Assignment: Invoice Reimbursement System

Documentation

By

Aditya Kumar Pandey

MTech CSE (AI & DS)

IIIT Bhagalpur

Invoice Reimbursement Chatbot – RAG-based AI System

This project implements a Retrieval-Augmented Generation (RAG) chatbot to answer questions related to invoice reimbursements. It uses semantic search using ChromaDB and SentenceTransformers, and generates natural language answers using transformer-based LLMs, including Flan-T5 and Mistral 7B.

Key Features

- Utilizes Gemini 1.5 Flash for smart invoice vs. policy analysis.
- Semantic Search over embedded invoices
- Support for uploading invoice files and storing their metadata
- Question Answering with Flan-T5 and Mistral-7B
- Exposed via an interactive FastAPI backend
- Integrated in Google Colab with support for Ngrok tunneling

Technologies & Tools Used

Components	Technology/Tools used
PDF Parsing	pdfplumber
LLM – based analysis	Google.generativeai: Gemini 1.5 Flash
Concurrent processing of multiple invoices	Threading
Vector Store	ChromaDB
LLMs Used	google/flan-t5-base, mistralai/Mistral-7B-Instruct-v0.1
API Backend	FASTAPI
Ngrok Tunneling	pyngrok
Serving Model	transformers, AutoModelForCausalLM, pipeline()
Deployment (in process)	Uvicorn, Google Colab, nest_asyncio

Setup Instructions

1. Clone or Open in Google Colab

You can run this entire project inside Google Colab.

2. Install Required Packages

Python code snippet:

pip install chromadb sentence-transformers transformers fastapi uvicorn pyngrok nest_asyncio

Phase wise Implementation

Phase 1: PDF Parsing and Invoice–Policy Analysis with Gemini

Objective:

Extract text from invoice and policy PDFs, then use a Gemini LLM to determine if each invoice complies with the policy.

Tools & Tech:

- pdfplumber: PDF parsing
- google.generativeai: Gemini 1.5 Flash for LLM-based analysis
- threading: Concurrent processing of multiple invoices

Key Logic:

Gemini Analysis Function

```
# -- Gemini Analysis Function --
def analyze_invoice(file_name, invoice_text, results_dict):
    try:
        prompt = f"""
You are a reimbursement policy expert.

Policy:
{policy_text}

Invoice:
{invoice_text}

Return exactly:
Reimbursement Status: <Fully Reimbursed | Partially Reimbursed | Declined>
Reason: <One-sentence justification>
"""
        response = model.generate_content(prompt)
        results_dict[file_name] = response.text.strip()
        print(f"{file_name} analyzed.")
```

```
    except Exception as e:
        print(f"{file_name} failed: {e}")
        results_dict[file_name] = f"Error: {str(e)}"
```

Using Threads for analysis of each Invoice against HR Policy

Key Logic:

```
results = {}
threads = []

for file_name, text in invoice_texts.items():
    t = threading.Thread(target=analyze_invoice, args=(file_name, text, results))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

# -- Save Results --
with open("invoice_analysis_results.json", "w") as f:
    json.dump(results, f, indent=2)

print("\n Results saved to invoice_analysis_results.json")
```

Output:

- Extracted .txt files for each invoice
- invoice_analysis_results.json: JSON with LLM reimbursement results

Phase 2: Semantic Embedding & Storage with ChromaDB Objective:

In this phase, I semantically embed each invoice along with its Gemini-generated analysis results and store them in a vector database (ChromaDB) to enable efficient similarity search and future querying.

What This Step Does

- Loads a local SentenceTransformer model (all-MiniLM-L6-v2) to convert invoice–analysis text into dense vector representations.
- Parses and extracts metadata from invoice analysis results, including:
- Employee name (from filename)
- Reimbursement status (e.g., Fully/Partially Reimbursed)
- Reason (brief justification)

Creates a ChromaDB collection named invoice_embeddings.

Adds each invoice+analysis pair to the vector database along with extracted metadata.

Exports all embedded records to a JSON file for backup or further use.

Tools & Tech:

- sentence-transformers (all-MiniLM-L6-v2): Embedding model for transforming invoice content into vector space
- ChromaDB: Lightweight, local vector store to store and index embeddings
- Json, OS: For metadata extraction and file operations

Key Files & Paths

- Invoices Directory: /content/drive/MyDrive/invoice_reimbursement_system/data/invoices
- Analysis Results Input: /content/invoice_analysis_results.json
- Exported Embeddings: /content/invoice_embeddings_export.json

Output

- All invoices are vectorized and stored in ChromaDB.
- Metadata enriched records are saved in a JSON export for traceability.

Phase 3: Conversational Query Engine

Objective

Developed a local Conversational Query System that enables users to ask natural language questions related to invoice reimbursements and receive precise, context-aware answers. This is achieved via vector similarity search and Retrieval-Augmented Generation (RAG).

Tools & Libraries Used:

ChromaDB**:** Lightweight vector database to store and search invoice embeddings.

Sentence-Transformers : For generating semantic embeddings of invoice content (`MiniLM-L6-v2`).

