Implementation Guide: Multi-Agent Workflow System for Nexhelm

Project Overview

Build an agentic workflow automation system using LangGraph where two AI agents (Advisor Agent and Operations Agent) collaborate to complete financial workflows like opening IRA accounts.

Project Structure

```
nexhelm_workflow/
├── src/
│   ├── state.py            # State schema definition
│   ├── storage.py          # Simulated document store
│   ├── nodes/
│   │   ├── __init__.py
│   │   ├── orchestrator.py   # Initial planning node
│   │   ├── supervisor.py     # Routing logic
│   │   ├── advisor.py        # Advisor agent node
│   │   └── operations.py     # Operations agent node
│   ├── tools/
│   │   ├── __init__.py
│   │   ├── advisor_tools.py  # Advisor agent tools
│   │   └── operations_tools.py # Operations agent tools
│   ├── graph.py            # Graph construction
│   └── main.py             # Entry point
├── tests/
│   └── test_workflow.py      # Test cases
├── requirements.txt
└── README.md
```

STEP 1: Define State Schema

File: `src/state.py`

Requirements:

- Create TypedDict class named `WorkflowState`
- Include all fields needed to track workflow progress
- Use proper type hints (List, Dict, Optional)

State Fields:

WorkflowState:
- request: dict
  - type: str (e.g., "open_roth_ira")
  - client_id: str
  - client_name: str
  - initiator: str (advisor name)

- workflow_id: str (UUID format)

- status: str
  - Allowed values: "pending", "in_progress", "completed", "failed"

- created_at: str (ISO timestamp)
- updated_at: str (ISO timestamp)

- context: dict
  - client_age: int
  - existing_accounts: List[str]
  - available_documents: List[str]

- tasks: List[dict]
  Each task dict contains:
    - id: str
    - description: str
    - owner: str ("advisor_agent" or "operations_agent")
    - status: str ("pending", "in_progress", "completed", "failed")
    - dependencies: List[str] (list of task IDs)
    - result: Optional[str]

- messages: List[dict]
  Each message dict contains:
    - from_agent: str
    - to_agent: str
    - timestamp: str (ISO format)
    - content: str
    - type: str ("question", "response", "update")

- decisions: List[dict]
  Each decision dict contains:
    - agent: str
    - timestamp: str
    - decision: str
    - reasoning: str

- blockers: List[dict]
  Each blocker dict contains:

```
    - id: str
    - description: str
    - identified_by: str
    - assigned_to: str
    - status: str ("active", "resolved")
    - created_at: str

- next_actions: List[dict]
  Each action dict contains:
    - agent: str
    - action: str
    - priority: str ("low", "medium", "high")

- outcome: Optional[dict]
```

Validation:

- Ensure proper imports: `from typing import TypedDict, List, Dict, Optional`
- Export WorkflowState class

STEP 2: Create Simulated Document Store

File: `src/storage.py`

Requirements:

- Create in-memory data storage using Python dicts
- Simulate client database, document storage, and account system
- Provide methods to read/write data

Classes to Implement:

Class 1: `SimulatedDocumentStore`

Attributes:

- `documents: dict` - Stores documents by client_id

Methods:

- `__init__()` - Initialize with sample data for "john_smith_123"
- `get_document(client_id: str, doc_type: str) -> Optional[dict]`
- `update_document(client_id: str, doc_type: str, data: dict) -> bool`
- `list_documents(client_id: str) -> List[str]`

Sample Data:

```
Client: john_smith_123
Documents:
  - drivers_license: {status: "valid", uploaded: True, verified: True}
  - tax_return_2023: {status: "valid", income: 145000, year: 2023}
  - ira_application: {status: "pending", signature_page3: False, submitted: False}
```

Class 2: `SimulatedCRM`

Attributes:

- `clients: dict` - Stores client information

Methods:

- `__init__()` - Initialize with sample client "john_smith_123"
- `get_client(client_id: str) -> Optional[dict]`
- `update_client(client_id: str, field: str, value: any) -> bool`

Sample Data:

```
Client: john_smith_123
  - name: "John Smith"
  - age: 45
  - email: "john@example.com"
  - existing_accounts: ["checking", "brokerage"]
  - income: 145000
```

Class 3: `SimulatedAccountSystem`

Attributes:

