# LangGraph Orchestrator with Data Preprocessing

import asyncio  
import pandas as pd  
from langchain.chains import GraphChain  
from typing import Dict, Any  
  
  
# Helper function to read and preprocess Excel data  
async def read\_and\_preprocess(file\_path, sheet\_name, date\_columns=None, text\_columns=None):  
 """Read and preprocess data: standardize text, dates, and numeric values."""  
 df = pd.read\_excel(file\_path, sheet\_name=sheet\_name)  
  
 # Convert date columns to consistent format  
 if date\_columns:  
 for col in date\_columns:  
 df[col] = pd.to\_datetime(df[col], errors='coerce').dt.strftime('%Y-%m-%d')  
  
 # Trim whitespaces and standardize case for text columns  
 if text\_columns:  
 for col in text\_columns:  
 df[col] = df[col].astype(str).str.strip().str.lower()  
  
 # Ensure all numeric columns are floats  
 for col in df.select\_dtypes(include='number').columns:  
 df[col] = df[col].astype(float)  
  
 return df  
  
  
# Helper function to parse summarized text file  
def parse\_summary\_file(file\_path: str) -> Dict[str, Any]:  
 """Parse a manually summarized text file to extract key claim details."""  
 details = {}  
 with open(file\_path, "r") as file:  
 for line in file:  
 if ":" in line:  
 key, value = line.split(":", 1)  
 details[key.strip()] = value.strip()  
 return {  
 "claim\_id": details.get("Claim Number", "").replace("\*", "").strip(),  
 "service\_date\_from": pd.to\_datetime(details.get("Service date from", "").strip()).strftime('%Y-%m-%d'),  
 "service\_date\_to": pd.to\_datetime(details.get("Service date to", "").strip()).strftime('%Y-%m-%d'),  
 "total\_charged": float(details.get("Total Charged amount", 0)),  
 "allowed\_amount": float(details.get("Total allowed amount", 0)),  
 "patient\_responsibility": float(details.get("Patients Responsibility", 0)),  
 "elevance\_responsibility": float(details.get("Elevance Responsibility", 0)),  
 "elevance\_paid": float(details.get("Total Elevance paid", 0)),  
 "provider\_name": details.get("Provider Name", "").strip().lower(),  
 "member\_name": details.get("Member Name", "").strip().lower(),  
 }  
  
  
# Agent Nodes as GraphChain Nodes  
class MemberEligibilityNode(GraphChain):  
 async def run(self, inputs: Dict[str, Any]):  
 membership\_df = await read\_and\_preprocess(  
 membership\_file,  
 "Membership file",  
 date\_columns=["effective\_date", "termination\_date"],  
 text\_columns=["member\_name"]  
 )  
 match = membership\_df[  
 (membership\_df["member\_name"] == inputs["member\_name"])  
 & (membership\_df["effective\_date"] <= inputs["service\_date\_from"])  
 & (membership\_df["termination\_date"] >= inputs["service\_date\_to"])  
 ]  
 return {"eligible": not match.empty, "reason": "Member eligibility failed" if match.empty else "Success"}  
  
  
class ProviderEligibilityNode(GraphChain):  
 async def run(self, inputs: Dict[str, Any]):  
 provider\_df = await read\_and\_preprocess(  
 provider\_file,  
 "Provider file",  
 date\_columns=["Contract\_from", "Contract\_to"],  
 text\_columns=["provider\_name"]  
 )  
 match = provider\_df[  
 (provider\_df["provider\_name"] == inputs["provider\_name"])  
 & (provider\_df["Contract\_from"] <= inputs["service\_date\_from"])  
 & (provider\_df["Contract\_to"] >= inputs["service\_date\_to"])  
 ]  
 return {"eligible": not match.empty, "reason": "Provider eligibility failed" if match.empty else "Success"}  
  
