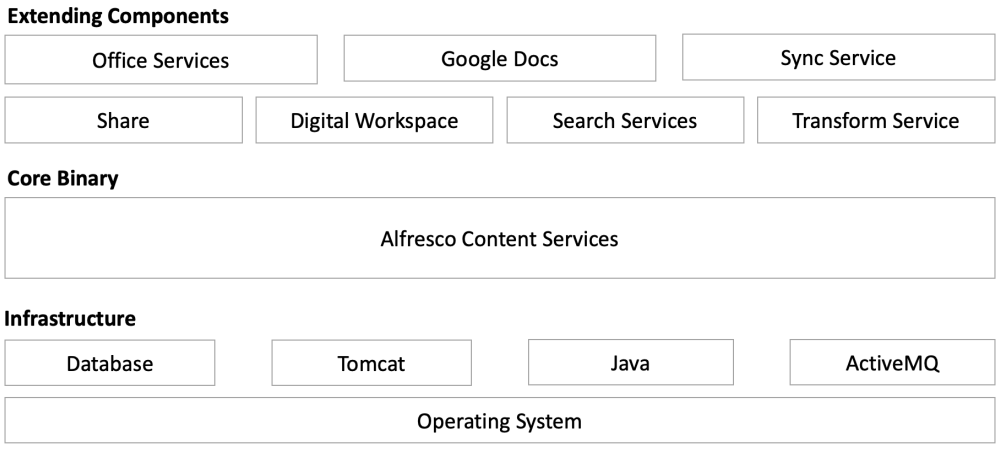

Figure 5: Key components of a typical Content Services (ACS) installation [5] - this project uses the share-component as the central UI, based on ACS.

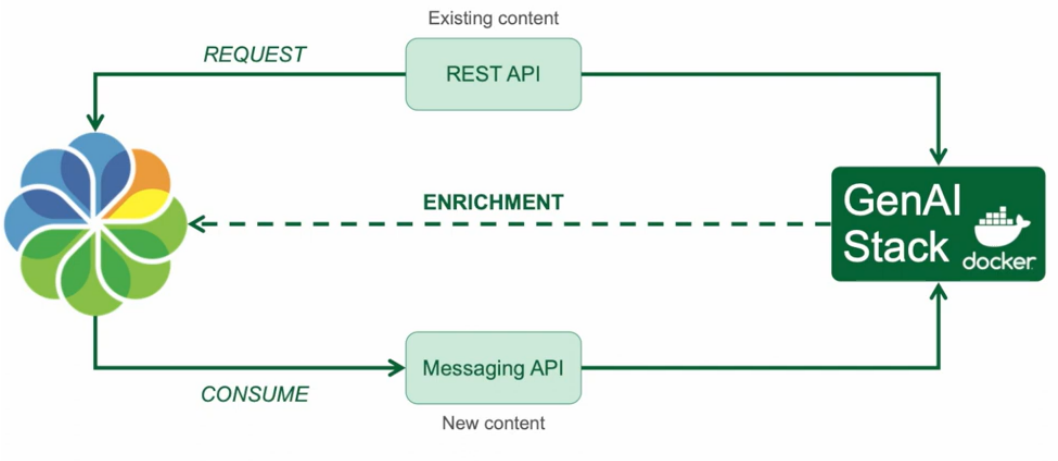

Figure 6: Alfresco Integration of the Gen-AI Stack, from [9] and [15]: "Alfresco provides two main APIs for integration purposes: the Alfresco REST API and the Alfresco Messaging API." [15]

Repository Information

As with the repository for PDFstral, all code developed for this practical work is available on GitHub public repositories. You can find the relevant repositories here:

- PDFstral on GitHub/ PDFstral on Devpost, confirming the hackathon team-project execution using the project idea of the Practical Work topic at hand.
- Main Work: Hoffman on GitHub, developed by the author and exclusively for this report - uses Docker GenAI stack and implements Alfresco integration. Developed as standalone copy (not GitHub fork) from Docker GenAI Stack on GitHub

3.1 A Document-Centric Framework

The following integration and demo tests will exercise some important parts of the new (for the purposes of this report) Docker stack code base: implementing a modern ECM, the setup being established for experimenting here can be considered document-centric, as it puts documents front-and-center.

Knowledge required for setup can be found online and is often provided by companies: for example, in a tutorial, Docker Developer Relations specialist Ajeet Singh Raina explains, "GenAI [the Docker GenAI stack] was announced for the first time during the DockerCon [**?**] and came about through the collaboration of Docker [2], LangChain [21], Neo4j [7], and Ollama [14] [... and] it is basically a set of Docker containers that are orchestrated by the Docker Compose [] tool." [4] The difference from publicly available tools: "With the GenAI Stack you have control over your data, you can run it locally." [4] This, together with the open source aspect, makes this particular stack attractive for industry, as in the present industry collaboration with FAW GmbH: GenAI Stack is an open source stack hosted on GitHub. Moreover, this stack is licensed under the Creative Commons Legal Code CC0 1.0 Universal (CC0 1.0), which allows authors to waive all ownership rights, contributing their work to the "Commons" for unrestricted public use. [6] The Alfresco adaption of the stack is licensed under GNU Lesser General Public License (LGPL) Version 3, which specifically applies to libraries, allowing them to be linked with open-source and proprietary software without requiring the proprietary software to be distributed under the same license.

3.2 Self-Hosted LLM Methods in the Docker/Alfresco/Java-Stack: Out-of-the-Box Functionality of Current Solutions, and Use Cases

The Docker GenAI stack facilitates a streamlined, declarative setup for managing the required machine learning models and their dependencies using Docker Compose. By structuring the service

definitions in a `compose.yaml` file, an organized and reproducible development environment is enabled:

```
services:

  llm:
    image: ollama/ollama:latest
    profiles: ["linux"]

  pull-model:
    image: genai-stack/pull-model:latest
    build:
      dockerfile: pull_model.Dockerfile
    environment:
      - OLLAMA_BASE_URL=${OLLAMA_BASE_URL-http://host.docker.internal:11434}
      - LLM=${LLM-llama2}

  pull-vision-model:
    image: genai-stack/pull-model:latest
    build:
      dockerfile: pull_model.Dockerfile
    environment:
      - OLLAMA_BASE_URL=${OLLAMA_BASE_URL-http://host.docker.internal:11434}
      - LLM=${LLM_VISION-llava}
```

The `llm` service pulls in the latest version of the Ollama language model container and activates under the `linux` profile, indicating it's tailored for Linux environments. It is the core LLM service, which forms the backbone of NLP capabilities within the stack, and is supplemented with services to pull down specific models as per requirements dynamically set with environment variables, in this case llama2 from Meta [23] and llava (Microsoft) [17] for vision/multimodal in this stack. The GenAI code is all contained within `hoffman/gen-ai`.

On the Alfresco side of things (`hoffman/alfresco` and `hoffman/alfresco-ai`), the so-called AI Applier (`hoffman/alfresco-ai/alfresco-ai-applies`) is a Spring Boot [25] application to do command line calls on documents managed through the Alfresco web UI, see Figure 7 and 8 - the out-of-the-box example is document summarizing.