Hugging Face Transformers : Used for the local text generation model (`google/flan-t5-base`).

Key Components:

3.1 Embedding Generation & Storage

- Embedding model used: all-MiniLM-L6-v2
- Data stored in a **ChromaDB collection** named "invoice_embeddings", along with metadata (e.g., employee_name, status, file_name, etc.)

Python code snippet:

```
embedding_function = SentenceTransformerEmbeddingFunction(model_name="all-MiniLM-L6-v2") collection = client.get_or_create_collection(name="invoice_embeddings", embedding_function=embedding_function)
```

3.2 Semantic Search Function

Searches invoice documents based on user query with optional metadata filtering (e.g., only retrieve documents for a certain employee or status).

Python code snippet:

```
search_invoices(query="cab invoices", filters={"employee_name": "Rahul", "status": "Rejected"})
```

- Uses ChromaDB's query() to perform vector similarity search.
- Returns results formatted in Markdown with invoice metadata and snippet.

3.3 Local RAG Pipeline

Integrates retrieved invoice documents into a context-aware prompt and uses a local LLM (flan-t5-base) to answer questions.

Python code snippet:

```
qa_pipeline = pipeline("text2text-generation", model="google/flan-t5-base")
```

- Custom prompt template designed to include:
- Retrieved invoices.
- Explicit task definition for the model.
- Example query:

Python code snippet:

```
query = "Why was Rahul's cab reimbursement partially rejected?" answer = rag_chatbot_local(query)
```

Example Output:

Answer:

Rahul's cab reimbursement was partially rejected because it exceeded the policy cap for airport drops. The approved limit was ₹600, but the submitted invoice was ₹1200.

Phase 4: API Deployment with FastAPI, Mistral LLM, ChromaDB & ngrok

This step involves deploying a question-answering system as a web API using FastAPI. It leverages a retrieval-augmented generation (RAG) pipeline, combining ChromaDB for document

retrieval and the Mistral 7B Instruct model for generating answers. The API is exposed via a public URL using ngrok, enabling real-time interaction with embedded documents.

Technologies & Tools Used

Web Framework : FASTAPI

Hosting Funnel: ngrok via pyngrok

Async pathching: nest_asyncio

Document Storage : ChromaDB

Embedding Model: all-MiniLM-L6-v2 via sentence-transformers

LLM (Anwsering) : Mistral-7B-Instruct-v0.1 via HuggingFace

Tokenizer: HuggingFace AutoTokenizer

Feautures Implemented: User Instructions

- File Upload Endpoint (/upload/): Accepts raw text files (like parsed invoices), embeds the content using a sentence transformer, and stores them in ChromaDB with metadata.
- Question Answering Endpoint (/ask/):
    - Takes a user query
    - Retrieves top matching document context from ChromaDB
    - Passes context + query to Mistral LLM
    - Returns an answer and associated document metadata
- Public Hosting:
    - Automatically creates a public URL via ngrok to allow remote usage and testing
    - Provides interactive Swagger docs at /docs

How It Works (Step-by-Step)

1. **Model Initialization**:
    a. Loads the all-MiniLM-L6-v2 sentence transformer for embeddings.
    b. Loads the mistralai/Mistral-7B-Instruct-v0.1 for generating answers.
2. **Database Setup**:
    a. Initializes ChromaDB with a persistent collection invoice_embeddings.
    b. Uses sentence embeddings for vector-based search.
3. **Embedding & Storage**:

   a. On upload, reads and decodes file content
   b. Embeds the text using the MiniLM model
   c. Stores both vector and metadata into ChromaDB
4. **Query Processing**:
   a. Accepts a query from the user
   b. Searches for similar documents in ChromaDB
   c. Feeds the result into Mistral model to generate a context-aware answer
5. **API & Hosting**:
   a. Exposes /upload/ and /ask/ endpoints via FastAPI
   b. Sets up ngrok tunnel to expose the server to the internet
   c. Enables testing and demo via auto-generated docs (/docs)

Link: 👉 https://42eb-34-71-196-200.ngrok-free.app/docs

Challenges:

- LLM Selection Dilemma: Initially explored various models to analyze invoice-to-policy compliance. OpenAI models were effective but paid, while free models lacked accuracy and depth for domain-specific analysis.
- Processing Bottlenecks: With 28 invoice PDFs, response times became long. Some models had input limits or failed to handle large batch processing.
- Quota Limitations: Gemini Flash was fast but API quota limits stopped complete batch analysis from being processed in a single go.

Solutions:

- Switched to Gemini Flash which drastically improved performance and cost (free tier) for structured document reasoning.
- Implemented multithreading to parallelize invoice processing, reducing overall execution time.
- Introduced FastAPI with ChromaDB + local Mistral to build a self-hosted RAG pipeline that bypasses external LLM limits while still giving intelligent responses.

Future Enhancements

- Replace Gemini Flash with an on-premise LLM like LLaMA 3 for policy analysis (avoid quotas).
- Add PDF upload support directly in FastAPI (currently it expects .txt).
- Improve UI/UX by integrating with a streamlit or React frontend.
- Add proper authentication and rate-limiting on API endpoints for real-world usage