ChatGPT

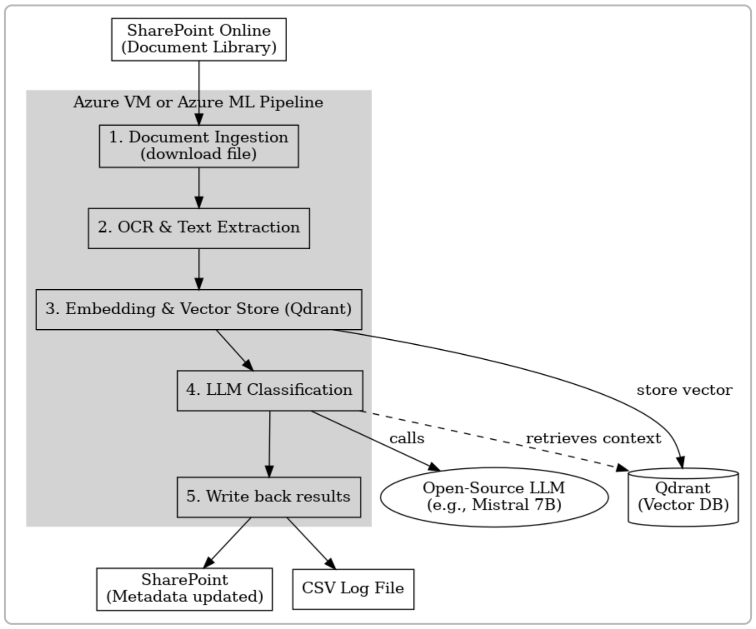
# Open-Source RAG Document Classification System on Azure

## Project Overview

This project is an open-source Retrieval-Augmented Generation (RAG) system for classifying legal documents stored in SharePoint Online. It combines OCR (optical character recognition), a vector database, and a large language model (LLM) to automatically assign each document two attributes: **Document Type** and **Document Category**. By leveraging RAG, the system can handle varied document formats (Word, PDF, scanned images) and use semantic search to improve classification accuracy. Key features include:

- **Azure Deployment:** Runs on an Azure environment (either a VM or Azure ML workspace) under the user's Azure Sponsorship subscription for easy management.
- **Multi-format Ingestion:** Accepts Word (.docx), PDFs (including scanned PDFs), and image files (PNG/JPG), extracting text via OCR when needed.
- **Vector Database (Qdrant):** Stores document embeddings to enable semantic retrieval of relevant content or examples for the LLM [1].
- **Open-Source LLM:** Uses an open-source large language model (such as **Mistral 7B** or **LLaVA**) to perform zero-shot classification of documents into the desired Type/Category labels.
- **Results Integration:** Writes back the classification results to SharePoint (as document metadata) and logs all outputs to a CSV file for auditing.
- **Future-Proof Design:** Provides guidance to later migrate the solution to the Arandia Law Firm’s own Azure tenant with minimal effort.

## Architecture Overview



*Architecture diagram of the proposed RAG-based document classification system. The pipeline processes each document through ingestion, OCR, embedding, and LLM-based classification* [2] *.*

At a high level, the system follows a pipeline of sequential phases from document ingestion to result output. The architecture (shown above) consists of the following steps:

1. **Document Ingestion:** A new or updated file in the SharePoint document library triggers the pipeline. The file is downloaded from SharePoint into the Azure environment for processing.
2. **OCR & Text Extraction:** The system determines how to extract text from the document:
3. If the file is a Word document or a text-based PDF, the text is directly parsed.
4. If the file is an image or a scanned PDF with no embedded text, an OCR engine (e.g. Microsoft's OCR or TrOCR) is used to read the text from the page images.
5. **Embedding & Vector Storage:** The extracted text is converted into a vector embedding using a sentence transformer or similar model. This embedding is stored in a Qdrant vector database along with metadata (document ID, etc.) [1] . The vector store enables semantic search of document content.
6. **LLM Classification (RAG):** An open-source LLM is prompted with the document's content (or retrieved semantic summaries) to predict the *Document Type* and *Document Category*. The LLM can be provided additional context retrieved from the vector store – for example, descriptions of possible categories or similar documents – to ground its classification (this is the retrieval-augmented step).
7. **Write Back Results:** The resulting classifications are written back to SharePoint as metadata fields on the file (e.g. updating "Document Type" and "Document Category" columns). Additionally, the outcome (file name, type, category, timestamp, etc.) is appended to a CSV log file for record-keeping.

Each of these steps is detailed in the phase-by-phase guide below, including recommended tools and configurations for an Azure-based deployment.