- `accounts: dict` - Stores opened accounts
- `counter: int` - Account number generator (start at 1000)

Methods:

- `__init__()`
- `open_account(client_id: str, account_type: str) -> dict` - Returns {account_number, status, created_at}
- `get_account(account_number: str) -> Optional[dict]`

Module-level instances:

```
python
```

```
doc_store = SimulatedDocumentStore()
crm = SimulatedCRM()
account_system = SimulatedAccountSystem()
```

STEP 3: Build Orchestrator Node (Hardcoded)

File: `src/nodes/orchestrator.py`

Requirements:

- Function that takes WorkflowState as input
- Returns updated WorkflowState with hardcoded tasks
- No LLM - just static task generation for "Open IRA" workflow

Function Signature:

```
python

def orchestrator_node(state: WorkflowState) -> WorkflowState
```

Logic:

1. Generate workflow_id using uuid4
2. Set status to "in_progress"
3. Set timestamps (created_at, updated_at) using datetime.now()
4. Load client context from CRM (import from storage.py)
5. Create hardcoded task list (see below)
6. Set next_actions to [{"agent": "operations_agent", "action": "verify_eligibility", "priority": "high"}]
7. Return updated state

Hardcoded Task List for "Open Roth IRA":

```
python
```

```
tasks = [
    {
        "id": "task_1",
        "description": "Verify Roth IRA income eligibility",
        "owner": "operations_agent",
        "status": "pending",
        "dependencies": [],
        "result": None
    },
    {
        "id": "task_2",
        "description": "Send IRA application form to client",
        "owner": "advisor_agent",
        "status": "pending",
        "dependencies": ["task_1"],
        "result": None
    },
    {
        "id": "task_3",
        "description": "Review and validate submitted IRA application",
        "owner": "operations_agent",
        "status": "pending",
        "dependencies": ["task_2"],
        "result": None
    },
    {
        "id": "task_4",
        "description": "Open Roth IRA account in system",
        "owner": "operations_agent",
        "status": "pending",
        "dependencies": ["task_3"],
        "result": None
    },
    {
        "id": "task_5",
        "description": "Notify client of account opening",
        "owner": "advisor_agent",
        "status": "pending",
        "dependencies": ["task_4"],
        "result": None
    }
]
```

Imports:

- from src.state import WorkflowState

- `from src.storage import crm`
- `import uuid`
- `from datetime import datetime`

STEP 4: Build Supervisor Router

File: `src/nodes/supervisor.py`

Requirements:

- Function that determines which node to execute next
- Rule-based routing (no LLM)
- Returns string indicating next node name

Function Signature:

```
python

def supervisor_node(state: WorkflowState) -> str
```

Routing Logic:

1. **Check completion:**
   - If `state["status"] == "completed"` → return `"end"`
   - If `state["status"] == "failed"` → return `"end"`
2. **Check next_actions:**
   - If `state["next_actions"]` is not empty:
     - Get first action's agent field
     - If agent == "advisor_agent" → return `"advisor_agent"`
     - If agent == "operations_agent" → return `"operations_agent"`
3. **Fallback:**
   - If no next_actions and status is "in_progress":
     - Find first task with status "pending" and no unmet dependencies
     - Return that task's owner
   - Otherwise → return `"end"`

Helper function (optional):

```
python
```

```
def check_task_dependencies(task: dict, all_tasks: List[dict]) -> bool:
    """Returns True if all dependencies are completed"""
```

Imports:

- `from src.state import WorkflowState`
- `from typing import List`

STEP 5: Build Agent Nodes (Hardcoded Logic)

File: `src/nodes/advisor.py`

Requirements:

- Function representing Advisor Agent
- Hardcoded logic (no LLM yet)
- Finds pending tasks assigned to advisor and completes them
- Updates state with results and next actions

Function Signature:

```
python

def advisor_agent_node(state: WorkflowState) -> WorkflowState
```

Logic:

1. Find first task where `owner == "advisor_agent"` and `status == "pending"` and dependencies are met
2. If no task found → set next_actions to empty and return state
3. Process task based on description:

Task: "Send IRA application form to client"

- Print: `"Advisor Agent: Sending IRA application form to {client_name}"`
- Update task status to "completed"
- Set task result to "Form sent via email"
- Add message to state:

```
python
```

```
{
  "from_agent": "advisor_agent",
  "to_agent": "operations_agent",
  "timestamp": datetime.now().isoformat(),
  "content": "IRA application form sent to client. Awaiting submission.",
  "type": "update"
}
```