Everything together is up and running with the simple terminal command `docker compose up` in the root project folder (the Docker daemon, the service responsible for orchestrating Docker container lifecycle management [3], is required to be running).

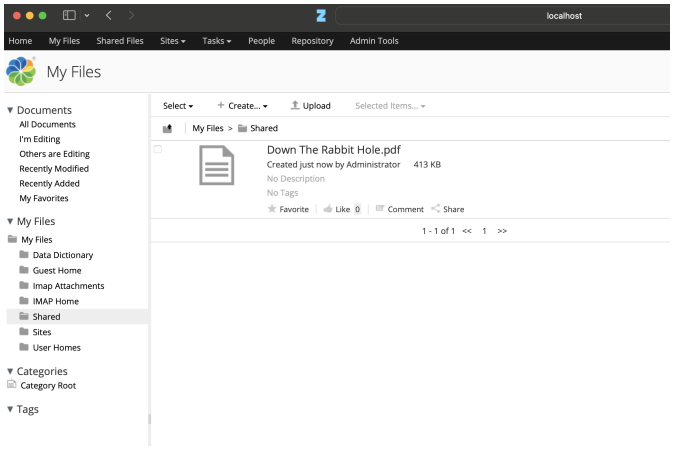

Figure 7: A document managed through the Alfresco UI: (1) Document Upload/Mangagement View Before Enrichment

The advantage of a modern ECM solution is that background concepts like metadata, folder structure and other connections can be relied on to, say, perform an action [5] to create a summary for every document in a certain folder, and write it into the relevant metadata field on the document.

The Java code to execute the action:

```
java -jar target/alfresco-ai-applier-0.8.0.jar \
  --applier.root.folder=/app:company_home/app:shared \
  --applier.action=DESCRIPTION
```

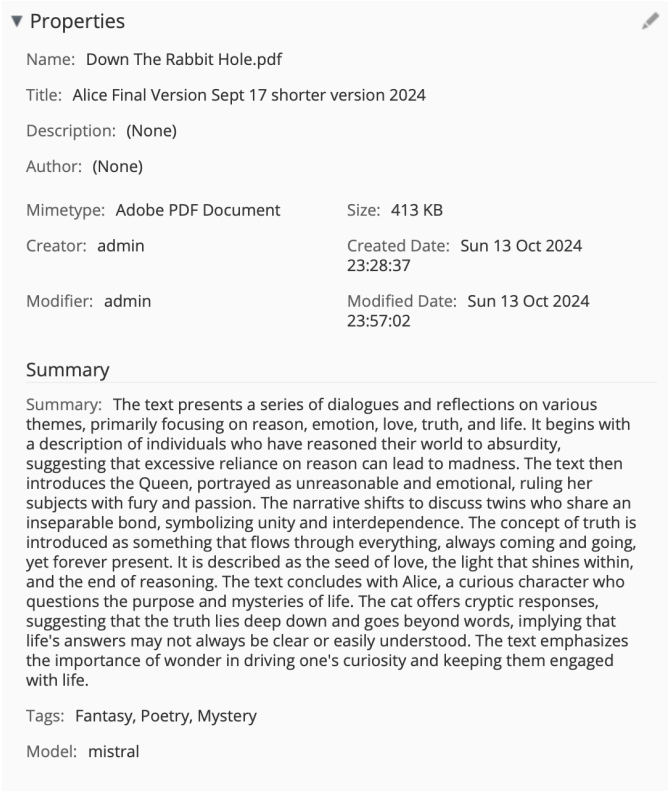

Figure 8: A document managed through the Alfresco UI: (2) Document Detail View After Enrichment (Summary Added Programmatically)

The relevant Java code showing exactly the point where PDF is passed to LLM API and metadata loaded to the ECM (in more context online in the repo):

```
    public Summary getSummary(File pdfFile) throws IOException {

        RequestBody requestBody = new MultipartBody
                .Builder()
                .setType(MultipartBody.FORM)
                .addFormDataPart("file", pdfFile.getName(), RequestBody.create(pdfFile,
                    MediaType.parse("application/pdf")))
                .build();

        Request request = new Request
                .Builder()
                .url(genaiUrl + "/summary")
                .post(requestBody)
                .build();

        String response = client.newCall(request).execute().body().string();
```

```
        Map<String, Object> aiResponse = JSON_PARSER.parseMap(response);
        return new Summary()
                .summary(aiResponse.get("summary").toString().trim())
                .tags(Arrays.asList(aiResponse.get("tags").toString().split(",", -1)))
                .model(aiResponse.get("model").toString());
    }
```

It is this code piece that will be adapted and expanded to analyse the PDF in terms of accessibility features and perform new LLM API requests in the following section, demonstrating the direction for the code leading up to qualitative results that pass accessibility checks: this use case represents **Use Case 1 (Existing Content)** as described in the Readme in the repository - essentially, this use case is about generating content but not placing it back in the ECM, just printing to command-line.

Use Case 2 (New Content) deals with applying Alfresco aspects, a feature that allows the content manager to extend and customize document and folder metadata beyond the standard properties with modular attributes or behaviors and without altering the core data model [5], and using these to take the control over the GenAI functionality and offer it in the UI to the user (rather than command-line, as in the previous example).

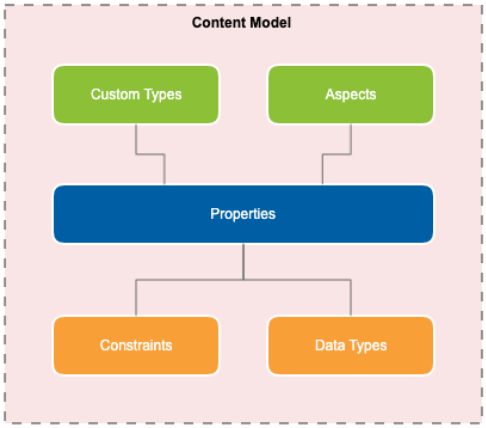

Figure 9: Alfresco aspects in the context of the content model [5]

Functionality out of the box covers summary, classification and even prompting (see Readme).

3.2.1 PDF-Analysis and Manipulation: Accessing PDFs via Alfresco and Interfacing with GenAI on the Java-Stack (Alfresco-AI-Applier)

We now want to recreate the PDFstral-functionality (minus the screenreader-end audio-generation) in the Alfresco GenAI Stack (Alfresco collects the documents into the management system, generative AI features are called inside the Java stack, running in the Docker containers) to round out this Practical Work and prepare the ground for extensive testing in this setup (for the thesis): to this end we want to prepare the relevant Java classes to be triggered in a minimal example by command-line (Use Case 1, as per the previous section) before we move on to the use of Alfresco aspects (Use Case 2, triggering via UI) in this work.