## Phase 1: Azure Environment Setup

**Goal:** Prepare an Azure compute environment to run the pipeline and host necessary services (LLM and vector database). You have two main options for deployment: an **Azure Virtual Machine** or **Azure Machine Learning (Azure ML)**. We recommend choosing the option that best fits your familiarity and operational needs:

- **Azure VM:** A single VM can host the entire pipeline (including the OCR tool, Qdrant server, and the LLM runtime). This is often simpler to set up and debug. Choose a Linux VM for better compatibility with open-source ML tools. Ensure the VM has a GPU if you plan to run a larger LLM locally for faster inference. For example, a VM size such as `Standard_NC6s_v3` (Nvidia V100 16GB GPU) or an `NC4as_T4_v3` (T4 GPU 16GB) is suitable. Models like Mistral 7B (7.3B parameters) can run on ~15–16 GB of GPU memory in FP16 [3], so a 16 GB GPU provides a good balance. If needed, the model can also be quantized to 8-bit or 4-bit to fit on smaller GPUs (or even CPU, at the cost of speed). Ensure the VM has at least 16 GB of RAM and adequate disk space (50+ GB) to store model files and data.
- **Azure ML:** If using Azure Machine Learning, you can create a **Compute Instance or Cluster** with similar specs (GPU-enabled). Azure ML provides a managed environment with easy scaling, experiment tracking, and the possibility to deploy the model as a web service. For ease, you might start with an Azure ML Compute Instance (which acts like a VM with Jupyter interface) for development. Configure the instance with a GPU VM size that meets the model requirements (e.g., a V100 or A100-based SKU for larger models). Using Azure ML's **Prompt Flow** or pipelines is also possible for orchestrating the RAG steps, but it introduces more overhead. If you prefer simplicity, an Azure VM might be the quicker path.

**Setup Steps:**

- **Provision the Compute:** In your Azure subscription (Sponsorship), create the VM or Azure ML compute. For a VM, install the latest Ubuntu LTS and update packages. For Azure ML, set up a Compute Instance with a GPU image (like "AzureML DSVM" which has many ML packages pre-installed).
- **Networking and Access:** Ensure the VM can access SharePoint Online (it will need internet access for SharePoint API calls). For Qdrant, if self-hosted on the VM, you can keep it local or expose a port if needed. Lock down inbound access to the VM (use Azure VPN or private link if security is a concern, since it will handle potentially sensitive legal docs).
- **Install Dependencies:** Set up Python (e.g., via Anaconda or Miniconda) and needed libraries:
- Python libraries: `pypdf` or `PyMuPDF` (for PDF text extraction), `python-docx` (for .docx), Hugging Face Transformers (for LLM and possibly TrOCR), `sentence-transformers` (for embedding models if used), `qdrant-client` (for vector DB connectivity), `pandas` (for CSV logs), etc.
- Install OCR tools: For open-source OCR, you might use **TrOCR** via transformers or **Tesseract**. If using Tesseract, install it on the VM (`sudo apt-get install tesseract-ocr`). If using TrOCR, ensure you have `torch` and the model weights will download on first use. Alternatively, for Azure OCR, you'd need the Azure AI Vision SDK or REST API endpoint (which requires an Azure Cognitive Services resource).
- Install Qdrant: The simplest way is using Docker – e.g., `docker run -p 6333:6333 qdrant/qdrant` will launch a Qdrant service listening on port 6333. You can also use the Qdrant Cloud

(hosted) or run it in-memory via the Python client for testing [4]. Ensure Qdrant is running before you attempt to index or query vectors.

- (Optional) Install PnP PowerShell: If you plan to use PowerShell scripts for SharePoint, install the cross-platform PnP PowerShell module. On a Linux VM, you can install PowerShell 7 and then `Install-Module PnP.PowerShell`. Alternatively, ensure you have the Microsoft Graph API Python SDK if you prefer coding the SharePoint updates in Python.

- **GPU Configuration:** If using a GPU VM, install NVIDIA drivers and CUDA as needed (Azure ML instances typically have this ready). For GPU acceleration in Transformers (for LLM and maybe for TrOCR which uses a vision model), verify that PyTorch can detect the CUDA device.

By the end of Phase 1, you should have an environment ready with all required software, and an operational Qdrant database.

**Recommended VM Specs:** A single GPU VM (V100 or better) with 6+ vCPUs, 56+ GB RAM is a robust choice. This allows the LLM to run in memory and OCR/embedding tasks to run in parallel. Disk space of ~200 GB is suggested if storing many documents and vectors (though vectors themselves are small, any cached model weights and temporary files can accumulate).

## Phase 2: Document Ingestion & OCR Pipeline

**Goal:** Retrieve documents from SharePoint and extract their text content. Because the documents can be in various formats and may contain scanned images, the pipeline needs to detect and apply OCR when necessary.

**2.1 SharePoint Ingestion:** The pipeline can be triggered in multiple ways. For example, you might schedule a periodic job to scan the SharePoint library for new or modified files, or use a SharePoint webhook/Flow to notify the Azure service of changes. In either case, the first step is to **download the file from SharePoint**.