- Add decision to state:

```
python
{
  "agent": "advisor_agent",
  "timestamp": datetime.now().isoformat(),
  "decision": "Sent personalized IRA application to client",
  "reasoning": "Client is existing customer, pre-filled form with known details"
}
```

- Set next_actions to `[{"agent": "operations_agent", "action": "wait_for_form", "priority": "medium"}]`

Task: "Notify client of account opening"

- Print: `"Advisor Agent: Notifying {client_name} of successful account opening"`
- Update task status to "completed"
- Set task result to "Client notified"
- Add message
- Set status to "completed" (last task)
- Set next_actions to empty

4. Update state["updated_at"]
5. Return state

Imports:

- `from src.state import WorkflowState`
- `from datetime import datetime`

File: `src/nodes/operations.py`

Requirements:

- Function representing Operations Agent

- Hardcoded logic (no LLM yet)
- Finds pending tasks assigned to operations and completes them
- Interacts with simulated storage systems

Function Signature:

```python
def operations_agent_node(state: WorkflowState) -> WorkflowState
```

Logic:

1. Find first task where `owner == "operations_agent"` and `status == "pending"` and dependencies are met
2. If no task found → set next_actions to empty and return state
3. Process task based on description:

Task: "Verify Roth IRA income eligibility"

- Get client from CRM: `client = crm.get_client(state["request"]["client_id"])`
- Get tax return: `doc = doc_store.get_document(client_id, "tax_return_2023")`
- Check if income < 161000
- Print: `"Operations Agent: Verifying eligibility - Income: ${income}"`
- If eligible:
  - Update task status to "completed"
  - Set result to f"Eligible - Income ${income} under $161k limit"
  - Add message to advisor_agent confirming eligibility
  - Add decision
  - Set next_actions to advisor_agent for next task
- If not eligible:
  - Add blocker to state
  - Set task status to "failed"

Task: "Review and validate submitted IRA application"

- Get document: `doc = doc_store.get_document(client_id, "ira_application")`
- Check if signature_page3 is True
- Print: `"Operations Agent: Validating IRA application form"`
- If valid:
  - Update task status to "completed"

- Set result to "Form validated successfully"
- Add message
- Set next_actions to operations_agent (next task)

- If invalid (missing signature):
  - Create blocker:

```python
{
  "id": "blocker_1",
  "description": "IRA application missing signature on page 3",
  "identified_by": "operations_agent",
  "assigned_to": "advisor_agent",
  "status": "active",
  "created_at": datetime.now().isoformat()
}
```

- Add message to advisor_agent requesting signature
- Set next_actions to advisor_agent to resolve blocker
- Keep task as "pending"

Task: "Open Roth IRA account in system"

- Call: `result = account_system.open_account(client_id, "roth_ira")`
- Print: `"Operations Agent: Opening account - {account_number}"`
- Update task status to "completed"
- Set result to f"Account {account_number} opened successfully"
- Add message
- Set next_actions to advisor_agent for final notification
- Store account_number in state["outcome"]

4. Update state["updated_at"]
5. Return state

Imports:

- `from src.state import WorkflowState`
- `from src.storage import crm, doc_store, account_system`
- `from datetime import datetime`

STEP 6: Wire Up Graph

File: `src/graph.py`

Requirements:

- Import all nodes
- Create StateGraph with WorkflowState
- Add all nodes to graph
- Define edges (entry point, unconditional, conditional)
- Compile graph
- Export compiled graph

Implementation Steps:

1. **Import dependencies:**

```python
from langgraph.graph import StateGraph, END
from src.state import WorkflowState
from src.nodes.orchestrator import orchestrator_node
from src.nodes.supervisor import supervisor_node
from src.nodes.advisor import advisor_agent_node
from src.nodes.operations import operations_agent_node
```

