```
%pip install llama_index ftfy regex tqdm
%pip install llama-index-vector-stores-qdrant llama-index-embeddings-clip
%pip install git+https://github.com/openai/CLIP.git
%pip install torch torchvision
%pip install matplotlib scikit-image
%pip install qdrant_client
%pip install pymupdf
%pip install qdrant_client
%pip install llama_index.embeddings.huggingface


import os
import fitz  # PyMuPDF
from PIL import Image
import io
import hashlib

import qdrant_client
from llama_index.vector_stores.qdrant import QdrantVectorStore
from llama_index.core import StorageContext, Settings, SimpleDirectoryReader
from llama_index.core.indices import MultiModalVectorStoreIndex
from llama_index.embeddings.huggingface import HuggingFaceEmbedding


# --------------------
# Utility: deterministic file name
# --------------------
def make_filename(base_name, page, suffix, ext="jpg"):
    return f"{os.path.splitext(base_name)[0]}_p{page}_{suffix}.{ext}"


# --------------------
# Extraction Pipeline
# --------------------
def extract_images(pdf_path, method="both", dpi=200):
    """
    Extract images from PDF.
    method = "embedded" | "fullpage" | "both"
    Saves images into /artifacts/embedded and /artifacts/fullpage
    """
    pdf_document = fitz.open(pdf_path)

    file_name = os.path.basename(pdf_path)
    pdf_dir = os.path.dirname(pdf_path)

    # Output dirs
    artifact_dir = os.path.join(pdf_dir, "artifacts")
    embedded_dir = os.path.join(artifact_dir, "embedded")
    fullpage_dir = os.path.join(artifact_dir, "fullpage")

    if method in ["embedded", "both"]:
        os.makedirs(embedded_dir, exist_ok=True)
    if method in ["fullpage", "both"]:
        os.makedirs(fullpage_dir, exist_ok=True)
    for page_num, page in enumerate(pdf_document):
        page_number = page_num + 1

        # --- Embedded images ---
        if method in ["embedded", "both"]:
            for img_index, img in enumerate(page.get_images(full=True)):
                xref = img[0]
                base_image = pdf_document.extract_image(xref)
                image_bytes = base_image["image"]
                image_ext = base_image["ext"]

                filename = make_filename(file_name, page_number, f"emb{img_index+
                filepath = os.path.join(embedded_dir, filename)
                with open(filepath, "wb") as f:
                    f.write(image_bytes)

        # --- Full page render ---
        if method in ["fullpage", "both"]:
            zoom = dpi / 72
            mat = fitz.Matrix(zoom, zoom)
            pix = page.get_pixmap(matrix=mat)
            img_bytes = pix.tobytes("png")
            img = Image.open(io.BytesIO(img_bytes))

            filename = make_filename(file_name, page_number, "page", "jpg")
            page_img_path = os.path.join(fullpage_dir, filename)
            img.convert("RGB").save(page_img_path, "JPEG", quality=90)

    pdf_document.close()


# -------------------
# Qdrant Setup
# -------------------
embed_model_text = HuggingFaceEmbedding(model_name="BAAI/bge-small-en-v1.5")
embed_model_image = HuggingFaceEmbedding(model_name="sentence-transformers/clip-V

Settings.chunk_size = 512  # no global embed_model
Settings.embed_model = embed_model_image

client = qdrant_client.QdrantClient(path="qdrant_db_test")

text_store = QdrantVectorStore(
    client=client,
    collection_name="text_collection",
    embed_model=embed_model_text,
)

image_store = QdrantVectorStore(
    client=client,
    collection_name="image_collection",
    embed_model=embed_model_image,
)
```

```
storage_context = StorageContext.from_defaults(
    vector_store=text_store,
    image_store=image_store,
)