- Using **PnP PowerShell**: One way is to use the PnP PowerShell command `Get-PnPFile` to download the file by URL or ID [5]. In a PowerShell script, you would first connect to the SharePoint site:

```
Connect-PnPOnline -Url "https://<tenant>.sharepoint.com/sites/<SiteName>" -
Interactive
$file = Get-PnPFile -Url "/Shared Documents/LegalDocs/<FileName>" -
AsListItem
```

  This gets the file (and its List Item ID for later metadata update). You can also filter or search for recently added files via PnP or Graph API. If scripting entirely in Python, you can use the Microsoft Graph API to fetch the file content (Graph has an endpoint to download file content given the Drive ID or item path).
- **File Handling:** Save the file to a local working directory (e.g., a temp folder). Note the file name or ID to correlate with SharePoint item for later update. Also capture any metadata if needed (but typically we will write new metadata rather than read existing).

- If the pipeline runs as a long-running service, you might continuously poll or await triggers. In a simple implementation, a script can iterate through all files in a library and classify those not yet classified (or re-classify updated ones).

**2.2 Text Extraction & OCR:** Once the file is downloaded, the system determines how to extract text from it:

- **Word Documents (.docx):** Use the `python-docx` library or the `zipfile` approach (docx is a ZIP of XML) to extract text. `python-docx` can iterate over paragraphs and tables to retrieve text. Another option is to save the .docx as PDF or text via LibreOffice in headless mode, but that is heavier. Since .docx is structured, using a library is straightforward. The output is the full text content (perhaps with some formatting or newline separation).
- **PDFs:** PDFs might have embedded text (selectable text) or might be scanned images. You can attempt to extract text using libraries like **PyMuPDF (fitz)** or **PDFPlumber**. For example:

```python
import fitz
doc = fitz.open("file.pdf")
text = ""
for page in doc:
    text += page.get_text()
```

  This will give text if the PDF has it. If `text` comes back essentially empty or with very few characters, it's likely a scanned PDF.
- **Images/Scanned PDFs:** For documents where no text could be extracted (e.g., a scanned contract in PDF or an image file), invoke the OCR pipeline. There are two main OCR approaches:
- *Open-Source OCR (TrOCR):* **TrOCR** is a Transformer-based OCR model by Microsoft that can handle printed text (and even handwriting, with the appropriate model) [6]. You can use Hugging Face Transformers pipeline for TrOCR. For instance, using `trocr-base-printed` model:

```python
from transformers import TrOCRProcessor, VisionEncoderDecoderModel
from PIL import Image
processor = TrOCRProcessor.from_pretrained('microsoft/trocr-base-printed')
model = VisionEncoderDecoderModel.from_pretrained('microsoft/trocr-base-
printed')
image = Image.open("page1.png").convert("RGB")
pixel_values = processor(images=image, return_tensors="pt").pixel_values
generated_ids = model.generate(pixel_values)
text = processor.batch_decode(generated_ids, skip_special_tokens=True)[0]
```

  You would apply this to each page image of a PDF (you can render PDF pages to images via PyMuPDF or use PDFium). TrOCR will output the recognized text for each page.
    - *Alternative OCR options:* **Tesseract** (OCR engine), or **EasyOCR**, or other models like LayoutLM if layout info is needed. EasyOCR (as used in some tutorials [7]) is another deep learning OCR that supports many languages.
- *Azure Cognitive Service OCR:* Azure's Computer Vision "Read" API is a robust option if you prefer a managed service. It can handle printed and handwritten text in images/PDFs in many languages [8]. You would call the REST API with the image/PDF, and it returns text lines and their bounding boxes.

Azure OCR might be faster and more accurate on complex scans, at the cost of using cloud credits. Given this is an open-source project focus, using TrOCR (which is free and local) might be preferred for consistency.

- **Post-processing OCR Output:** OCR results may contain newlines or spacing issues, especially across lines or page breaks. It's a good practice to normalize the text: remove excessive newlines, fix hyphenated word splits across lines, etc. Ensure that the final text for classification is a clean, continuous string that the LLM can read easily. If the document has multiple pages, you can either concatenate all text (with perhaps page separators) or summarize per page if extremely large.

By the end of Phase 2, for each document we should have a variable (or file) containing the **raw text content** of the document. If the document is very large (dozens of pages), you might note the length – since feeding an extremely long text directly to an LLM might require chunking in the next phase.

## Phase 3: Embedding and Vector Database Setup

**Goal:** Generate vector embeddings from the document text and store them in the Qdrant vector database for later retrieval. This enables the "Retrieval" part of RAG, allowing semantic similarity search among documents or between documents and predefined category descriptions.