2. **Create graph function:**

```python
```

```python
def build_workflow_graph():
    # Initialize StateGraph
    workflow = StateGraph(WorkflowState)

    # Add nodes
    workflow.add_node("orchestrator", orchestrator_node)
    workflow.add_node("supervisor", supervisor_node)
    workflow.add_node("advisor_agent", advisor_agent_node)
    workflow.add_node("operations_agent", operations_agent_node)

    # Set entry point
    workflow.set_entry_point("orchestrator")

    # Add unconditional edges
    workflow.add_edge("orchestrator", "supervisor")
    workflow.add_edge("advisor_agent", "supervisor")
    workflow.add_edge("operations_agent", "supervisor")

    # Add conditional edges from supervisor
    workflow.add_conditional_edges(
        "supervisor",
        lambda state: supervisor_node(state),  # Routing function
        {
            "advisor_agent": "advisor_agent",
            "operations_agent": "operations_agent",
            "end": END
        }
    )

    # Compile and return
    return workflow.compile()
```

3. **Export graph:**

```python
# Create compiled app
app = build_workflow_graph()
```

STEP 7: Create Main Entry Point

File: `src/main.py`

Requirements:

- Entry point to run workflow
- Accept user input
- Initialize state
- Invoke graph
- Print results

Implementation:

```python
```

```python
from src.graph import app
from src.state import WorkflowState
from datetime import datetime
import uuid

def run_workflow(client_id: str, request_type: str):
    """Execute workflow for given request"""

    # Initialize state
    initial_state = {
        "request": {
            "type": request_type,
            "client_id": client_id,
            "client_name": "",  # Will be filled by orchestrator
            "initiator": "sarah_advisor"
        },
        "workflow_id": "",
        "status": "pending",
        "created_at": "",
        "updated_at": "",
        "context": {},
        "tasks": [],
        "messages": [],
        "decisions": [],
        "blockers": [],
        "next_actions": [],
        "outcome": None
    }

    # Invoke workflow
    print(f"\n{'='*60}")
    print(f"Starting workflow: {request_type}")
    print(f"Client: {client_id}")
    print(f"{'='*60}\n")

    result = app.invoke(initial_state)

    # Print results
    print(f"\n{'='*60}")
    print("WORKFLOW COMPLETED")
    print(f"{'='*60}")
    print(f"Status: {result['status']}")
    print(f"Workflow ID: {result['workflow_id']}")
    print(f"\nTasks Completed: {len([t for t in result['tasks'] if t['status'] == 'completed'])}/{len(result['tasks'])}")
    print(f"\nMessages Exchanged: {len(result['messages'])}")
    print(f"\nDecisions Made: {len(result['decisions'])}")
```

```
    if result['outcome']:
        print(f"\nOutcome: {result['outcome']}")

    if result['blockers']:
        print(f"\nBlockers: {len([b for b in result['blockers'] if b['status'] == 'active'])} active")

    print(f"\n{'='*60}\n")

    return result

if __name__ == "__main__":
    # Test with sample client
    result = run_workflow(
        client_id="john_smith_123",
        request_type="open_roth_ira"
    )

    # Optionally print detailed state
    print("\n=== DETAILED STATE ===")
    print(f"Tasks: {result['tasks']}")
    print(f"\nMessages: {result['messages']}")
    print(f"\nDecisions: {result['decisions']}")
```

STEP 8: Create Test File

File: `tests/test_workflow.py`

Requirements:

- Test successful workflow completion
- Test workflow with document issues
- Verify state changes at each step

Test Cases:

Test 1: Happy Path

```
python
```

```python
def test_successful_ira_opening():
    """Test complete workflow without issues"""

    # Setup: Ensure documents are valid
    from src.storage import doc_store
    doc_store.update_document("john_smith_123", "ira_application", {
        "status": "submitted",
        "signature_page3": True,
        "submitted": True
    })

    # Run workflow
    from src.main import run_workflow
    result = run_workflow("john_smith_123", "open_roth_ira")

    # Assertions
    assert result["status"] == "completed"
    assert len(result["tasks"]) == 5
    assert all(t["status"] == "completed" for t in result["tasks"])
    assert result["outcome"] is not None
    assert "account_number" in result["outcome"]

    print("✅ Test 1 passed: Happy path successful")
```