This adaption of the so-called "AI-Applier" can be tested by going to the folder `hoffman/alfresco-ai/alfresco-ai-applier` and building and running like so, with the Docker setup triggered with `docker compose up` on the root folder beforehand:

```
mvn clean package
java -jar target/alfresco-ai-applier-0.8.0.jar \
    --applier.root.folder=/app:company_home/app:shared \
    --applier.action=A11Y
```

This will process the document in the cursory way we define for this project, for now, but already has the basic Java code level access to the documents and the LLM calls. Wheres PDFstral represented a

smaller-scale, but green-field, prototype, this second half of the practical work is really a customization of the "AI-Applier" in this industry solution for the Alfresco ecosytem.

The requirements for this GenAI setup are thus (Java and Maven, for compiling using the pom.xml-files, are used outside of Docker):

- Docker 4.25 (with 20 GB of RAM allocated)
- Ollama
- Java 17
- Maven 3.9

3.2.2 Mutlimodality-Test

To start, one more command-line example for calling a model on the sample image of a dog and curling the GenAI module directly, now locally in Ollama:

```
(base) jack@Jacks-Mac-Studio samples % curl --location 'http://localhost:8506/describe'
    --form 'image=@"shabu_on_a_beach.jpg"'
{"description":" The image shows a cute dog on a sandy beach, lying down and appearing relaxed.
    The dog has a distinctive tan coat with darker markings around the eyes and ears,
    and it's wearing a collar with a small tag. In the background, there is a clear sky and
    part of a body of water that could be an ocean or sea. ","model":"llava"}%
```

This performance appears similar to Pixtral's at first glance, on images, and can be extended with textual context as well, for true multi-modality, a feature likely of interest in the thesis project that will follow this Practical Report.

The LLM default is Ollama, parameterized with the Python package langchain:

```
    ChatOllama(
            temperature=0,
            base_url=config["ollama_base_url"],
            model=llm_name,
            streaming=True,
            # seed=2,
            top_k=10,  # A higher value (100) will give more diverse answers,
            while a lower value (10) will be more conservative.
            top_p=0.3,  # Higher value (0.95) will lead to more diverse text,
            while a lower value (0.5) will generate more focused text.
            num_ctx=3072,  # Sets the size of the context window
            used to generate the next token.
        )
```

4 Preparing a Dataset and First Tests

With a view towards a thesis focused on evaluation and more complex and complete document datasets, the following workflow exercises a mini-dataset used for development in both the PDFstral and the Hoffman projects to demonstrate how this setup helps to (1) collect training documents, (2) build a solid software framework foundation for advanced techniques like RAG, and (3), provide a simple UI for handling document transformations at various scales and under different hyper-parameters.

4.1 A First Workflow

So far this report has outlined two paths, new-document-generation and document-enrichment, as well as two use cases, command-line-call and working with aspects: now a combination is selected, namely **new-document-generation (path) and working with aspects to create new content (use case)**, to demonstrate how an evaluation system might work: hyper-parameters are in principle all the

settings that can be applied through environment variables like the ones found in `genai-stack/.env`, models, model sizes, embedding models and the like.

The concrete Alfresco actions [5], as reusable units of work, implemented in Alfresco for the purpose of this practical work is:

- `accessibility-checkable`: In theory extensible to non-PDF formats: focusing on PDF for now, this action/aspect is for making a non-AI/LLM API call to the Adobe API offering an accessibility checking endpoint.
- `accessible-PDF-makable`: Specific to PDF. For PDFs that can be transformed to a uniform, accessible output PDF based on the Docling intermediary format, considered in more detail below. This action makes use of the GenAI-Stack proper.

The names above correspond to the aspect names found in the Alfresco UI: the ECM fitted with these actions should serve as a suitable test framework for various hyper-parameter settings and evaluation metrics applicable to the problem at hand, for example by running multiple clones of the system and comparing by outcome. This will already be possible, at least for the Adobe PDF checking tool as a metric, at the end of this practical work.

This tool was introduced in January 2024 in Beta and while it 'is not meant to be a "complete" accessibility solution, it can go a long way in automating a good chunk of the work required to properly make documents more accessible.' [13]

4.1.1 Development Workflow

This Practical Work outlines a workflow for a testing framework, but making changes to the framework itself also fits into a workflow, for development purposes: it can be useful to document this especially for non-Alfresco-developers.

At the Alfresco enterprise software level, this workflow mainly encompasses compiling via `mvn package` and (manually) copying the output .jar-files from the `hoffman/alfresco-ai/alfresco-ai-model/genai-model-repo/target` or the `hoffman/alfresco-ai/alfresco-ai-model/genai-model-share/target` directories to the `hoffman/alfresco/alfresco/modules/jars` and the `hoffman/alfresco/share/modules/jars` directories respectively. Command-line scripts might be employed for these kinds of repetitive tasks. The command to load the Docker-environments (building containers afresh) is `docker compose up --build --force-recreate`.

A more streamlined, efficient way of dockerizing and serving the project is `docker compose up alfresco-ai-listener --build` after the corresponding `mvn package`, relying on the Out-of-Process Extension project segmentation by containerizing the project individually. The base system operates independently.

Other important file/implementation locations reflect the extension project SDK, where events and REST API reliance play central roles.

- `hoffman/alfresco-ai/alfresco-ai-listener/src/main/resources/application.properties` maps model properties defined in-XML to their Spring Boot runtime property equivalents - ultimately injected in the following classes.
- Implemented Java classes:
  - Event (trigger): modeled after `ContentClassifyCreatedHandler` and `ContentClassifyUpdatedHandler`, these classes connect the configuration properties for runtime with an actual event-trigger, via filtering of the events marked with the relevant accessibility scoring aspects. The classes in question are `ContentA11yCreatedHandler` and `ContentA11yUpdatedHandler`. Both call the relevant service method for manipulating the document node, described below. The event trigger is also the site of the GenAI call, directly to the Python(-framework-provided) API, via another service, the `GenAiClient` (described later). The exact filtering logic is based on aspect-presence (here: `accessibility-checkable`) and no further filtering based on other properties or document format (e.g. PDF) is made at this level. In more detail, `ContentA11yCreatedHandler` is actually only