**3.1 Embedding Model:** We need to choose a model to convert text into embeddings (dense vector representations). Options include: - **Sentence Transformers:** e.g., `all-MiniLM-L6-v2` (a lightweight 384-dim embedding model) or `multi-qa-MiniLM-cos-v1`. These are quick and work well for semantic similarity. For better accuracy on legal text, models like `sentence-transformers/all-mpnet-base-v2` or larger ones can be used (768-dim vectors). If using Qdrant's `fastembed` feature, it may use a default model behind the scenes [9]. - **OpenAI text-embedding-ada-002:** (Only if allowed, though not fully open-source.) Since we focus on open-source, we'll use a local model instead. - **Custom Legal Embeddings:** If you have a taxonomy of legal document types, you could also fine-tune an embedding model or use a model like `legal-BERT` for vectorization. Initially, a general model is sufficient.

For each document, you might consider **chunking** it into smaller passages before embedding (as is common in RAG for long documents [2]). However, for classification, we often want an embedding of the whole document's content or at least the key parts: - If the document is not too large (fits in a few thousand tokens), you can embed the entire text as one vector. - If it's very large, one approach is to split into chunks (e.g., by page or sections), embed each chunk, and either store all chunk vectors (for future QA use cases) or aggregate them (e.g., averaging vectors) to get an overall document vector. Another approach is to generate a concise summary of the document and embed that summary to represent the document's meaning.

**3.2 Qdrant Setup:** By now Qdrant should be running (from Phase 1). We need to create a collection in Qdrant and define the vector dimension. - If using Qdrant via Python client, you can do:

```
from qdrant_client import QdrantClient
client = QdrantClient(url="http://localhost:6333")  # or your Qdrant URL
client.recreate_collection(collection_name="documents", vector_size=768,
distance="Cosine")
```

Here, vector_size should match the embedding dimension (e.g., 384 or 768). Cosine distance is common for similarity [10] . - **Store the embedding:** Once you have the document's embedding (a list of floats) and perhaps some metadata, upsert it to Qdrant. For example:

```
vector = embed_model.encode(document_text)
client.upsert(
    collection_name="documents",
    points=[{
       "id": doc_id,
       "vector": vector,
       "payload": {"file_name": filename, "sharepoint_id": item_id, "text":
document_text[:1000]}
    }]
)
```

You can store a snippet of the text or other info in the payload. This helps in debugging or in providing context to the LLM. Qdrant will return an ID for the point if not provided [11] . - **Optional – Category Vectors:** If you already know the list of possible Document Types and Categories, you can **embed descriptions of each category** and store those as well in a separate collection or with a tag in the payload. For example, create a small JSON with definitions of each Document Type (e.g., *"Contract: A legal agreement between parties", "Pleadings: Formal statements of cause of action",* etc.) and store their embeddings in a `categories` collection. This way, at classification time, you could query which predefined category vector is closest to the document vector. This is an alternative classification approach using nearest-neighbor search in the embedding space.

**3.3 Validation:** After storing, try a sample similarity search to ensure Qdrant is working:

```
results = client.search(collection_name="documents", query_vector=vector, top=3)
```

This should return the most similar documents to the one you just inserted (which might be itself and perhaps others if similar). The vector database can later be used to find related docs for a new input or to find which category description is most similar.

By end of Phase 3, every document has a corresponding vector in Qdrant, enabling semantic retrieval. This lays the groundwork for augmenting the LLM's input with relevant context (if needed) during classification.

## Phase 4: LLM Classification Module

**Goal:** Use an open-source Large Language Model to classify the document's content into "Document Type" and "Document Category". This step involves prompt engineering and possibly retrieval of additional context from Qdrant to help the LLM make an informed decision.

**4.1 Choosing an Open-Source LLM:** The model must be capable of understanding legal text and following instructions to output structured classification. A few candidates:

- **Mistral 7B (Instruct Variant):** Mistral is a 7.3B parameter model released under Apache 2.0 license (no usage restrictions) [12]. Despite its relatively small size, it outperforms older 13B models in general tasks. The Instruct fine-tuned versions of Mistral (e.g., `Mistral-7B-Instruct-v0.2`) are optimized to follow prompts and would be suitable for classification tasks. This model would require ~16GB VRAM to load in half precision, which fits in a single GPU on Azure VM. It's a pure text model, so it would rely on extracted text (OCR output) for input.
- **LLaVA 1.5 or LLaVA-Next:** LLaVA (Large Language and Vision Assistant) is a multimodal model that can accept images as input by combining a vision encoder with an LLM. LLaVA-1.5 introduced a simpler design with strong performance on visual QA tasks, and LLaVA-NeXT (a.k.a LLaVA 1.6) further improved reasoning and OCR capabilities [13] [14]. A model like `LLaVA-1.5 Mistral 7B` essentially pairs a CLIP-like image encoder with the Mistral language model, enabling the model to "see" images. The advantage of LLaVA in this project would be the ability to feed a scanned document image directly to the model and ask, "What type of document is this?" – it could then internally do OCR and reasoning. However, using LLaVA solely for classification might be overkill since we already have an OCR pipeline. If you want to avoid a separate OCR step, LLaVA is an option: it was explicitly designed to handle OCR in images and improve world knowledge reasoning [14]. Keep in mind running LLaVA is more complex (needs loading a vision encoder and possibly large model weights, e.g., a 34B variant exists which is heavy).
- **Other models:** There are other open models like **Llama 2** (community license), **Vicuna** (based on Llama), **Falcon 7B/40B**, etc. Llama 2 13B could be used if the license terms (non-commercial use) are acceptable or for prototyping. **GPT4All** variants or **Dolly 2.0** could also do basic classification. For our purposes, Mistral 7B Instruct is a good starting point for an Apache-licensed model.

