Fallback\_agent.py

import os

from app\_config import RAG\_DOCS\_PATH

FRAUD\_RULES\_PATH = os.path.join(RAG\_DOCS\_PATH, "fraud\_rules.txt")

def fallback\_explanation(transaction: dict = None) -> str:

"""

Provides a fallback explanation when the RAG agent fails.

Optionally takes the current transaction for rule matching.

"""

try:

if not os.path.exists(FRAUD\_RULES\_PATH):

return "Policy fallback triggered, but fraud\_rules.txt is missing."

with open(FRAUD\_RULES\_PATH, "r") as f:

rules = f.read()

# === Simple rule-based matches ===

if transaction:

amt = float(transaction.get("amt", 0))

category = transaction.get("category", "").lower()

job = transaction.get("job", "").lower()

gender = transaction.get("gender", "").upper()

lat = float(transaction.get("lat", 0))

merch\_lat = float(transaction.get("merch\_lat", 0))

city\_pop = int(transaction.get("city\_pop", 0))

unix\_time = int(transaction.get("unix\_time", 0))

merchant = transaction.get("merchant", "").lower()

# Rule 1

if category in ["misc\_net", "entertainment"] and amt > 200:

return "Transaction flagged due to high amount in a discretionary category (misc\_net/entertainment)."

# Rule 2

if "unknown" in job and city\_pop < 500:

return "Transaction flagged due to missing occupation and small city population."

# Rule 3

if abs(lat - merch\_lat) > 5.0:

return "User and merchant locations are significantly far apart — possible location spoofing."

# Rule 4

if gender == "M" and category == "personal\_care":

return "Unusual pattern: Male customer spending in personal care category."

# Rule 5

if amt > 500:

return "Transaction amount exceeds $500 — flagged for manual review per policy."

# Rule 6

hour = (unix\_time % 86400) // 3600 # Derive hour from unix\_time

if 2 <= hour <= 4 and amt > 100:

return "Transaction occurred between 2–4 AM with high value — suspicious time window."

# Rule 7

if merchant.startswith("fraud\_") and amt > 0:

return "Transaction involves a known blacklisted merchant (prefix 'fraud\_')."

# Fallback default

return "Transaction flagged due to policy-based heuristics. Further investigation required."

except Exception as e:

return f"⚠️ Fallback engine failed: {e}"

fraud\_agent.py  
import joblib

import numpy as np

import os

from app\_config import MODEL\_PATH, ENCODER\_PATHS

from agentscope.trace\_logger import log\_trace

# === Load model and encoders once ===

if not os.path.exists(MODEL\_PATH):

raise FileNotFoundError(f"Model not found at {MODEL\_PATH}")

model = joblib.load(MODEL\_PATH)

encoders = {}

for feature, path in ENCODER\_PATHS.items():

if not os.path.exists(path):

raise FileNotFoundError(f"Encoder for {feature} not found at {path}")

encoders[feature] = joblib.load(path)

# === Helper: Safe transform for unseen labels ===

def safe\_transform(encoder, value):

if value in encoder.classes\_:

return encoder.transform([value])[0]

else:

return -1 # unknown value

# === Predict Function ===

def predict\_fraud(transaction: dict) -> int:

try:

# === Encode categorical fields safely ===

merchant = safe\_transform(encoders["merchant"], transaction["merchant"])

category = safe\_transform(encoders["category"], transaction["category"])

gender = safe\_transform(encoders["gender"], transaction["gender"])

job = safe\_transform(encoders["job"], transaction["job"])

# === Build feature vector in correct order ===

features = [

merchant,

category,

float(transaction["amt"]),

gender,

float(transaction["lat"]),

float(transaction["long"]),

int(transaction["city\_pop"]),

job,

int(transaction["unix\_time"]),

float(transaction["merch\_lat"]),

float(transaction["merch\_long"])

]

X = np.array([features])

prediction = model.predict(X)[0]

# === Log the trace ===

log\_trace(

step\_name="Fraud Prediction",

input\_summary=f"merchant={transaction['merchant']}, amt={transaction['amt']}",

output\_summary=f"prediction={prediction}",

method="svc"

)

return int(prediction)

except Exception as e:

raise RuntimeError(f"Prediction failed: {e}")