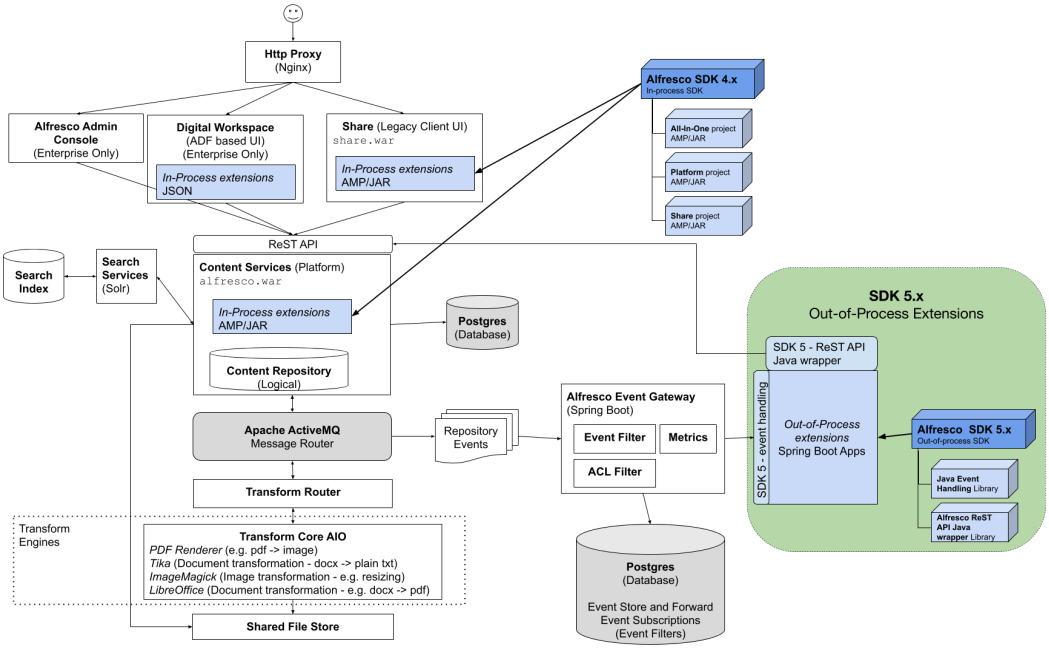

Figure 10: Schematic illustrating where the SDK fits into the big picture: in this conception, the Alfresco AI Listener project is an Out-of-Process Extension project. [19]

called when a document is uploaded to or created in the "shared" (Alfresco directory structure) `Checkable`-folder: a rule [18] on this folder automatically adds the aspect that triggers the event. If the aspect is added to an existing document anywhere inside the Alfresco system, `ContentA11yUpdatedHandler` is called. In both cases the method `handleEvent` contains the relevant business logic. The abstract class `AbstractContentTypeHandler` and the interface `OnNodeUpdatedEventHandler` are the Java Object-Oriented Programming mechanisms offered by the Alfresco system for implementing this sort of functionality.

- Model Class (data transfer): `alfresco-ai/alfresco-ai-listener/.../model/A11yScore` encapsulates the score as a Data Transfer Object (DTO) including metadata like model or API that produced the score.
- Service (`NodeUpdateService`): provides the scoring service to the event in terms of updating the document itself, i.e. a method taking the score-DTO and a UUID for the document and setting the appropriate Alfresco node. This is the method `updateNodeA11yScore` in `alfresco-ai/alfresco-ai-listener/.../service/NodeUpdateService`, destructuring the DTO to a Java-map the Alfresco node component can digest. This service method is called by the event trigger.
- Service (scoring API implementation): In `.../service/PdfProcessingService`. Refers to the PDF Accessibility Checker examples available from Adobe [8], where "the Accessibility Checker API verifies if PDF files meet the machine-verifiable requirements of PDF/UA and WCAG 2.0." The output is a report summarizing the findings of the accessibility checks in JSON format, which is summarized to the most important parts defined by the A11yScore data type (and likely to change over time). The model class already described communicates with this service. The Adobe API provides cloud-based capabilities for PDF manipulation: they are based on credentials specific to this project, the service is a affiliated with Adobe's (Document Cloud) Services Team at the time of writing, and so is a point at which the current work communicates with external services over the internet (relevant for on-premises discussion about the overall solution). Technical details, e.g. class hierarchies, for this library are omitted at this point: the following subsection about neural network training of the scoring capability has more technical detail and a practical comment on the

chosen approach. The project repository contains sample usages provided by Adobe, in `/pdfservices/dc-pdf-services-sdk-java/src` (Java-examples).

– Service (`GenAiClient`): Not added to or configured for this unit of functionality, the Adobe-API-based checker. For the first GenAI-interfacing (still to be done in this practical work), the methods defined here, in turn calling the Python-API, are called at the level of the events typically. As described, particular attention needs to be paid to which document format is filtered and sent to the Python-API: in general, PDF could be pre-selected for already at this level (either at the event filtering or at the `GenAiClient`-passing). Considerations like which logs this information should be captured in, if at all, need to be made.

Cloud Service Credential Information

Important: PDF Services credential information should be exported to the system in line with Adobe's best practices. These credentials are not included in this project repository but are required for API-based scoring.
The following credentials are necessary (obtained from Adobe PDF Services API):

- `PDF_SERVICES_CLIENT_ID`
- `PDF_SERVICES_CLIENT_SECRET`

On macOS and Linux, these credentials should be exported before running the project:

```
export PDF_SERVICES_CLIENT_ID=<CLIENT ID>
export PDF_SERVICES_CLIENT_SECRET=<CLIENT SECRET>
```

These environment variables are required for running the PDF example code locally. To make them available to the `hoffman` project, they must be set in the `alfresco/.env` (untracked in the repository) file. A template `alfresco/.env-template` is provided:

```
# Docker Image versions
ALFRESCO_CE_TAG=23.1.0
SEARCH_CE_TAG=2.0.8.2
SHARE_TAG=23.1.0
ACA_TAG=4.3.0
POSTGRES_TAG=14.4
TRANSFORM_ENGINE_TAG=5.0.0
ACTIVEMQ_TAG=5.17.1-jre11-rockylinux8

# Server properties
SERVER_NAME=localhost

# PDF Services
PDF_SERVICES_CLIENT_ID=<CLIENT ID>
PDF_SERVICES_CLIENT_SECRET=<CLIENT SECRET>
```

The client ID and secret must be provided; otherwise, a Java runtime error occurs in the `A11yScore` constructor due to missing credentials.

The rule-based applications of aspect serves as structuring option for the following testes, where directories might be used to collect documents by scoring metric and/or model. Alternatively, different scores and model results might be stored in properties and document versions, so for now a standard directory `A11y` in the Alfresco shared files is provisioned for further practical work.

Logging output can be monitored in the container output stream in the application Docker Desktop or a command-line shell.

4.1.2 Add-on question: Scoring as an LLM-Task?

As a practical comment on the chosen cloud-based PDF Accessibility Checker approach and a late addition to the project, using the API-yielded scores to train a network is explored. API-reports look like the included samples in the `output` folder in `pdfservices/dc-pdf-services-sdk-java`.

There is an overview summary: in this case, the checker found issues that may prevent the document from being fully accessible.

- **Needs manual check:** 2
- **Passed manually:** 0
- **Failed manually:** 0
- **Skipped:** 0
- **Passed:** 12
- **Failed:** 18

The overview is based on rules by different categories/levels of analyis: Document-level, page content, forms, alternate text, tables, lists, headings. One such category might include these results:

Rule	Status
Accessibility permission flag	Passed
Image-only PDF	Passed
Tagged PDF	Failed
Logical Reading Order	Needs manual check
Primary language	Failed
Title	Failed
Bookmarks	Passed
Color contrast	Needs manual check

Table 1: Document-Level Accessibility Rules

The important aspect here is that these are different levels of concern, even within-category: for the work of document-transformation, this suggests addressing rules in a failed state, in this example (pertaining to a sample slide deck `accessibility2025-03-14T14-24-08.pdf` in the output-directory), one-by-one and checking again.

In this way, the scoring mechanism and structure introduced here actually suggests the solution approach: that is, dividing and conquering the accessibility transformation tasks, and of course, only addressing those issues which actually need to be addressed.