Given the above, we'll assume using a Mistral 7B instruct model for classification (and note that a LLaVA variant could be explored for direct image input if needed).

**4.2 Prompt Engineering:** We need to craft prompts that reliably elicit the correct Type and Category from the LLM. There are two fields to output, so we must instruct the model accordingly. A few strategies: - **Direct zero-shot prompt:** Provide the document text (or a summary) and ask the model to identify the type and category. For example:

```
You are a legal document classification assistant.
Determine the Document Type and Document Category of the following document.

Document text:
"<full text of the document>"

Provide the answer in the format:
Document Type: <type>
Document Category: <category>
```

This straightforward instruction often works, but the model might need guidance on what the possible types or categories are (to avoid arbitrary answers). If you have a fixed set of Types/Categories, you should mention them. - **Candidate labels in prompt:** If the list of Types and Categories is known, you can phrase the task as a choice. For instance:

```
Classify the document into one of the following Document Types and one of the
following Document Categories.

Possible Document Types: Contract, Court Filing, Correspondence, Legal Memo,
Other.
Possible Document Categories: Corporate, Litigation, Intellectual Property,
Employment, Other.

Document text: "...(text)..."

Output format: Type - <Type>; Category - <Category>
```

Providing the options helps constrain the model. This is similar to how zero-shot classification pipeline works by considering candidate classes [15] . - **RAG Context augmentation:** This is where Qdrant comes into play. Before calling the LLM, you can use the document's embedding to **retrieve** relevant information that might help classification: - **Similar documents:** Find the nearest neighbors of this document in the vector database. If any of those have already been classified (in a log or metadata), their types could hint at what this document is. (This requires that some documents in the collection are pre-classified or that an initial round with the model has labeled a few.) - **Category definitions:** As mentioned, store short definitions or prototypical examples of each category in Qdrant. Then for a new doc, do a similarity search among those definitions using the doc's embedding. You might retrieve the top 1-3 category definitions that are most similar to the doc. These can be prepended to the prompt as context, e.g., "The document is similar to category X which is defined as ...". - **Keyword hints:** Alternatively, you could create a simple rules-based extraction of key terms in the document (like if words "Agreement", "Party", "Clause" appear, it's likely a Contract). These hints can be given to the LLM. However, a well-prompted LLM might not need this if it sees the content.

For example, using category definitions:

```
# Retrieve top category descriptions similar to the document
cat_results = cat_client.search(collection_name="doc_categories",
query_vector=document_vector, top=2)
for res in cat_results:
    print(res.payload["category_name"], res.payload["description"])
```

Suppose this returns something like Category "Litigation" and "Corporate" descriptions. You can then prompt:

```
Document excerpt: "<first 500 words of doc>"

The document is likely related to one of these categories:
- Litigation: legal documents related to lawsuits, court filings, case law.
- Corporate: documents related to corporate transactions, company records,
agreements.

Question: What is the Document Type and Document Category of this document?
```

The model can use those hints to make a decision and will hopefully output one of the provided categories.

**4.3 Few-shot Examples (optional):** If the model's responses are not consistent, you can include a few-shot prompt (i.e., give an example of a document and its classification as a demonstration). For instance: "Example: [text of known NDA] -> Type: Contract; Category: Corporate. Now classify the following document: [text]." However, this will consume more context window. With a 7B model, context length might be limited (Mistral has 4k tokens context by default; some variants may extend this). Ensure the combined prompt (instructions + any context + document text) stays within the model's limit.