Narrative\_agent.py  
from llama\_index.core import VectorStoreIndex, SimpleDirectoryReader, StorageContext, load\_index\_from\_storage

from llama\_index.vector\_stores.chroma import ChromaVectorStore

from chromadb.config import Settings

from llama\_index.embeddings.huggingface import HuggingFaceEmbedding

import os

from app\_config import RAG\_DOCS\_PATH, CHROMA\_PERSIST\_DIR

from agents.fallback\_agent import fallback\_explanation

from agentscope.trace\_logger import log\_trace

def generate\_narrative() -> tuple[str, str]:

try:

# === Step 1: Setup embedding + vector store ===

embed\_model = HuggingFaceEmbedding(model\_name="sentence-transformers/all-MiniLM-L6-v2")

if not os.path.exists(CHROMA\_PERSIST\_DIR):

print("🔧 No existing Chroma index found. Creating new index...")

documents = SimpleDirectoryReader(RAG\_DOCS\_PATH).load\_data()

chroma\_store = ChromaVectorStore(

persist\_path=CHROMA\_PERSIST\_DIR,

settings=Settings(anonymized\_telemetry=False)

)

storage\_context = StorageContext.from\_defaults(vector\_store=chroma\_store)

index = VectorStoreIndex.from\_documents(documents, storage\_context=storage\_context, embed\_model=embed\_model)

index.storage\_context.persist(persist\_dir=CHROMA\_PERSIST\_DIR)

else:

print("📥 Loading existing Chroma index...")

chroma\_store = ChromaVectorStore(persist\_path=CHROMA\_PERSIST\_DIR)

storage\_context = StorageContext.from\_defaults(vector\_store=chroma\_store)

index = load\_index\_from\_storage(storage\_context, embed\_model=embed\_model)

# === Step 2: Ask question about why this transaction might be fraudulent ===

query\_engine = index.as\_query\_engine()

query = "Why was this transaction flagged as fraudulent?"

response = query\_engine.query(query)

narrative = str(response)

# === Step 3: Log and return ===

log\_trace(

step\_name="Narrative Generation",

input\_summary=query,

output\_summary=narrative,

method="rag"

)

return narrative, "rag"

except Exception as e:

# === Fallback if LLM or RAG fails ===

fallback = fallback\_explanation()

log\_trace(

step\_name="Narrative Generation (Fallback)",

input\_summary="RAG failed",

output\_summary=fallback,

method="fallback"

)

return fallback, "fallback"

trace\_logger.py  
import sqlite3

import os

from datetime import datetime

from app\_config import TRACE\_DB\_PATH

# === Ensure DB + Table exist ===

def \_init\_trace\_db():

os.makedirs(os.path.dirname(TRACE\_DB\_PATH), exist\_ok=True)

conn = sqlite3.connect(TRACE\_DB\_PATH)

cursor = conn.cursor()

cursor.execute("""

CREATE TABLE IF NOT EXISTS trace\_log (

id INTEGER PRIMARY KEY AUTOINCREMENT,

step\_name TEXT,

input\_summary TEXT,

output\_summary TEXT,

method TEXT,

timestamp TEXT

);

""")

conn.commit()

conn.close()

# === Log one agent step ===

def log\_trace(step\_name: str, input\_summary: str, output\_summary: str, method: str):

\_init\_trace\_db()

conn = sqlite3.connect(TRACE\_DB\_PATH)

cursor = conn.cursor()

timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

cursor.execute("""

INSERT INTO trace\_log (step\_name, input\_summary, output\_summary, method, timestamp)

VALUES (?, ?, ?, ?, ?);

""", (step\_name, input\_summary, output\_summary, method, timestamp))

conn.commit()

conn.close()

trace\_reader.py  
import sqlite3

import os

from app\_config import TRACE\_DB\_PATH

def read\_trace\_log(limit=10):

if not os.path.exists(TRACE\_DB\_PATH):

return "⚠️ Trace database not found."

try:

conn = sqlite3.connect(TRACE\_DB\_PATH)

cursor = conn.cursor()

cursor.execute("""

CREATE TABLE IF NOT EXISTS trace\_log (

id INTEGER PRIMARY KEY AUTOINCREMENT,

step\_name TEXT,

input\_summary TEXT,

output\_summary TEXT,

method TEXT,

timestamp TEXT

);

""")

cursor.execute("""

SELECT step\_name, input\_summary, output\_summary, method, timestamp

FROM trace\_log

ORDER BY id DESC

LIMIT ?;

""", (limit,))

rows = cursor.fetchall()

conn.close()

if not rows:

return "ℹ️ No traces logged yet."

