MVP Extension: Chatbot Integration Specification

Overview: This document specifies how to integrate the existing Scope 3 emissions MVP (invoice ingestion + emissions estimation) with a conversational chatbot interface. The goal is to let a user upload invoices, run the existing pipeline, and then ask natural language questions about results (e.g. hotspots, suppliers, savings opportunities). This spec is written so that a code-generation model (e.g. Codex) can implement the new components.

Assumptions: - The MVP pipeline already exists and can take an invoice file and return a structured JSON result with emissions. - Language model access (e.g. OpenAI/Claude) is available via HTTP API. - Stack: Python backend (FastAPI or Flask) plus optional simple web frontend (React or Streamlit).

High-Level Architecture: 1. Analysis API: - Endpoint to upload invoice(s) and run existing MVP pipeline. - Returns standardized JSON (invoice analysis). 2. Chat API: - Endpoint to send user messages plus a reference to an invoice analysis. - Builds an LLM prompt using stored analysis JSON + user message. - Returns a natural language reply from the LLM. 3. Frontend: - Screen to upload invoice and trigger analysis. - Simple chat UI bound to the chat endpoint for follow-up questions about results. 4. Persistence: - Minimal: in-memory store or simple key-value persistence for invoice analyses keyed by invoice_id.

---------- 1. Data Contract for Analysis Output ----------

The existing MVP should expose its result in a consistent JSON format. Target shape:

{ "invoice_id": "INV-123", "summary": { "total_emissions_kg": 11201.2, "currency": "USD", "total_spend": 13500.0 }, "by_supplier": [{ "supplier": "SteelCo", "emissions_kg": 10000.0, "spend": 10000.0 }, { "supplier": "ShipFast", "emissions_kg": 1201.2, "spend": 3500.0 }], "by_category": [{ "category": "steel", "emissions_kg": 10000.0 }, { "category": "transport", "emissions_kg": 1201.2 }], "hotspots": { "top_supplier": "SteelCo", "top_category": "steel" } }

Implementation requirements: - Existing pipeline function should be wrapped as:

def run_pipeline(file_path: str) -> dict: """Run OCR, parse, categorize, calculate emissions, and return JSON as above."""

- invoice_id: - Can be generated as a UUID string or hash of filename + timestamp. - All numeric fields should be floats in kg CO2e or standard currency units.

---------- 2. Backend: REST API Endpoints ----------

2.1 /analyze_invoice (POST)

Purpose: - Accept a single invoice file, run the MVP pipeline, store the result, and return analysis JSON.

Request: - Content-Type: multipart/form-data - Parameters: - file: uploaded file (PDF/image)

Response (200): - JSON with fields: - invoice_id: string - analysis: JSON object matching the data contract above.

Example:

{ "invoice_id": "INV-123", "analysis": { ... } }

Behavior: - Save uploaded file to a temporary folder. - Call run_pipeline(path) to generate analysis. - Store analysis in a simple store keyed by invoice_id.

Implementation detail: - Use a global dict or a simple file-based store for hackathon:

ANALYSES = {} # dict[str, dict]

ANALYSES[invoice_id] = analysis

2.2 /chat (POST)

Purpose: - Accept a user chat message and an invoice_id. - Retrieve corresponding analysis JSON. - Call the LLM API with a constructed prompt. - Return the assistant's natural language reply.

Request: - Content-Type: application/json - Body:

{ "invoice_id": "INV-123", "message": "Which supplier is my biggest carbon hotspot?" }

Response (200): - JSON:

{ "reply": "SteelCo is your largest carbon hotspot with about 10,000 kg CO2e, mainly due to steel purchases." }

Error Handling: - If invoice_id not found: return 404 with message "Unknown invoice_id". - If message missing or empty: return 400.

---------- 3. Prompt Construction for Chatbot ----------

Define a function:

def build_prompt(user_message: str, analysis: dict) -> str: """Create a text prompt for the LLM using analysis + user message."""

Guidelines: - Provide role, context, and data in a clear, structured format. - Ask model to be concise and avoid making up numbers not in analysis.

Template (example):

SYSTEM_PROMPT = """You are a sustainability assistant that explains supply chain emissions to non-experts. You receive: 1) Structured emissions analysis for an invoice. 2) A user's question.

You must: - Use only the numbers and suppliers in the analysis JSON. - Highlight the biggest carbon hotspots. - Suggest 1–3 concrete ways to reduce emissions, when relevant. - If the question cannot be answered from the analysis, say so and suggest what data is missing. """

def build_prompt(user_message: str, analysis: dict) -> str: return f"""{SYSTEM_PROMPT}

Here is the invoice analysis JSON: {analysis}

User's question: {user_message}

Now answer clearly in a few sentences."""