# --------------------
# Main function: add PDF to index
# --------------------
def add_pdf_to_index(doc_dir, pdf_path, method="both"):
    # Step 1: Extract images
    extract_images(pdf_path, method=method)

    # Step 2: Use SimpleDirectoryReader to load extracted artifacts
    pdf_dir = os.path.dirname(pdf_path)
    embedded_artifact_dir = os.path.join(pdf_dir, "artifacts/embedded")
    fullpage_artifact_dir = os.path.join(pdf_dir, "artifacts/fullpage")

    documents = SimpleDirectoryReader(doc_dir).load_data()
    if os.path.exists(embedded_artifact_dir):
      documents += SimpleDirectoryReader(embedded_artifact_dir).load_data()
    if os.path.exists(fullpage_artifact_dir):
      documents += SimpleDirectoryReader(fullpage_artifact_dir).load_data()

    if not documents:
        print(f"No documents found for {pdf_path}")
        return None

    # Step 3: Insert into multimodal index
    index = MultiModalVectorStoreIndex.from_documents(
        documents,
        storage_context=storage_context,
        show_progress=True,
    )
    return index




# Upload a file from frontend, then make a folder for that pdf and put it in ther
doc_dir = "data_fin/Duravent"
pdf_path = "data_fin/Duravent/DURAVENT_SK_LD_CT-001E_02-24_v3.pdf"
ll_index = add_pdf_to_index(doc_dir, pdf_path, method="fullpage")

doc_dir = "data_fin/Nvidia"
pdf_path2 = "data_fin/Nvidia/NVIDIA-2025-Annual-Report.pdf"
ll_index = add_pdf_to_index(doc_dir, pdf_path2, method="embedded")

print(client.get_collections())


from llama_index.core.query_engine import CustomQueryEngine
from llama_index.core.retrievers import BaseRetriever
from llama_index.core.schema import ImageNode, NodeWithScore, MetadataMode, Te
from llama_index.core.prompts import PromptTemplate
from llama_index.core.base.response.schema import Response
from google import genai
from google.genai import types
from typing import Optional, List
import base64
from pydantic import BaseModel, Field
import json
from google.colab import userdata

llm_client = genai.Client(api_key=userdata.get("GOOGLE_API_KEY"))

QA_PROMPT_TMPL = """Below we give parsed text and images as context.

Use both the parsed text and images to answer the question.
Write your response in markdown

---------------------
{context_str}
---------------------
Given the context information and not prior knowledge, answer the query. Expl
from the text or image, and if there's discrepancies, and your reasoning for

Query: {query_str}
Answer: """

QA_PROMPT = PromptTemplate(QA_PROMPT_TMPL)


class Structure(BaseModel):
    text_response: str = Field(description="Text response from the LLM")
    file_name_used: List[str] = Field(description="List of image nodes file_n


class MultimodalGeminiEngine(CustomQueryEngine):
    """More robust version with better error handling and image format detect

    qa_prompt: PromptTemplate
    retriever: BaseRetriever

    def __init__(self, qa_prompt: Optional[PromptTemplate] = None, **kwargs)
        """Initialize."""
        super().__init__(qa_prompt=qa_prompt or QA_PROMPT, **kwargs)

    def _get_image_mime_type(self, image_path: str) -> str:
        """Determine MIME type from file extension."""
        extension = image_path.lower().split('.')[-1]
        mime_types = {
            'jpg': 'image/jpeg',
            'jpeg': 'image/jpeg',
            'png': 'image/png',
            'gif': 'image/gif',
            'webp': 'image/webp',
            'bmp': 'image/bmp'
        }
        return mime_types.get(extension, 'image/jpeg')  # Default to JPEG
    def _process_image_node(self, image_node: ImageNode) -> Optional[types.Pa
        """Process a single image node into a GenAI Part."""