result = ""

for i, row in enumerate(rows, 1):

step, inp, out, method, ts = row

result += (

f"\n🧩 Step {i} — {step}\n"

f"📥 Input: {inp}\n"

f"📤 Output: {out}\n"

f"⚙️ Method: {method} | ⏱ {ts}\n"

f"{'-'\*40}\n"

)

return result.strip()

except Exception as e:

return f"❌ Error reading trace log: {e}"

main.py  
from fastapi import FastAPI, HTTPException

from fastapi.middleware.cors import CORSMiddleware

from app.model import TransactionInput, PredictionResponse, NarrativeResponse

from app\_config import DEBUG

from agents.fraud\_agent import predict\_fraud

from agents.narrative\_agent import generate\_narrative

from agentscope.trace\_reader import read\_trace\_log

app = FastAPI(

title="FinGuard Agents API",

description="Multi-agent fraud detection and explainability API",

version="1.0"

)

# === CORS (for frontend) ===

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"], # Allow local Gradio

allow\_credentials=True,

allow\_methods=["\*"],

allow\_headers=["\*"],

)

# === Root ===

@app.get("/")

def root():

return {"status": "FinGuard Agents API running."}

# === Predict Endpoint ===

@app.post("/predict", response\_model=PredictionResponse)

def predict(transaction: TransactionInput):

try:

result = predict\_fraud(transaction.dict())

return {"prediction": result}

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Prediction failed: {e}")

# === Narrative Generation Endpoint ===

@app.post("/narrate", response\_model=NarrativeResponse)

def narrate():

try:

narrative, method = generate\_narrative()

return {"narrative": narrative, "method": method}

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Narration failed: {e}")

# === Trace Log Endpoint ===

@app.post("/trace")

def trace():

try:

log = read\_trace\_log()

return log

except Exception as e:

raise HTTPException(status\_code=500, detail=f"Trace log failed: {e}")

# === Debug Mode Info ===

if DEBUG:

print("✅ FinGuard API is running in DEBUG mode")

model.py  
from pydantic import BaseModel, Field

from typing import Optional

# === Input Schema ===

class TransactionInput(BaseModel):

merchant: str = Field(..., example="fraud\_Hoppe-Parisian")

category: str = Field(..., example="personal\_care")

amt: float = Field(..., example=111.84)

gender: str = Field(..., example="M")

lat: float = Field(..., example=29.0393)

long: float = Field(..., example=-95.4401)

city\_pop: int = Field(..., example=28739)

job: str = Field(..., example="Futures trader")

unix\_time: int = Field(..., example=1388534349)

merch\_lat: float = Field(..., example=29.661049)

merch\_long: float = Field(..., example=-96.186633)

# === Output Schema ===

class PredictionResponse(BaseModel):

prediction: int # 0 = not fraud, 1 = fraud

class NarrativeResponse(BaseModel):

narrative: str

method: str # "rag" or "fallback"

class TraceLogEntry(BaseModel):

step: str

input\_summary: str

output\_summary: str

method: str

timestamp: str

class TraceResponse(BaseModel):

trace: list[TraceLogEntry]

trainmodel.py  
import pandas as pd

import numpy as np

import joblib

import os

from sklearn.preprocessing import LabelEncoder

from sklearn.ensemble import RandomForestClassifier # ✅ Fast, scalable

from app\_config import MODEL\_PATH, ENCODER\_PATHS

print("🔄 Loading dataset...")

df = pd.read\_csv("credit\_card\_transactions.csv") # or your actual path

print(f"✅ Dataset loaded: {df.shape[0]} rows")

# === Label encode all categorical columns ===

encoders = {}

for col in ["merchant", "category", "gender", "job"]:

le = LabelEncoder()

df[col] = le.fit\_transform(df[col])

encoders[col] = le

print(f"✅ Encoded column: {col}")

joblib.dump(le, ENCODER\_PATHS[col])

# === Prepare features and target ===

X = df[["merchant", "category", "amt", "gender", "lat", "long",

"city\_pop", "job", "unix\_time", "merch\_lat", "merch\_long"]]

y = df["is\_fraud"]

print("🚀 Training RandomForest model...")

model = RandomForestClassifier(n\_estimators=100, max\_depth=10, random\_state=42)

model.fit(X, y)

print("✅ Model training complete.")