The scoring DTO is presented in the UI as boiled down to the summary description.

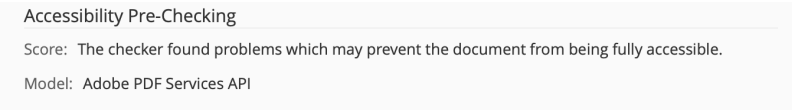

Figure 11: Metainformation fields like these in document properties are configured in the respective model-.xml-file and filled programmatically by the relevant service classes.

The added benefit of working inside a standard ECM solution is the nicety of version histories, such as for accessibility reports in the case of multiple uploads of the same document (as determined by the filename - but these controlling factors, where the file name becomes such for the user, are ultimately configurable), see Figure 12.

Workspace-User-space-model (Discarded)

This concept was ultimately discarded in favor of an immediate property-representation (see following subsection): The actual JSON report is added in a manner that make it both manually retrievable

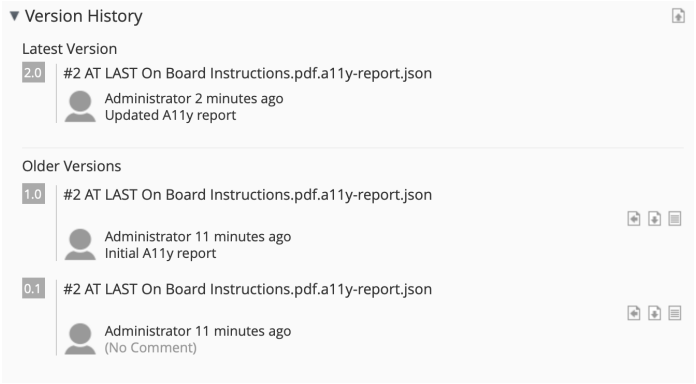

Figure 12: Added niceties of working inside an ECM solution: the in-built versioning will later be used for tracking and managing LLM transformation responses.

and following a logical separation in a workspace for "behind the scenes" auxilliary documents like (intermittent) checker reports. The standard Alfresco location "My Files" (admin user) is selected for this, where documents uploaded and exposed to the user, in a kind of user space (or interface), are kept separately in "Shared Files". The convention established at this point: per file, as determined by the filename, one auxiliary directory is created for the relevant workspace, in the "My Files" and the "A11y" subdirectory, as pertains to this thesis work. This subdirectory might be user UUID or handle in the future. The ECM functionalities nodes, directory-nodes, aspects and node-properties are utilized to initiate a pipeline of sorts: the final thing to do is to create a link between the original document in user space to the workspace report, before going on to triggering the GenAI-part of the pipeline, which will be directly fed from the workspace and should ultimately produce an output for user space.

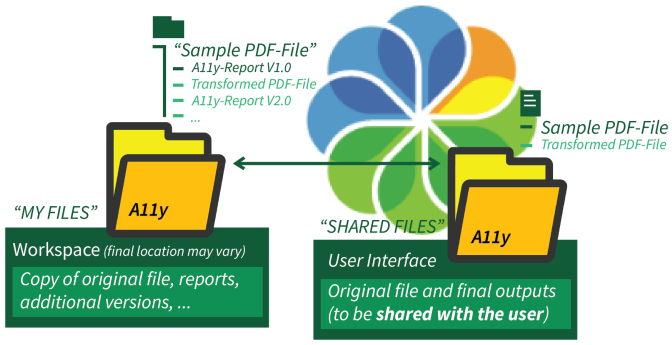

Figure 13: The logical division introduced in this mode, requiring practical user navigation between "My Files" and "Shared Files" in the Alfresco (Share) UI.

Immediate Property Model

The problem with the previous model (from a usability perspective) is the division into a workspace, requiring additional user knowledge and navigation, along with a management overhead of having to tend to two locations instead of one in the case of consistency-relevant events like deletion or name changes. The idea was parked in a branch `workspace-user-space-model` and development continued on the main branch, implementing the property-based model: the defining feature is no separate report-document, instead storage of checking-results in properties immediately visible on the document.

A tight loop continuing from report results-entry in the properties ("naturally" triggering the update events described earlier) is envisionable, providing the mechanism for iterative improvement, once this project moves to transformation-stage: preparing for this, a count property field is added per report field reflected in the properties. The idea is simple: if, going from an existing failing check,

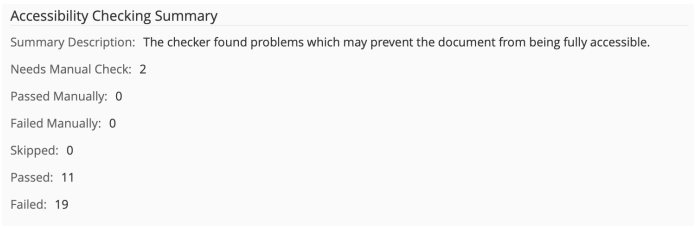

Figure 14: The "accessibility checking" summary section is followed by document, page content, forms, alternate text, tables, lists and headings related rules that are either passing or failing. The failing rules would be natural starting point for document transformation tests.

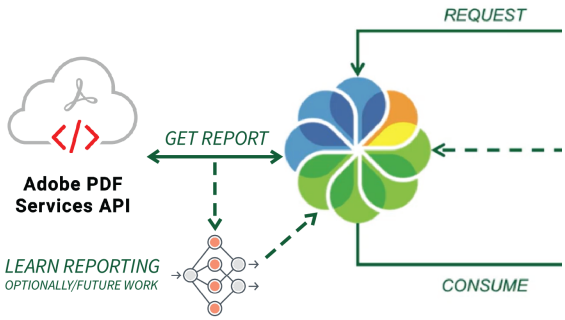

Figure 15: *LEARNing REPORTING* is currently not a goal of this practical work (or thesis) but represents a possible add-on feature with the added benefit of removing cloud-dependence of the present solution.

the report representation or a respective prompt is sent to GenAI, we will probably want to keep track of "attempts" made to improve the document, maybe moving on to other failing tests to address with LLM PDF source code manipulation in the meantime, before returning to tried and potentially "hard" tasks for LLM to solve. The details of such a process remain to be covered at the testing stage, but a useful result of this practical work is to check fragmenting the task of making documents accessible, along with the fragmenting of the document itself, potentially, in order for it to remain manageable. Restricting the use case to small PDFs seems like useful initial step.

In any case the property-based report keeps track of the passing results for the most recent document version, so the current model thinking, and is the local platform for calls to the GenAI-stack (containing the relevant metadata).

The Add-on Question

Of particular interest to this project are the rules that require manual checking: can these be solved by LLM, is the relevant question. An exploration of completely LLM-based scoring might be interesting here (but is not required as a focus of the overall thesis work).

- Logical Reading Order - Ensure the document provides a logical reading order.
- Color Contrast - Verify that text and background have sufficient contrast.

Initial tests including image analysis with Pixtral in the introductory part of this report suggest this might be the case: this scoring contribution of LLMs is posited as a worthwhile effort of rounding out the thesis, for not just automating transformation, but also checking - potentially allowing for an LLM-driven back-and-forth, which should be tested as a first step in LLM-capability evaluation in this domain.

