# LangGraph StateGraph-Based Workflow Code with Updates

## Overview

This updated code includes the following enhancements:  
1. Entry point added to the StateGraph.  
2. Asynchronous execution using `ainvoke`.  
3. Data preprocessing for all files to ensure consistent datatypes for comparison.  
4. Updated decision logic to handle robust checks and avoid errors during key lookups.

## Imports and File Handling

import asyncio  
import pandas as pd  
from langgraph.graph import StateGraph  
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(),  
 }

## StateGraph Nodes

async def member\_eligibility(inputs: Dict[str, Any]):  
 df = await read\_and\_preprocess(membership\_file, "Membership file",   
 date\_columns=["effective\_date", "termination\_date"],   
 text\_columns=["member\_name"])  
 match = df[  
 (df["member\_name"] == inputs["member\_name"]) &  
 (df["effective\_date"] <= inputs["service\_date\_from"]) &  
 (df["termination\_date"] >= inputs["service\_date\_to"])  
 ]  
 return {"eligible": not match.empty}  
  
async def provider\_eligibility(inputs: Dict[str, Any]):  
 df = await read\_and\_preprocess(provider\_file, "Provider file",   
 date\_columns=["Contract\_from", "Contract\_to"],   
 text\_columns=["provider\_name"])  
 match = df[  
 (df["provider\_name"] == inputs["provider\_name"]) &  
 (df["Contract\_from"] <= inputs["service\_date\_from"]) &  
 (df["Contract\_to"] >= inputs["service\_date\_to"])  
 ]  
 return {"eligible": not match.empty}  
  
async def provider\_network(inputs: Dict[str, Any]):  
 df = await read\_and\_preprocess(provider\_network\_file, "Provider network file",   
 text\_columns=["provider\_name", "Provider network status"])  
 match = df[  
 (df["provider\_name"] == inputs["provider\_name"]) &  
 (df["Provider network status"] == "active")  
 ]  
 return {"eligible": not match.empty}  
  
async def decision\_logic(inputs: Dict[str, Any]):  
 df = await read\_and\_preprocess(claims\_file, "Claims")  
 df.columns = df.columns.str.strip().str.lower()  
 row = df[df["dcn"] == inputs["claim\_id"]]  
 if row.empty:  
 return {"decision": "no\_match"}  
 row = row.iloc[0]  
 checks = (  
 row["total\_charged\_amt"] == inputs["total\_charged"] and  
 row["allowed\_amount"] == inputs["allowed\_amount"] and  
 row["patient\_responsibility"] == inputs["patient\_responsibility"]  
 )  
 if checks and row["elevance paid"] < inputs["elevance\_responsibility"]:  
 return {"decision": "adjust"}  
 return {"decision": "no\_adjust"}  
  
async def adjustment\_creation(inputs: Dict[str, Any]):  
 df = pd.read\_excel(claims\_file, "Claims")  
 new\_row = {  
 "dcn": inputs["claim\_id"],  
 "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"],  
 }  
 df = df.append(new\_row, ignore\_index=True)  
 df.to\_excel(claims\_file, sheet\_name="Claims", index=False)  
 return {"status": "adjustment\_created"}

## StateGraph Workflow

# 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 Definition Using StateGraph  
workflow = StateGraph(dict)  
workflow.add\_node("member\_eligibility", member\_eligibility)  
workflow.add\_node("provider\_eligibility", provider\_eligibility)  
workflow.add\_node("provider\_network", provider\_network)  
workflow.add\_node("decision\_logic", decision\_logic)  
workflow.add\_node("adjustment\_creation", adjustment\_creation)  
  
# Define dependencies  
workflow.add\_edge("member\_eligibility", "decision\_logic")  
workflow.add\_edge("provider\_eligibility", "decision\_logic")  
workflow.add\_edge("provider\_network", "decision\_logic")  
workflow.add\_edge("decision\_logic", "adjustment\_creation")  
  
# Set entry point  
workflow.set\_entry\_point("member\_eligibility")  
  
# Compile the graph  
app = workflow.compile()  
  
# Orchestrate the Workflow  
async def orchestrate\_workflow():  
 inputs = parse\_summary\_file(summary\_file)  
 result = await app.ainvoke(inputs)  
 print("Workflow Execution Result:", result)  
  
asyncio.run(orchestrate\_workflow())