```
import os
import json
import numpy as np
from pathlib import Path
from typing import List, Dict, Any
from dotenv import load_dotenv
from langdetect import detect
from PyPDF2 import PdfReader
import faiss
import torch
from transformers import BertTokenizer, BertForQuestionAnswering
from transformers import pipeline

# Load environment variables (e.g., API keys, model paths)
load_dotenv()

# Define the path to the financial policy PDF document
pdf_path = "জ্বালানি নীতিমালা ২০২৫ (গেজেট)"

# Load BERT model and tokenizer for Question Answering
tokenizer = BertTokenizer.from_pretrained('bert-large-uncased')
model = BertForQuestionAnswering.from_pretrained('bert-large-uncased')

# Embeddings Backend Setup (SentenceTransformers for embeddings)
from sentence_transformers import SentenceTransformer

# Define embedding model backend (can be 'sentence_transformers' or OpenAI
embeddings)
EMBEDDING_BACKEND = os.getenv("EMBEDDING_BACKEND",
"sentence_transformers")
EMBEDDING_MODEL = os.getenv("EMBEDDING_MODEL", "all-MiniLM-L6-v2")
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY", "")

# Define Faiss vector database for storing and searching text chunks
class EmbeddingEngine:
    def __init__(self):
        self.backend = EMBEDDING_BACKEND
        if self.backend == "sentence_transformers":
            self.model = SentenceTransformer(EMBEDDING_MODEL)
        else:
            if OPENAI_API_KEY:
                import openai
                openai.api_key = OPENAI_API_KEY
            else:
                self.model = None
```

```
    def embed_texts(self, texts: List[str]) -> List[List[float]]:
        """ Embed the given texts into numerical vectors """
        if not texts:
            return []
        if self.backend == "sentence_transformers":
            embeddings = self.model.encode(texts, convert_to_numpy=True)
            return embeddings.astype("float32").tolist()
        else:
            embeddings = []
            for text in texts:
                response = openai.Embedding.create(model="text-embedding-ada-002",
input=text)
                embeddings.append(response['data'][0]['embedding'])
            return embeddings

# Utility functions

def detect_language(text: str) -> str:
    """ Detect the language of the given text (e.g., English, Bengali) """
    try:
        return detect(text)
    except Exception:
        return "en"

def chunk_text(text: str, chunk_size: int = 1000, overlap: int = 200) -> List[str]:
    """ Chunk the text into smaller parts for processing """
    chunks = []
    start = 0
    while start < len(text):
        end = min(start + chunk_size, len(text))
        chunk = text[start:end].strip()
        if chunk:
            chunks.append(chunk)
        start = end - overlap
    return chunks

# ------------------------
# Ingest PDF & Build Index
# ------------------------

def read_pdf_by_page(pdf_path: str) -> List[Dict[str, Any]]:
    """ Extract text from each page of the PDF """
    reader = PdfReader(pdf_path)
    pages = []
    for i, page in enumerate(reader.pages):
        try:
            text = page.extract_text() or ""
```

```
        except Exception:
            text = ""
        pages.append({"page_number": i + 1, "text": text})
    return pages

def build_faiss_index(pages: List[Dict[str, Any]], index_dir: str, chunk_size=1000,
overlap=200):
    """ Create a Faiss index from the embedded text chunks """
    index_dir = Path(index_dir)
    index_dir.mkdir(parents=True, exist_ok=True)
    docs, metadatas = [], []

    for p in pages:
        page_num = p["page_number"]
        text = p["text"]
        chunks = chunk_text(text, chunk_size, overlap)
        for i, c in enumerate(chunks):
            docs.append(c)
            metadatas.append({"page": page_num, "chunk": i, "preview": c[:300]})

    if len(docs) == 0:
        raise ValueError("No text extracted from PDF.")