Test 2: Document Issue

python

```python
def test_missing_signature():
    """Test workflow with missing signature"""

    # Setup: Document missing signature
    from src.storage import doc_store
    doc_store.update_document("john_smith_123", "ira_application", {
        "status": "submitted",
        "signature_page3": False,
        "submitted": True
    })

    # Run workflow
    from src.main import run_workflow
    result = run_workflow("john_smith_123", "open_roth_ira")

    # Assertions
    assert len(result["blockers"]) > 0
    assert any("signature" in b["description"].lower() for b in result["blockers"])
    assert any(m["from_agent"] == "operations_agent" and
               m["to_agent"] == "advisor_agent"
               for m in result["messages"])

    print("✅ Test 2 passed: Blocker detected correctly")
```

Test 3: State Evolution

```python
```

```python
def test_state_updates():
    """Verify state updates correctly through workflow"""

    from src.main import run_workflow
    result = run_workflow("john_smith_123", "open_roth_ira")

    # Check workflow_id was generated
    assert result["workflow_id"] != ""
    assert len(result["workflow_id"]) > 0

    # Check timestamps
    assert result["created_at"] != ""
    assert result["updated_at"] != ""

    # Check messages were added
    assert len(result["messages"]) > 0
    assert all("from_agent" in m for m in result["messages"])
    assert all("to_agent" in m for m in result["messages"])

    # Check decisions were logged
    assert len(result["decisions"]) > 0
    assert all("reasoning" in d for d in result["decisions"])

    print("✅ Test 3 passed: State updates correctly")
```

Run all tests function:

```python
if __name__ == "__main__":
    test_successful_ira_opening()
    test_missing_signature()
    test_state_updates()
    print("\n✅ All tests passed!")
```

STEP 9: Requirements File

File: `requirements.txt`

```
langgraph==0.2.0
langchain==0.3.0
langchain-openai==0.2.0
python-dotenv==1.0.0
```

STEP 10: README

File: `README.md`

````
markdown

# Nexhelm Multi-Agent Workflow System

## Setup

1. Install dependencies:
```bash
pip install -r requirements.txt
````

2. Run workflow:

```
bash

python src/main.py
```

3. Run tests:

```
bash

python tests/test_workflow.py
```

Architecture

- **Orchestrator**: Plans initial tasks
- **Supervisor**: Routes to appropriate agent
- **Advisor Agent**: Handles client communication
- **Operations Agent**: Handles backend operations
- **Shared State**: All nodes read/write to common state

Workflow Flow

1. User submits request
2. Orchestrator creates task plan
3. Supervisor routes to agent
4. Agent executes task
5. Returns to supervisor
```

6. Repeats until complete

Current Status

✅ Hardcoded workflow for "Open IRA"
✅ Two agents (Advisor, Operations)
✅ State management
✅ Blocker detection
⏳ LLM integration (next phase)
⏳ Dynamic task generation (next phase)

```
---

## Validation Checklist

After implementation, verify:

- [ ] Can import all modules without errors
- [ ] `python src/main.py` runs successfully
- [ ] Workflow completes and reaches "completed" status
- [ ] All 5 tasks are marked complete
- [ ] Messages are exchanged between agents
- [ ] Decisions are logged
- [ ] Account number is generated
- [ ] Tests pass
- [ ] Can simulate document issue and blocker is created
- [ ] Supervisor correctly routes between agents
- [ ] State updates after each node execution

---

## Expected Output

When running `python src/main.py`, you should see:
```

==========

Starting workflow: open_roth_ira Client: john_smith_123

Operations Agent: Verifying eligibility - Income: $145000

Advisor Agent: Sending IRA application form to John Smith

Operations Agent: Validating IRA application form

Operations Agent: Opening account - IRA-1000

Advisor Agent: Notifying John Smith of successful account opening

==========

WORKFLOW COMPLETED

Status: completed
Workflow ID: [uuid]

Tasks Completed: 5/5

Messages Exchanged: 5

Decisions Made: 5

Outcome: {'account_number': 'IRA-1000'}

==========

Notes for Implementation

1. Start with Step 1 and proceed sequentially
2. Do not skip steps or implement out of order
3. Test each step before moving to next
4. Use exact field names and structure specified
5. Print statements should match examples for debugging
6. All timestamps should use `datetime.now().isoformat()`
7. All IDs should be simple strings (task_1, blocker_1, etc.) except workflow_id (use uuid4)
8. Follow Python best practices (type hints, docstrings)
9. Keep functions focused and simple
10. After Step 7, the system should be fully functional with hardcoded logic

---

End of implementation guide. Begin with Step 1.