        # Try each method in order of preference
        methods = [
            self._try_base64_image,
            self._try_resolve_image,
            self._try_file_path,
            self._try_image_url
        ]

        for method in methods:
            try:
                part = method(image_node)
                if part is not None:
                    return part
            except Exception as e:
                continue

        print(f"Warning: Could not process ImageNode {image_node.id_}")
        return None

    def _try_base64_image(self, image_node: ImageNode) -> Optional[types.Part
        """Try to get image from base64 encoded data."""
        try:
          if hasattr(image_node, 'image') and image_node.image:
            image_bytes = base64.b64decode(image_node.image)
            return types.Part.from_bytes(data=image_bytes, mime_type="image/j
          return None
        except Exception as e:
          return f"ERROR : {e}"

    def _try_resolve_image(self, image_node: ImageNode) -> Optional[types.Par
        """Try to get image using resolve_image method."""
        if hasattr(image_node, 'resolve_image'):
            image_buffer = image_node.resolve_image()
            image_bytes = image_buffer.getvalue()
            return types.Part.from_bytes(data=image_bytes, mime_type="image/j
        return None

    def _try_file_path(self, image_node: ImageNode) -> Optional[types.Part]:
        """Try to get image from file path."""
        if hasattr(image_node, 'image_path') and image_node.image_path:
            with open(image_node.image_path, 'rb') as f:
                image_bytes = f.read()
            mime_type = self._get_image_mime_type(image_node.image_path)
            return types.Part.from_bytes(data=image_bytes, mime_type=mime_typ
        return None

    def _try_image_url(self, image_node: ImageNode) -> Optional[types.Part]:
        """Try to get image from URL."""
        if hasattr(image_node, 'image_url') and image_node.image_url:
            # For cloud storage URLs (gs://) or public URLs
            return types.Part.from_uri(
                file_uri=image_node.image_url,
                mime_type="image/jpeg"
            )
        return None

    def custom_query(self, query_str: str):
        """Execute the query with robust image processing."""
        # Retrieve nodes
        nodes = self.retriever.retrieve(query_str)
        img_nodes = [n for n in nodes if isinstance(n.node, ImageNode)]
        text_nodes = [n for n in nodes if isinstance(n.node, TextNode)]

        # Create context string
        context_str = "\n\n".join(
            [r.get_content(metadata_mode=MetadataMode.LLM) for r in nodes]
        )

        fmt_prompt = self.qa_prompt.format(context_str=context_str, query_str=

        # Prepare content parts
        content_parts = [fmt_prompt]

        # Process image nodes
        successful_images = 0
        for img_node in img_nodes:
            image_part = self._process_image_node(img_node.node)
            if image_part:
                content_parts.append(img_node.get_content(metadata_mode=Metada
                content_parts.append(image_part)
                successful_images += 1

        print(f"Successfully processed {successful_images}/{len(img_nodes)} in

        try:
            # Generate content with structured output
            response = llm_client.models.generate_content(
                model="gemini-2.5-flash",
                contents=content_parts,
                config={
                    "response_mime_type": "application/json",
                    "response_schema": Structure,
                }
            )

            structured_response = response.parsed
            full_response = f"{structured_response.text_response}\n\nImages U

        except Exception as e:
            print(f"Structured output failed: {e}")
            # Fallback to regular response
            try:
                response = client.models.generate_content(
                    model="gemini-2.5-flash",
                    contents=content_parts,
                )
                full_response = response.text
                structured_response = None
                structured_response = None
            except Exception as fallback_error:
                print(f"Fallback failed: {fallback_error}")
                full_response = f"Error: {str(e)}"
                structured_response = None

        return Response(
            response=full_response,
            source_nodes=nodes,
            metadata={
                "text_nodes": text_nodes,
                "image_nodes": img_nodes,
                "successful_images": successful_images,
                "total_images": len(img_nodes),
                "structured_response": structured_response
            }
        )
```

```
ever = ll_index.as_retriever(similarity_top_k=70, image_similarity_top_k=10)
```

```
engine = MultimodalGeminiEngine(retriever=retriever)
```

```
response = engine.custom_query("What was nvidia's perfomance in this year")
```

```
from IPython.display import Markdown, display
resp = str(response)
display(Markdown(resp))
```