  
class ProviderNetworkNode(GraphChain):  
 async def run(self, inputs: Dict[str, Any]):  
 network\_df = await read\_and\_preprocess(  
 provider\_network\_file,  
 "Provider network file",  
 text\_columns=["provider\_name", "Provider network status"]  
 )  
 match = network\_df[  
 (network\_df["provider\_name"] == inputs["provider\_name"])  
 & (network\_df["Provider network status"] == "active")  
 ]  
 return {"eligible": not match.empty, "reason": "Provider network inactive" if match.empty else "Success"}  
  
  
class DecisionNode(GraphChain):  
 async def run(self, inputs: Dict[str, Any]):  
 claims\_df = await read\_and\_preprocess(  
 claims\_file,  
 "Claims",  
 date\_columns=["service\_date\_from", "service\_date\_to"],  
 text\_columns=[]  
 )  
 claim\_row = claims\_df[claims\_df["DCN"] == inputs["claim\_id"]]  
  
 if claim\_row.empty:  
 return {"decision": "Failed", "reason": "Claim not found"}  
   
 row = claim\_row.iloc[0]  
 checks\_passed = (  
 row["Total\_charged\_amt"] == inputs["total\_charged"]  
 and row["Allowed\_amount"] == inputs["allowed\_amount"]  
 and row["patient\_responsibility"] == inputs["patient\_responsibility"]  
 )  
 if checks\_passed and row["Elevance Paid"] < inputs["elevance\_responsibility"]:  
 return {"decision": "Adjust", "reason": "Adjustment Required"}  
 return {"decision": "No Adjust", "reason": "No Adjustment Needed"}  
  
  
class AdjustmentNode(GraphChain):  
 async def run(self, inputs: Dict[str, Any]):  
 claims\_df = pd.read\_excel(claims\_file, "Claims")  
 new\_row = {  
 "DCN": inputs["claim\_id"],  
 "Item code": 84,  
 "service\_date\_from": inputs["service\_date\_from"],  
 "service\_date\_to": inputs["service\_date\_to"],  
 "Total\_charged\_amt": inputs["total\_charged"],  
 "Allowed\_amount": inputs["allowed\_amount"],  
 "patient\_responsibility": inputs["patient\_responsibility"],  
 "Elevance responsibility": inputs["elevance\_responsibility"],  
 "Elevance Paid": inputs["elevance\_responsibility"],  
 }  
 claims\_df = claims\_df.append(new\_row, ignore\_index=True)  
 claims\_df.to\_excel(claims\_file, sheet\_name="Claims", index=False)  
 return {"status": "Adjustment Created", "claim\_id": inputs["claim\_id"]}  
  
  
# File Paths  
claims\_file = "claims\_file.xlsx"  
provider\_file = "provider\_file.xlsx"  
provider\_network\_file = "provider\_network.xlsx"  
membership\_file = "membership\_file.xlsx"  
summary\_file = "iqt\_summary.txt"  
  
  
# Workflow Orchestration using GraphChain  
async def orchestrate\_workflow():  
 # Parse summary file  
 inputs = parse\_summary\_file(summary\_file)  
  
 # Define nodes  
 member\_node = MemberEligibilityNode(name="Member Eligibility")  
 provider\_node = ProviderEligibilityNode(name="Provider Eligibility")  
 network\_node = ProviderNetworkNode(name="Provider Network")  
 decision\_node = DecisionNode(name="Decision")  
 adjustment\_node = AdjustmentNode(name="Adjustment")  
  
 # Connect nodes  
 decision\_node.add\_parent(member\_node)  
 decision\_node.add\_parent(provider\_node)  
 decision\_node.add\_parent(network\_node)  
 adjustment\_node.add\_parent(decision\_node)  
  
 # Run GraphChain workflow  
 workflow = GraphChain(root=member\_node)  
 result = await workflow.run(inputs=inputs)  
 print("Workflow Execution Result:", result)  
  
  
# Run the workflow  
asyncio.run(orchestrate\_workflow())