# === Save model ===

joblib.dump(model, MODEL\_PATH)

print(f"✅ Saved model to: {MODEL\_PATH}")

fraud\_rules.txt  
Fallback Fraud Detection Rules

==============================

The following rules are used to generate an explanation when AI-based narrative generation fails or is unavailable. These are based on internal policy, historical fraud patterns, and compliance heuristics.

Rules:

------

1. If `category` is "misc\_net" or "entertainment" AND `amt` > 200.00 → HIGH RISK.

Explanation: High-value transactions in discretionary categories are often exploited by fraudsters.

2. If `job` is unknown AND `city\_pop` < 500 → MEDIUM RISK.

Explanation: Sparse population + no identifiable occupation may indicate a spoofed identity.

3. If `lat` and `merch\_lat` differ by more than 5.0 degrees → LOCATION MISMATCH.

Explanation: Large geospatial discrepancy between user and merchant location.

4. If `gender` is "M" AND `category` is "personal\_care" → UNUSUAL SPENDING PATTERN.

Explanation: Demographic mismatch with transaction category suggests anomaly.

5. If `amt` > 500.00 → FLAG IMMEDIATELY.

Explanation: Any transaction above $500 must undergo manual review regardless of other factors.

6. If `unix\_time` is between 2 AM – 4 AM AND `amt` > 100 → SUSPICIOUS TIME WINDOW.

Explanation: High-value late-night transactions often correlate with account takeover.

7. If merchant name includes "fraud\_" prefix AND amount > 0 → POTENTIAL FRAUD.

Explanation: Internal merchant blacklist triggered. Investigation required.

Usage Notes:

------------

- These rules are purely heuristic and may produce false positives.

- Always use them as fallback explanations when RAG agent cannot construct a meaningful rationale.

- You may update these rules as fraud behavior evolves or regulatory policies change.

Schema.py  
from sqlalchemy import Column, Integer, Float, String, DateTime, create\_engine

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker

import datetime

# Base class for SQLAlchemy models

Base = declarative\_base()

# === Transaction Record (Optional - for storing inputs) ===

class TransactionRecord(Base):

\_\_tablename\_\_ = "transactions"

id = Column(Integer, primary\_key=True, index=True)

merchant = Column(String)

category = Column(String)

amt = Column(Float)

gender = Column(String)

lat = Column(Float)

long = Column(Float)

city\_pop = Column(Integer)

job = Column(String)

unix\_time = Column(Integer)

merch\_lat = Column(Float)

merch\_long = Column(Float)

is\_fraud = Column(Integer)

created\_at = Column(DateTime, default=datetime.datetime.utcnow)

# === Agent Trace Log (Optional - for custom trace logs) ===

class AgentTrace(Base):

\_\_tablename\_\_ = "agent\_traces"

id = Column(Integer, primary\_key=True, index=True)

step\_name = Column(String)

input\_summary = Column(String)

output\_summary = Column(String)

method = Column(String) # e.g., "prediction", "RAG", "fallback"

timestamp = Column(DateTime, default=datetime.datetime.utcnow)

# === Setup SQLite Engine (Only if you use this DB explicitly) ===

SQLITE\_PATH = "db/db.sqlite3"

engine = create\_engine(f"sqlite:///{SQLITE\_PATH}")

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

def init\_db():

Base.metadata.create\_all(bind=engine)

Ui.py  
import gradio as gr

import requests

API\_BASE = "http://localhost:8000"

def build\_ui():

with gr.Blocks(title="FinGuard Agents - Real-Time Fraud Detection") as demo:

gr.Markdown("## 🔍 FinGuard Agents")

gr.Markdown("Multi-agent fraud detection with real-time prediction, explanation, and traceability.")

with gr.Row():

merchant = gr.Textbox(label="Merchant")

category = gr.Dropdown(label="Category", choices=[

"misc\_net", "grocery\_pos", "entertainment", "gas\_transport", "misc\_pos", "food\_dining",

"personal\_care", "health\_fitness", "travel", "kids\_pets"

])

amt = gr.Number(label="Transaction Amount")

with gr.Row():

gender = gr.Dropdown(label="Gender", choices=["M", "F"])

job = gr.Textbox(label="Job Title")

city\_pop = gr.Number(label="City Population")

with gr.Row():

lat = gr.Number(label="User Latitude")

long = gr.Number(label="User Longitude")

unix\_time = gr.Number(label="Transaction Unix Time (int)")

with gr.Row():

merch\_lat = gr.Number(label="Merchant Latitude")

merch\_long = gr.Number(label="Merchant Longitude")

submit\_btn = gr.Button("🚦 Predict Fraud")

explain\_btn = gr.Button("🧠 Explain Decision")

trace\_btn = gr.Button("📜 Show Trace")

output\_pred = gr.Label(label="Fraud Prediction")

output\_narrative = gr.Textbox(label="Explanation", lines=3)

output\_trace = gr.Textbox(label="Trace Log", lines=5)