In the view of dividing and conquering the accessibility transformation tasks, completing scoring with LLM assistance is the pre-extension of the use case that is being evaluated: this can also be done before settling the Path 1 vs 2 question outlined, to which this report now returns.

4.2 Collecting Documents for First Mini-Dataset and the PDF-Checking Mode

A small-scale PDF-testset was provided by Institut Integriert Studieren and will be sent transformed to Docling Document/JSON format using the Alfresco action, using the `accessible-PDF-makable` aspect - down-stream event trigger and services follow the `accessibility-checkable` example detailed by implementation locations above, with the notable exception that the result is not a node property on the document but a separate document version.

4.2.1 Document Transformation: Talking to the GenAI Stack via Document Loading

So far, with the Adobe API, we have not talked to the GenAI Stack at all yet: this changes in this part, where transmitting and chunking a document (including scoring information) is demonstrated. The starting off point is the moment checker-results are added as properties to an existing document.

The starting-off point is configured to be the `GenAIClient` call in `ContentA11yCreatedHandler` (to `#getAccessibleDocumentVersion`). The switch of domain (with the described REST-connection in between) is logged. The important thing is here: in the Docker Desktop console, where logging is most easily retrieved, the GenAI logs need to be checked under the `gen-ai` container, not anymore the `alfresco-ai-listener` one, that is, until any result is passed back to Alfresco.

In the future manual sending actions in the UI may be usefully implemented, especially when it comes to a comparison of prompting methods.

For now the signature of the involved Python FastAPI endpoint reads:

```
@app.post("/accessible-document-version")
async def new_version(
    file: UploadFile = File(...),
    metadata: str = Form(...)
):
    """
    Accepts a PDF file and a JSON metadata report (e.g. accessibility score).
    """
    # ...
```

And the main element is here (potentially modified later in what exactly is sent to the LLM):

```
doc = fitz.open(stream=file_bytes, filetype="pdf")
extracted_text = "\n".join([page.get_text() for page in doc])
doc.close()

print("Extracted text length:", len(extracted_text))

prompt = (
    "You are an assistant that improves document accessibility.\n\n"
    "Accessibility metadata:\n"
    f"{json.dumps(a11y_data, indent=2)}\n\n"
    "Document content:\n"
    f"{extracted_text[:5000]}...\n\n"
    "Based on the content and metadata, suggest how to improve accessibility."
)

print("Prompt sent to LLM:\n", prompt)

response = requests.post(
    f"{ollama_base_url}/api/generate",
    json={
        "model": llm_name,
        "prompt": prompt,
        "stream": False
```

```
    }
)

print("LLM HTTP response status:", response.status_code)
print("LLM response body:", response.text)
```

The corresponding log output, to illustrate what is going on under the hood (without structural information, which is less readable):

```
2025-05-08 13:11:28 ...
2025-05-08 13:11:28
2025-05-08 13:11:28 Based on the content and metadata, suggest how to improve accessibility.
2025-05-08 13:11:28 LLM HTTP response status: 200
2025-05-08 13:11:28 LLM response body: {"model":"mistral",
    "created_at":"2025-05-08T11:11:28.266484Z",
    "response":" Based on the provided document content and accessibility metadata,
    here are some suggestions for improving document accessibility:\n\n1. **Title**: ...
```

(Prompt is cut off for readability in the above.) After one iteration the log output can look something like:

```
2025-05-20 10:44:00         (Seite 2) Tj
2025-05-20 10:44:00         ET
2025-05-20 10:44:00         ET
2025-05-20 10:44:00         endstream
2025-05-20 10:44:00         endobj
2025-05-20 10:44:00     trailer
2025-05-20 10:44:00     << /Root 1 0 R /Encrypt 0 R >> endobj
2025-05-20 10:44:00     startxref
2025-05-20 10:44:00     0 65536
2025-05-20 10:44:00     %%EOF
2025-05-20 10:44:00 2025-05-20 08:44:00.535 WARNING document: LLM response is not valid
    PDF code, saving anyway as .pdf text file.
2025-05-20 10:44:00 2025-05-20 08:44:00.535 INFO    document: Sending prompt to LLM
    (attempt 2/2)...
```

The attempts (looping behavior) will be described in more detail. The LLM response is still abstract response and needs to be processed, say to extract just the relevant PDF object code, or to remove unhelpful markdown syntax and similar content.

A good demonstration at this point would be the Path 1 discussed earlier, producing a standard-format at both intermediary and final stages, rather than Path 2, where the original document is enriched - this is the more complicated, but also more desirable task, which follows from the original conception of PDF as a typesetting format. Typesetting (appearance) matters.

At this point, **Path 2 is tested**, as a basic call to the LLM level, with a minimal document loading [20] approach, that is limited conversion to another format. For this purpose the `UnstructuredPDFLoader` [22] was investigated. Output for sample PDF files is found in the repository `notebooks` directory, but essentially, does not include structural object code, the kind we will want to manipulate.

The package `PyMuPDF` on the other hand retains structural information. Sample output is found in `/notebooks/object_log-extraction-pymupdf.ipynb`. (The readme for the folder covers environment setup details in case code tests are desired.)

The focus is naturally low-level, that is PDF object code level, so at the level of the underlying data structure and data types that a PDF file uses to represent its content. This is reflected in the final working version of the endpoint.

4.2.2 The First Workflow Revisited

This main sample set is located in `samples/prototype-for-open-innovation` in the repository and is uploaded to the `A11y` directory for "Pre-Checking", that is, initial scoring: this is the chosen

initial specification for the aspect `accessibility-checkable`, auto-applied it to any PDF documents dropped in this folder in the ECM system, triggering an accessibility scoring process using the standard Adobe checker tool, via an API.

Implementation details and the (standard) checker tool were described in the implementation locations list. For triggering LLM-transformations a manual aspect mechanism is decided: different aspects for different models or strategies are conceivable and allow for a user (UI) driven use of the resulting project. The working directory, so to speak, is the `A11y` directory created previously.

4.3 Path 1 Test: Advancing LangChain PDF-Ingestion Functionality Through Docling, and Adding to the Genai-Stack-API (Alfresco-AI-Listener)

Without advanced PDF-ingestion upstream of the LLM-operationalization this work would be limited to the initial output results of PDFStral. Docling is an MIT-licensed open-source tool from IBM [10] that has been touted as "the missing document processing companion for generative AI" [12]: it integrates well with LangChain and was selected for this reason mainly. The Docling/LangChain functionality that serves as the heart of the "Gen-AI Stack" is also explored in the notebooks-directory in the hoffman submission repository.