    # Create embeddings
    engine = EmbeddingEngine()
    embeds = engine.embed_texts(docs)
    xb = np.array(embeds).astype("float32")
    dim = xb.shape[1]

    # Create a Faiss index and save it
    index = faiss.IndexFlatL2(dim)
    index.add(xb)
    faiss.write_index(index, str(index_dir / "index.faiss"))

    with open(index_dir / "metadata.json", "w", encoding="utf-8") as f:
        json.dump({"docs": docs, "metadatas": metadatas}, f, ensure_ascii=False, indent=2)

    print(f"Index and metadata saved to {index_dir}")

# ------------------------
# Question Answering with BERT
# ------------------------

def qa_bert_model(query: str, context: str):
    """ Use BERT model for question answering given a context """
    inputs = tokenizer(query, context, return_tensors="pt", truncation=True, padding=True)
    with torch.no_grad():
        outputs = model(**inputs)
```

```
        answer_start_scores = outputs.start_logits
        answer_end_scores = outputs.end_logits

    start = torch.argmax(answer_start_scores)
    end = torch.argmax(answer_end_scores)

    answer =
tokenizer.convert_tokens_to_string(tokenizer.convert_ids_to_tokens(inputs.input_ids[0][start:
end + 1]))
    return answer.strip()

# ------------------------
# Retriever + QA Logic
# ------------------------

class Retriever:
    def __init__(self, index_dir: str):
        """ Load the Faiss index and metadata """
        self.index_dir = Path(index_dir)
        idx_path = self.index_dir / "index.faiss"
        meta_path = self.index_dir / "metadata.json"

        if not idx_path.exists() or not meta_path.exists():
            raise FileNotFoundError(f"Index or metadata not found in {self.index_dir}")

        self.index = faiss.read_index(str(idx_path))
        with open(meta_path, "r", encoding="utf-8") as f:
            js = json.load(f)
        self.docs = js["docs"]
        self.metadatas = js["metadatas"]
        self.engine = EmbeddingEngine()

    def search(self, query: str, k: int = 5) -> List[Dict[str, Any]]:
        """ Search for the most relevant text chunks using Faiss """
        qv = np.array(self.engine.embed_texts([query])).astype("float32")
        D, I = self.index.search(qv, k)

        results = []
        for dist, idx in zip(D[0], I[0]):
            if idx < 0 or idx >= len(self.docs):
                continue
            results.append({"score": float(dist), "text": self.docs[idx], "meta": self.metadatas[idx]})

        return results

def generate_answer(query: str, retrieved: List[Dict[str, Any]], conversation_history=None,
language="en") -> str:
    """
```

```
    Generate an answer using either BERT QA model or retrieved text snippets.
    """
    if conversation_history is None:
        conversation_history = []

    # Add the current query to the conversation history
    conversation_history.append({"role": "user", "content": query})

    # Combine context and conversation history
    context = "\n\n".join([f"(page: {r['meta']['page']}) {r['text'][:800]}" for r in retrieved])
    conversation_context = "\n\n".join([entry["content"] for entry in conversation_history])

    # Use BERT for question answering
    answer = qa_bert_model(query, context)

    # Fallback to concatenating retrieved texts if no answer is found
    if not answer:
        answer = "Sorry, I couldn't find a precise answer. Here's some related information: "
        answer += "\n\n".join([r["text"][:500] for r in retrieved[:3]])

    return answer

# ------------------------
# Main Logic
# ------------------------

def main():
    # Read PDF and create Faiss index
    pdf_path = "path_to_pdf"
    pages = read_pdf_by_page(pdf_path)
    build_faiss_index(pages, "index_dir")

    # Initialize retriever
    retriever = Retriever("index_dir")

    # Example query
    query = "What is the budget for energy projects?"

    # Retrieve relevant chunks from the index
    retrieved = retriever.search(query, k=3)

    # Generate answer using the retrieved chunks
    answer = generate_answer(query, retrieved)
    print(answer)

if __name__ == "__main__":
    main()
```