# === HANDLERS ===

def on\_submit(merchant, category, amt, gender, job, city\_pop, lat, long, unix\_time, merch\_lat, merch\_long):

payload = {

"merchant": merchant,

"category": category,

"amt": amt,

"gender": gender,

"job": job,

"city\_pop": city\_pop,

"lat": lat,

"long": long,

"unix\_time": int(unix\_time),

"merch\_lat": merch\_lat,

"merch\_long": merch\_long

}

try:

res = requests.post(f"{API\_BASE}/predict", json=payload)

return "✅ Fraud" if res.json()["prediction"] == 1 else "❎ Not Fraud"

except Exception as e:

return f"❌ Error: {e}"

def on\_explain():

try:

res = requests.post(f"{API\_BASE}/narrate")

data = res.json()

return f"[{data.get('method', 'N/A').upper()}] {data.get('narrative', '')}"

except Exception as e:

return f"Explanation error: {e}"

def on\_trace():

try:

res = requests.post(f"{API\_BASE}/trace")

return res.text

except Exception as e:

return f"Trace error: {e}"

submit\_btn.click(

fn=on\_submit,

inputs=[merchant, category, amt, gender, job, city\_pop, lat, long, unix\_time, merch\_lat, merch\_long],

outputs=output\_pred

)

explain\_btn.click(fn=on\_explain, outputs=output\_narrative)

trace\_btn.click(fn=on\_trace, outputs=output\_trace)

return demo

if \_\_name\_\_ == "\_\_main\_\_":

demo = build\_ui()

demo.launch()

app\_config.py  
import os

# === MODEL & ENCODER PATHS ===

MODEL\_PATH = os.path.join("agents", "fraud\_model.pkl")

ENCODER\_PATHS = {

"merchant": os.path.join("encoders", "merchant.pkl"),

"category": os.path.join("encoders", "category.pkl"),

"gender": os.path.join("encoders", "gender.pkl"),

"job": os.path.join("encoders", "job.pkl"),

}

# === RAG CONFIG ===

RAG\_DOCS\_PATH = os.path.join("core", "rag\_docs")

RAG\_DOC\_FILES = [

os.path.join(RAG\_DOCS\_PATH, "aml\_guidelines.md"),

os.path.join(RAG\_DOCS\_PATH, "fraud\_rules.txt")

]

CHROMA\_PERSIST\_DIR = os.path.join("core", "rag\_docs", "chroma\_index")

# === TRACEABILITY ===

TRACE\_DB\_PATH = os.path.join("agentscope", "agentscope.db")

# === DATABASES (if using db/db.sqlite3 for examples) ===

EXAMPLE\_DB\_PATH = os.path.join("db", "db.sqlite3")

# === FRONTEND CONFIG ===

UI\_TITLE = "FinGuard Agents - Real-time Fraud Detection"

UI\_DESCRIPTION = "A multi-agent system that predicts fraud, explains decisions, and shows full traceability."

# === MISC ===

DEBUG = True

Issues….  
  
  
What ever the input may be it is giving 0 as output…  
  
next, the agent scope is completely mis understood. The actual agent scope  
 **Trace every step** of agent execution (decision points, checks, subtools).

 **Support any agent type** (fraud, legal, policy-based, narrative, etc.).

 **Log and visualize**:

* 🧠 Intent classification
* 📜 Rule/policy evaluation
* 🔁 Intermediate reasoning (confidence, fallback triggers)
* ⚙️ Tool usage and LLM prompts
* 🚨 Human handoff triggers

 **Stream live trace steps** via WebSockets or exposed APIs.

 **Allow devs to debug agents like a step debugger**, not just summary logs.

so, We'll keep the current trace format (the summary view like “🧩 Step 1 — Fraud Prediction...”) as a **lightweight, readable log for frontend users**.

🔍 Separately, we'll build **AgentScope-style introspection** for agent workflows — capturing every internal decision, rule, fallback, and confidence step during agent execution. It will be modular and general-purpose   
  
  
Without changing original decision code folder structure…