In general, the notebooks-directory serves for testing, outsourcing code to `genai-src` once it is ready to check as part of the pipeline in the Alfresco context. That code is then imported as usual during Docker container startup. Rebuild of `genai` is triggered analogously to `ai-listener` via `docker compose up genai --build --force-recreate`

This functionality is now included in the form of an endpoint to the existing document-LLM-manipulation code: The genai-stack-directory includes the central `document.py` using a combination of FastAPI, Streamlit, LangChain, and Neo4j Python packages to provide the endpoints to Alfresco, realizing the enrichment interface. To "jsonify" the input document, a DocumentConverter instance (Docling) is created to process the document passed by POST network call:

```
# Define the DocumentConverter
converter = DocumentConverter()

@app.post("/jsonify")
async def jsonify(file: UploadFile):
    """
    Endpoint to process an uploaded PDF file and output JSON format.
    """
    try:
        # Save the uploaded file temporarily
        temp_file_path = f"temp_{file.filename}"
        with open(temp_file_path, "wb") as temp_file:
            temp_file.write(await file.read())

        # Convert the document using Docling
        result = converter.convert(temp_file_path)

        # Export the result to a JSON-compatible dictionary
        json_output = result.document.export_to_dict()

        # Clean up temporary file
        os.remove(temp_file_path)

        # Return JSON output
        return {"document": json_output, "status": "success"}

    except Exception as e:
        return {"error": str(e), "status": "failure"}
```

Requirements were added to the Docker-pip-installed `requirements.txt` as well, to enable this functionality contained in `document.py` in `genai-stack`, which becomes the main entry point for extending API endpoints.

While LLM calls and chaining are not demonstrated in this example, this would be the successive prompt-based processing of text: the LangChain package derives its name from this method. There are other endpoints that demonstrate this functionality in the forked repository. The genai-stack's readme provides further overview over environment settings, which would also take Ollama LLM values for example. The `document.Dockerfile` works together with `document.py` to create a containerized environment for deploying the FastAPI-based application made up of endpoints like the one above, as well as installing dependencies.

This should serve as a brief demo of working with Docling document processing (subsequently LLMs) in this programming setting and as a taster for what might be possible in the domain of generating accessibility features in documents, using the present cutting-edge LLM stack as part of an Alfesco enterprise software architecture.

4.4 Target Docling Document (Loader) Approach (Path 1 Demo)

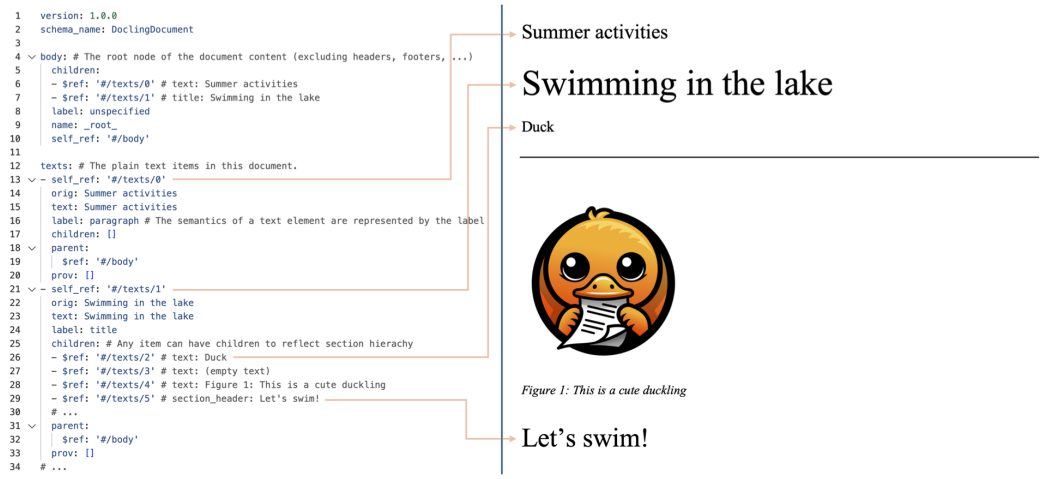

Figure 16: This example shows nesting below a title: taken from [16].

Initial notebooks-tests show that the Docling Document appears to correspond with the JSON export format.

4.5 Demo and Sample Documents for Making Accessible PDF Documents: A First Few-Shot-Learning Test with Re-Checking (Path 2 Demo)

Calling the new API endpoint from the Alfresco front-end requires installing the Hoffman repository and applying the aspect `accessibility-checkable` at the document or folder level - in the latter case the aspect triggering the action is applied to all content inside the given folder.

To close this chapter and the practical work, an evaluation of the sample documents with the presented minimal document loading approach (PyMuPDF, Path 2) in terms of the Adobe PDF-checker API (behind the `accessibility-checkable` aspect) follows: that is, a representative one-step loop is constructed, to re-check the documents after they have been transformed (one iteration). This loop is limited for now, but demonstrates a working principle, of iterative, checker-report-based transformations. A note on how this outer loop is constructed follows after a description of a second, inner loop.

This second loop is also present, as diagrammed in Figure 17: this is the multi-try LLM response check for valid PDF, since it only makes sense to pass back a valid PDF for accessibility checking in the first place. Looping behavior is made configurable via environment variables (see box). The inner loop is triggered by a simple validity check based on first and last line in the source code for now.

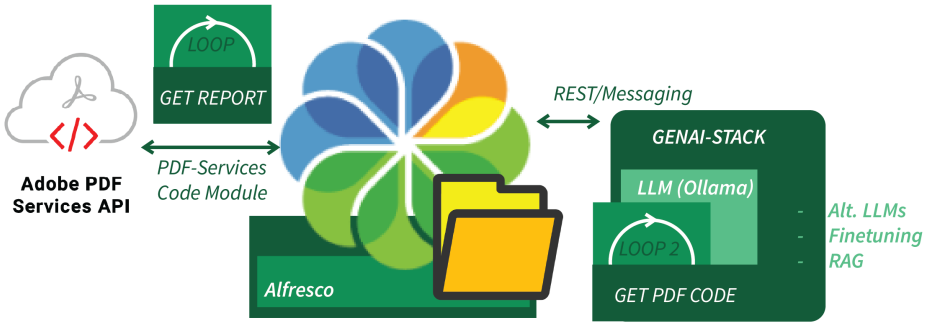

Figure 17: Two loops are part of the solution design: one at checking, one at LLM interfacing (prompting and checking the syntax of the response)

A note on the outer loop: This is realized in the interaction between `ContentA11yCreatedHandler` and `ContentA11yUpdatedHandler`: `ContentA11yCreatedHandler` triggers a non-conditional LLM call via the `genAiClient` and writes back a new version of the document to the node, as a non-major version. `ContentA11yUpdatedHandler` re-evaluates (accessibility-checks) this version and writes current accessibility metadata to the properties.

This part is not implemented as of submission but would be very similar to existing logic: Now a conditional `genAiClient` call happens (tightened or relaxed to a sensible specification in the future). The passing final version is saved as a new major version and is not sent to the `genAiClient` for LLM processing anymore.

Looping Environment Variables

Info: These environment variables configure default retry behavior in the GenAI PDF accessibility pipeline. They can be overridden at runtime by parameters in the API request.