**4.4 Running Inference:** Use the Hugging Face Transformers pipeline or the model's generate function to get the output. For Mistral (if using HF):

```
from transformers import AutoModelForCausalLM, AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("mistralai/Mistral-7B-Instruct-v0.1")
model = AutoModelForCausalLM.from_pretrained("mistralai/Mistral-7B-Instruct-
v0.1", device_map="auto", torch_dtype=torch.float16)
prompt = ...  # crafted as above
inputs = tokenizer(prompt, return_tensors="pt").to("cuda")
outputs = model.generate(**inputs, max_new_tokens=100)
result = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(result)
```

The result should ideally be a short text containing the type and category. We will need to parse it (e.g., using simple string matching or regex to find the categories). It's important to **validate the LLM output** – since it's not guaranteed the model will follow format perfectly. You might get an answer like: *"Document Type: Contract; Document Category: Corporate"* which is perfect. But sometimes the model might elaborate or add uncertainty. We can post-process to extract the keywords we need.

**4.5 Verification:** It's wise to include some basic checks. For example, if the model's confidence is in question, or you want a second opinion, you could run a simpler classifier in parallel (like a zero-shot classification pipeline with a model like BART MNLI [15] as shown in the Medium example) to cross-verify the category [16]. Large LLMs usually do well in zero-shot legal text classification [17], but combining approaches can improve reliability.

By the end of Phase 4, we have the LLM's determined **Document Type** and **Document Category** for the file. These will be passed to the next phase for output storage. It's helpful to also log the raw LLM output and perhaps a confidence score if you derive one (for instance, if you used a prompt that asked the LLM to give a reasoning or a score).

## Phase 5: Writing Back Results to SharePoint and Logging

**Goal:** Take the classification results and update the SharePoint document's metadata, and also record the result in a CSV log.

**5.1 Update SharePoint Metadata:** We have the SharePoint item's identifier from Phase 2 (either the item's ID in the library or the file's URL). We also have two new values to set (Document Type, Document Category). Using **PnP PowerShell** (or Graph API) we can update the list item. PnP PowerShell provides a straightforward cmdlet `Set-PnPListItem` to update fields by internal name [18]:

```
# Assume $itemId is the list item ID, and $libraryName is the document library
name
$docTypeValue = "Contract"
$docCategoryValue = "Corporate"
Set-PnPListItem -List $libraryName -Identity $itemId -Values @{"DocumentType" =
$docTypeValue; "DocumentCategory" = $docCategoryValue}
```

This will set the **DocumentType** and **DocumentCategory** columns for the item to the desired values. (Note: you must use the *internal name* of the columns, which might be something like "DocumentType" even if the display name is "Document Type" [19]. SharePoint's column internal names can be checked in the list settings.)

If you prefer using Python and Graph API: you would send an HTTP PATCH request to the SharePoint item's fields endpoint. For example,

```
PATCH /sites/{site-id}/lists/{list-id}/items/{item-id}/fields
Content-Type: application/json
{
   "DocumentType": "Contract",
   "DocumentCategory": "Corporate"
}
```

along with an OAuth Bearer token. This achieves the same result. Using PnP is simpler if you can run PowerShell in the environment. In Azure ML (Linux) you can install PowerShell Core and PnP, or run a separate process. On an Azure Windows VM, it's even easier since PowerShell is native.

**5.2 Append to CSV Log:** Maintain a CSV file (e.g., `classification_log.csv`) either locally**5.2 Append to CSV Log:** Maintain a CSV file (e.g., `classification_log.csv`) either on the VM's disk or in a blob storage. Each time a document is classified, append a line with details such as: *Document ID, File Name, Document Type, Document Category, Timestamp*. This provides an audit trail outside of SharePoint. Using Python's `csv` module or pandas is straightforward for this. For example:

```python
import pandas as pd
log_entry = {
    "Timestamp": pd.Timestamp.now(),
    "FileName": filename,
    "DocumentType": doc_type,
    "DocumentCategory": doc_category,
    "SharePointID": item_id
}
pd.DataFrame([log_entry]).to_csv("classification_log.csv", mode='a',
header=False, index=False)
```

Ensure that the CSV is stored in a secure location if it contains sensitive info (perhaps on an Azure File Share or Blob container). This log is useful for verifying that all files were processed and for debugging misclassifications.

**5.3 Notification (Optional):** If desired, the pipeline can send a notification or write to a Teams channel when classifications are done (e.g., “Document X has been classified as Y”). This can be done via Logic App or a simple webhook. It’s not required, but can enhance visibility.

After Phase 5, the SharePoint library items are enriched with metadata indicating their document type/category, and a comprehensive log exists. Users can filter or search in SharePoint by these metadata fields, which helps organize the legal documents.

