# LangGraph Orchestrator with Async Implementation (Final and Verified)

from langchain.chains import GraphChain  
from typing import Any, Dict  
import pandas as pd  
import asyncio  
  
# Helper function to read Excel data  
async def read\_excel(file\_path, sheet\_name):  
 """Read data from an Excel file asynchronously."""  
 return pd.read\_excel(file\_path, sheet\_name=sheet\_name)  
  
  
# Individual Eligibility Check Agent Classes  
class ProviderDetailsCheckAgent(GraphChain):  
 async def run(self, inputs: Dict[str, Any]):  
 """Check provider details asynchronously."""  
 df = await read\_excel(excel\_file, sheet\_name="Eligibility")  
 result = df[df["ProviderID"] == inputs["provider\_id"]]  
 if not result.empty:  
 return {"eligible": True, "details": result.to\_dict(orient="records")}  
 return {"eligible": False}  
  
  
class ProviderNetworkCheckAgent(GraphChain):  
 async def run(self, inputs: Dict[str, Any]):  
 """Check provider network asynchronously."""  
 df = await read\_excel(excel\_file, sheet\_name="Eligibility")  
 result = df[df["NetworkID"] == inputs["provider\_id"]]  
 if not result.empty:  
 return {"eligible": True, "details": result.to\_dict(orient="records")}  
 return {"eligible": False}  
  
  
class MemberEligibilityCheckAgent(GraphChain):  
 async def run(self, inputs: Dict[str, Any]):  
 """Check member eligibility asynchronously."""  
 df = await read\_excel(excel\_file, sheet\_name="Eligibility")  
 result = df[df["MemberID"] == inputs["member\_id"]]  
 if not result.empty:  
 return {"eligible": True, "details": result.to\_dict(orient="records")}  
 return {"eligible": False}  
  
  
class DecisionAgent(GraphChain):  
 async def run(self, inputs: Dict[str, Any]):  
 """Standalone pricing check asynchronously."""  
 df = await read\_excel(excel\_file, sheet\_name="Pricing")  
 result = df[df["ClaimID"] == inputs["claim\_id"]]  
 if not result.empty and result.iloc[0]["Price"] > 1000: # Assume $1000 threshold  
 return {"adjustment\_needed": True, "price": result.iloc[0]["Price"]}  
 return {"adjustment\_needed": False}  
  
  
class AdjustmentInitiationAgent(GraphChain):  
 async def run(self, inputs: Dict[str, Any]):  
 """Mock adjustment process asynchronously."""  
 print(f"Initiating adjustment for Claim ID: {inputs['claim\_id']}")  
 await asyncio.sleep(1) # Simulate delay  
 return {"status": "Adjustment Completed"}  
  
  
# Define the workflow graph  
provider\_details\_node = ProviderDetailsCheckAgent(name="Provider Details Check")  
provider\_network\_node = ProviderNetworkCheckAgent(name="Provider Network Check")  
member\_eligibility\_node = MemberEligibilityCheckAgent(name="Member Eligibility Check")  
decision\_node = DecisionAgent(name="Pricing Check")  
adjustment\_node = AdjustmentInitiationAgent(name="Adjustment Initiation")  
  
# Connect nodes  
decision\_node.add\_parent(provider\_details\_node)  
decision\_node.add\_parent(provider\_network\_node)  
decision\_node.add\_parent(member\_eligibility\_node)  
adjustment\_node.add\_parent(decision\_node)  
  
# Define the root node for the workflow  
root\_node = provider\_details\_node  
  
# Execute the workflow  
workflow = GraphChain(root=root\_node)  
output = asyncio.run(workflow.run(inputs={"claim\_id": "C789", "provider\_id": "P123", "member\_id": "M456"}))  
print(output)