- `MAX_RETRIES_LLM_LOOP_DEFAULT=2`
  - **Location:** `genai-stack/.env`
  - **Note:** This is the default value. It can be overridden via the `max_retries_param` query parameter or the `maxRetries` field inside the `metadata` JSON payload.
- `MAX_RETRIES_ACCESSIBILITY_CHECKING_LOOP=2`
  - **Location:** `alfresco-ai/alfresco-ai-listener/src/main/resources/application.properties`

These values ensure retry logic is configurable across both the GenAI service and the Alfresco integration layer. If not specified, a safe default of 2 attempts is used.

Sample documents are saved in `/samples/prototype-transformations-at-end-of-practical-work`. These are documents with their respective counterparts that have passed through both the described loops at least once. PDF-validity of these documents is not a given at this point in the development of the project.

5 Summary and Discussion

This project was focused on preparing a setup to accomplish the tasks described and establish a manageable workflow, in addition to demonstrating the availability of libraries in the LangChain ecosystem to destructure PDFs sufficiently to make LLM calls for accessibility feature enrichment. This author started from a prototype developed over 48 hours in the context of a hackathon and moved on to a setup proposed by industry partner FAW GmbH, demonstrating this setup's capacity to perform the tasks, by utilizing newly available PDF-centered tools like Docling. The concrete output was an Alfresco module with an API call to the LangChain document containing these tool calls, in the so-called GenAI stack. At the document level, aspects (an Alfresco concept) were employed to

facilitate additional Java/ECM level operations like PDF-checking (accessibility checking). At the same time the repo contains endpoints utilizing LLM functionality: for the thesis work, a combination approach, of Java and Python level tooling and LLM calls, will be needed.

The main proofs of concepts reached at the close of this practical work were (1) file generation from LLM output (LLM response to PDF) and (2) the iterative PDF-check after LLM output, significant for demonstrating a viable, stepwise improving approach.

In the course of this exploration a crucial design question was answered, namely, at what level of the stack the PDF-destructuring and LLM-interfacing should happen. Even in a Java/Alfresco context, it appears that new and capable tools are appearing in the Python/LangChain ecosystem, allowing for an easy integration at the Python level. If the need for Java-based low-level PDF manipulation arises, this choice might have to be revisited in the future.

A practical upshot is that the Python layer can be seen somewhat decoupled in this view, focusing on notebooks for implementation tests at the thesis stage, for example. If implemented to an API, this API can be addressed from different solutions beyond Alfresco as well. There is no Alfresco lock-in in this sense.

In addition, by understanding the Java level, inclusive of the checker tooling available and ways to handle meta-information via Alfresco properties, there is some understanding of how this platform can make what information available when, for calls to the GenAI stack. Thereby tests inside Python notebooks can be more directed, if not directly drawing on the documents already inside a running instance of the Alfresco system, before implementing to this platform - where a long-term goal of bringing viable LLM-processing to such a platform is considered worthwhile, for ease of use and provisioning to an end-user.

5.1 Limitations

There was no actual qualitative testing of document transformation results but for basic PDF-checking: Evaluation will be the focus of the subsequent Masters Thesis, instead. One idea is to train a neural network side by side to a formal checker as part of building out this platform. Purely PDF-checking approaches might lack qualitative nuance.

Vector Databases and RAG were also not touched, although GenAI enables this tooling in the form of Neo4J. This preempts the current trend of using RAG in addition to the methods we set out to compare, fine-tuning and in-context - all addressing different steps in the training pipeline and not necessarily excluding each other: fine-tuning at the LLM training level, in-context at the prompt/LLM calling level, and RAG at the retrieval one. Once again, these modalities and improvement potentials exist quite independently from the ECM side, inside the GenAI stack.

5.2 Technical Notes and Observations

Pertaining to the finetuning subject, these notes should be added:

- The LLM must be trained or prompted carefully to return correct, complete PDF syntax. This is not guaranteed with general-purpose LLMs unless the task is constrained and validated. Finetuning might be able to help.
- Doing PDF syntax validation using a library like PyPDF2 or fitz after creation to ensure it can be parsed seems necessary and may offer room for improvement in concept.
- If this becomes brittle, switching to generating instructions or a structured JSON, then using a Python library to implement Path 1 might be the more desirable route after all.

For scoring, the Adobe PDF Services API and specifically the PDF Accessibility Checker was used: this is a modern cloud solution and limits the on-premises use of this project. As a Machine Learning project, an add-on contribution of storing scores yielded by the current implementation for use for model training was posited: future, thesis-level A-B-testing-like switching of model and API scoring is conceivable and should be explored.

5.3 Next Steps and Conclusion

The directly next step will be to research larger data sets helpful for fine-tuning and RAG and collect them into this Alfresco setup, in addition to further tests related to breaking down the prompting as pertains to PDF source-code and the PDF itself (dealing with large PDF). The goal of establishing a viable evaluation framework was reached, in coordination with and supported by FAW GmbH and Hyland.

References

[1] Alfresco Docs - Content Model Extension Point.

[2] Docker: Accelerated Container Application Development, May 2022.

[3] What Is Docker Daemon ?, February 2024. Section: Docker.

[4] Ajeet Singh Raina – Docker Captain, February 2025.

[5] Alfresco Docs - Install overview, February 2025.

[6] Legal Code - CC0 1.0 Universal - Creative Commons, March 2025.

[7] Neo4j Graph Database & Analytics – The Leader in Graph Databases, February 2025.

[8] Adobe. PDF Accessibility Checker | How Tos | PDF Services API | Adobe PDF Services, March 2025.

[9] Angel Borroy. Integrating Alfresco with Docker GenAI Stack, November 2023.

[10] Christoph Auer, Maksym Lysak, Ahmed Nassar, Michele Dolfi, Nikolaos Livathinos, Panos Vagenas, Cesar Berrospi Ramis, Matteo Omenetti, Fabian Lindlbauer, Kasper Dinkla, Lokesh Mishra, Yusik Kim, Shubham Gupta, Rafael Teixeira de Lima, Valery Weber, Lucas Morin, Ingmar Meijer, Viktor Kuropiatnyk, and Peter W. J. Staar. Docling Technical Report, December 2024. arXiv:2408.09869 [cs].

[11] James Barlow. pikepdf Documentation — pikepdf 9.5.2 documentation, March 2025.

[12] William Caban. Docling: The missing document processing companion for generative AI, November 2024.

[13] Raymond Camden. Introducing the Adobe PDF Accessibility Checker API Beta, January 2024.

[14] Michael Chiang. Ollama, March 2025.

[15] Collabnix, Docker and DevOps. Getting Started with Docker GenAI Stack, December 2023.

[16] Docling. Docling Document - Docling, March 2025.

[17] Liu Haotian. LLaVA, December 2023.

[18] Hyland. Alfresco Docs - Alfresco SDK 6.x for out-of-process extensions, March 2025.

[19] Hyland. Alfresco Software | Hyland, March 2025.

[20] LangChain. How to load PDFs | LangChain, May 2025.

[21] LangChain. LangChain, March 2025.

[22] LangChain. UnstructuredPDFLoader — LangChain documentation, May 2025.

[23] Meta. Meta Llama 2, March 2025.

[24] PyMuPDF. PyMuPDF 1.25.4 documentation, March 2025.

[25] Tanzu VMware. Spring Boot, March 2025.