## Phase 6: Migration to Arandia Law Firm’s Tenant (Long-Term)

In the long run, the entire solution may need to move to the Arandia Law Firm’s Azure tenant (and their SharePoint). Planning for migration from the start will save effort later. Here’s the guidance for a smooth transition:

- **Separate Configuration:** Keep all tenant-specific configurations in one place (endpoints, credentials, site URLs). For example, the SharePoint site URL, library name, and any client IDs should be in a config file or environment variables. This way, switching to the new tenant’s SharePoint just means updating the config.
- **Infrastructure Redeployment:** Azure does not support a one-click “tenant to tenant” transfer for resources [20]. Instead, you will recreate resources in the new tenant’s Azure subscription:
- Provision a similar VM (or Azure ML setup) in the Arandia tenant. Install the same dependencies. Using infrastructure-as-code (ARM templates, Terraform, or Azure CLI scripts) to automate VM setup can be very helpful here.
- Deploy Qdrant on the new infrastructure. If using a self-hosted Qdrant, you can backup the data from the original (Qdrant stores data on disk, so you could copy the disk file or export the collection) and restore it on the new instance. Alternatively, re-run the embedding process on all documents in the new environment to rebuild the vector index (this might be simpler if the number of documents is not huge or if you can script the re-indexing).
- **Data Migration:** The documents themselves need to reside in Arandia’s SharePoint. If they already exist there, you will simply point the pipeline to that library. If not, you might need to migrate documents from the current SharePoint to Arandia’s SharePoint. This could be done via a content

migration tool or manual upload. Once the documents are in the new library, the pipeline can classify them anew (or copy over the metadata if it was exported).

- **Credentials and Access:** Set up new service accounts or app registrations in the Arandia tenant for access to SharePoint and any Azure resources:
- For PnP PowerShell or Graph API, register an Azure AD app in Arandia tenant with Sites.ReadWrite.All permission on SharePoint, or have a user account with appropriate SharePoint permissions. Update the authentication in your scripts to use the new app or credential.
- If using Azure Cognitive Services for OCR in the new tenant, provision those services and update keys in config.
- **Testing in New Tenant:** Before shutting down the old system, run the classification on a sample in the new environment to ensure everything (OCR, embedding, LLM, SharePoint update) works with Arandia's setup. Pay attention to any differences in SharePoint (e.g., content types or library settings) that might require adjusting field names or API calls.
- **Switchover:** Coordinate with stakeholders to perhaps run both systems in parallel for a short period. During migration, there might be a freeze on new document uploads or you have to re-process docs uploaded during the transition.
- **Shut Down Old Environment:** Once fully migrated, decommission the Azure resources in the sponsorship account to avoid confusion and cost. Keep backups of logs or any data if needed.

By following these steps, the move to Arandia's tenant will mostly involve re-deploying the open-source components and repointing to the new data sources. The modular design (separating OCR, vector DB, LLM, etc.) ensures that as long as each piece is set up in the new environment, the overall pipeline remains the same.

## Custom GPT Project Instructions (for Integration)

If integrating this solution into a **Custom GPT** (such as an Azure OpenAI custom chatbot or similar assistant), it's important to define the system's behavior and context clearly. Below are suggested instructions segmented by purpose, which can guide the GPT model to function as a document classification assistant:

- **System Behavior:** The AI system should behave as a *professional legal document assistant* that only outputs the document's type and category, optionally with a brief reasoning if asked. It should strictly follow the classification guidelines and not divulge any confidential content. The system should use the tools at its disposal (OCR, vector search) whenever needed to ensure it has the necessary information before answering. It should refrain from answering unrelated questions or anything beyond its scope (for example, if asked for legal advice, it should politely decline).
- **Tone:** The tone should be **formal, concise, and objective**. Since this is for internal classification, the language can be straightforward and technical. If interacting with a user (e.g., responding in a chat), it should be polite and use correct legal terminology. Example tone: "The document appears to be a **Contract** and falls under the **Corporate** category." The model should avoid casual language or jokes, as the context is a law firm's document management.
- **Goals:** The primary goal is to correctly identify the **Document Type** and **Document Category** for each input document. Secondary goals include explaining the reasoning (if needed for audit) and ensuring the information is logged correctly. The GPT should aim for high accuracy and should defer (or flag uncertainty) if it does not have high confidence. It should also be mindful of the boundaries:

if a document doesn't fit any known category, it can use "Other" or request human review, rather than guessing incorrectly.

- **Context:** The system has access to the content of documents (either as text or images via OCR). It also has context in the form of predefined category definitions and possibly examples of each type of document. For instance, the system knows that *"Contracts"* typically contain language like "Agreement", "Parties", "Terms", whereas *"Court Filings"* contain case captions, case numbers, etc. This domain knowledge is part of its context. Additionally, the vector database provides context by showing similar documents and their classifications or relevant reference text. In a running session, the context for each query might include the extracted text of the document and any retrieved references (such as "This document is similar to a Purchase Agreement in our records"). The GPT should use this context to ground its classification and avoid hallucination.
- **Relevant Tools:** This custom GPT is augmented with tools to help in classification:
- **OCR Tool** – The GPT can call an OCR function on images or PDFs to get text. In practice, this means the system will have already done OCR and provided the text, but conceptually, the GPT knows that for an image it should use OCR.
- **Vector Search Tool** – The GPT can query the Qdrant vector database to retrieve information. For example, it might use a tool like `SearchSimilarDocuments` with a chunk of text to see if there are known similar documents or category definitions. The results of this tool (like "Most similar stored document is 'Lease Agreement' of type Contract") can be used by the GPT in making a decision.
- **Knowledge Base** – A curated list of document type definitions or even a decision tree can be considered a tool. The GPT can access definitions: e.g., *DocumentTypeDefinitions["NDA"]* to recall what an NDA usually contains.