The resulting prompt string is passed to the LLM completion/chat API.

---------- 4. LLM Client Wrapper ----------

Abstract the LLM into a small client module so it can be swapped.

File: llm_client.py

Functions:

def generate_reply(prompt: str) -> str: """Call the chosen LLM API with the prompt and return the assistant's reply text."""

Implementation details (pseudo):

- Use environment variables for API key and model name. - Handle basic errors and timeouts.

Example skeleton:

import os import requests

API_KEY = os.getenv("LLM_API_KEY") MODEL_NAME = os.getenv("LLM_MODEL_NAME", "gpt-4.1-mini") # or similar

def generate_reply(prompt: str) -> str: # Pseudocode for OpenAI-style API; adjust endpoint as needed response = requests.post("https://api.openai.com/v1/chat/completions", headers={"Authorization": f"Bearer {API_KEY}"}, json={ "model": MODEL_NAME, "messages": [{"role": "user", "content": prompt}] }) data = response.json() # Extract first message text defensively return data["choices"][0]["message"]["content"].strip()

The /chat endpoint should:

- Call build_prompt(user_message, analysis). - Call generate_reply(prompt). - Return {"reply": reply_text}.

---------- 5. Frontend Interaction Flow ----------

Two-stage UX:

Stage 1: Analysis - User visits app. - Sees file upload control and "Analyze invoice" button. - When file is submitted: - Frontend calls POST /analyze_invoice with form-data. - On success, store invoice_id and analysis in frontend state. - Show a summary: total emissions, top supplier, top category.

Stage 2: Chat - Below the summary, show a chat panel: - Chat history list. - Text input box for user messages. - Send button.

On send: - Frontend POSTs to /chat with JSON: { "invoice_id": invoice_id, "message": userMessage } - Display assistant reply under the user's message.

Optional: - Show suggestion buttons like: - "Where are my biggest hotspots?" - "Which supplier should I focus on first?" - "How could I reduce emissions by 20%?"

These buttons can just pre-fill the chat input.

---------- 6. Example Folder Structure ----------

project_root/ src/ mvp/ ocr.py parser.py categorize.py factors.py emissions.py aggregate.py pipeline.py # exposes run_pipeline() api/ server.py # FastAPI/Flask app with /analyze_invoice and /chat llm_client.py prompts.py # contains SYSTEM_PROMPT and build_prompt() storage.py # simple in-memory or file-based storage frontend/ (optional React or Streamlit app) requirements.txt README.md

server.py responsibilities: - Initialize web framework. - Define /analyze_invoice and /chat routes. - Hold or initialize ANALYSES store.

storage.py (optional) can implement:

ANALYSES = {}

def save_analysis(invoice_id: str, analysis: dict) -> None: ... def get_analysis(invoice_id: str) -> dict | None: ...

---------- 7. Minimal Streamlit Chat Integration (Alternative) ----------

If no separate frontend is desired, a Streamlit-only implementation can embed both upload and chat.

High-level logic:

- Use st.file_uploader for invoice. - When "Analyze" is clicked: - Run pipeline and store analysis in st.session_state["analysis"]. - Show summary using st.metric and st.bar_chart. - Use st.chat_input to collect user messages. - On each message: - Build prompt with st.session_state["analysis"]. - Call generate_reply(prompt). - Display messages in chat container.

This avoids separate REST endpoints, but the core prompt and analysis logic remain the same.

---------- 8. Non-Functional Requirements ----------

- Code should be simple and readable, prioritizing hackathon speed. - Basic logging for each request: - invoice_id - actions taken (analyze/chat) - No sensitive data persistence beyond what is necessary for the demo. - Handle errors gracefully: - Missing invoice_id → human-readable error. - LLM API failures → "The assistant is temporarily unavailable; please try again."

---------- 9. Deliverables ----------

Backend: - Fully working /analyze_invoice and /chat endpoints. - Working LLM integration via llm_client.py. - run_pipeline wrapper that adapts existing MVP logic to standard JSON.

Optional Frontend: - Minimal web page or Streamlit app allowing: - Invoice upload + analysis display. - Interactive chat with the sustainability assistant.

This specification is intended to be consumed by a code generation model (e.g. Codex) to scaffold and implement the chatbot integration with the existing emissions MVP.