In Azure OpenAI's custom deployment, these "tools" would be implemented via an orchestration layer (like an Azure Function for OCR, or using the `RetrievalQA` approach for vector search). The GPT's instructions should mention when to use these: *"If the document text is not provided, first use the OCR tool to extract text. Next, use the Vector Search tool with key phrases from the document to see suggested categories."* By explicitly listing the steps, the GPT (or the system orchestrating it) will ensure each tool is used appropriately to gather all info before the final answer.

- **Example Instruction to GPT:**
- *System message:* "You are an AI assistant that classifies legal documents. You have the following tools: an OCR reader and a semantic search. Your responses should be factual and concise. Focus only on determining the document's type and category, nothing more."
- *User message (or developer prompt):* "Classify the uploaded document."
- *Assistant thinking:* (The system would use OCR tool on the file, then vector search, then...)
- *Assistant final answer:* "**Document Type:** Contract \n **Document Category:** Corporate" (for example).

By structuring the custom GPT project with clear sections for behavior, tone, goals, context, and tools, we ensure that the AI operates within the desired boundaries and produces consistent results. These instructions can be fed as the system prompt in an Azure OpenAI Service chat deployment or any orchestrated GPT solution, thereby aligning the model with the law firm's requirements and communication style.

## References

- Azure AI Services Documentation – *Optical Character Recognition (OCR)* [8]
- Qdrant Documentation – *Storing and querying embeddings* [1] [11]

• Mistral AI Announcement – *Open-source 7B model (Apache 2.0 license)* [12]
• Liu et al. (2023) – *LLaVA-1.5 and LLaVA-NeXT improvements (multimodal OCR and reasoning)* [13] [14]
• Tiwari, R. – *Zero-shot Legal Document Classification (LLM approach)* [16]
• PnP PowerShell Reference – *Updating SharePoint list item fields* [18]

---

[1] Multimodal Knowledge Extraction and Retrieval System for Generative AI Agents and RAG Systems
https://techcommunity.microsoft.com/blog/educatordeveloperblog/enhancing-retrieval-augmented-generation-with-a-multimodal-knowledge-extraction-/4241375

[2] Design and Develop a RAG Solution - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/rag/rag-solution-design-and-evaluation-guide

[3] Mistral 7B: Recipes for Fine-tuning and Quantization on Your ...
https://kaitchup.substack.com/p/mistral-7b-recipes-for-fine-tuning

[4] [9] [11] Quickstart — Qdrant Client documentation
https://python-client.qdrant.tech/quickstart

[5] Powershell script - Sharepoint- set metadata values for more than one item. - Microsoft Q&A
https://learn.microsoft.com/en-us/answers/questions/872129/powershell-script-sharepoint-set-metadata-values-f

[6] TrOCR - Hugging Face
https://huggingface.co/docs/transformers/en/model_doc/trocr

[7] Document Classification with LayoutLMv3 | MLExpert
https://www.mlexpert.io/blog/document-classification-with-layoutlmv3

[8] Azure AI Vision with OCR and AI | Microsoft Azure
https://azure.microsoft.com/en-us/products/ai-services/ai-vision

[10] FastEmbed: Qdrant's Efficient Python Library for Embedding ...
https://qdrant.tech/articles/fastembed/

[12] Mistral 7B | Mistral AI
https://mistral.ai/news/announcing-mistral-7b

[13] [14] LLaVA-NeXT
https://huggingface.co/docs/transformers/v4.39.2/model_doc/llava_next

[15] [16] Zero-shot Learning for Document Classification: A Powerful Approach for Adaptable Text Analysis | by Ranjeet Tiwari | Senior Architect - AI | IITJ | Medium
https://medium.com/@AI-Simplified/zero-shot-learning-for-document-classification-a-powerful-approach-for-adaptable-text-analysis-66a75c6dd480

[17] The unreasonable effectiveness of large language models in zero ...
https://www.frontiersin.org/journals/artificial-intelligence/articles/10.3389/frai.2023.1279794/full

[18] [19] Set-PnPListItem | PnP PowerShell
https://pnp.github.io/powershell/cmdlets/Set-PnPListItem.html

[20] Moving resources to another tenant, is this supported? : r/AZURE
https://www.reddit.com/r/AZURE/comments/1ix5029/moving_resources_to_another_tenant_is_this/