LangGraph Agent Workflow Structure

Stateful agent workflows with graph-based control flow and persistence.

#langchain #langgraph #python #agents #state-machine

PNG PDF Copy Prompt

Project Directory

```
myproject/
    main.py  Graph execution…
  app/
    __init__.py
    config.py
  graph/  LangGraph defin…
      __init__.py
      state.py  TypedDict state…
      nodes.py  Node functions
      edges.py  Conditional rou…
      builder.py  Graph construct…
  agents/
    __init__.py
    planner.py
    executor.py
  tools/
    __init__.py
    search.py
    calculator.py
  checkpointers/  State persisten…
    __init__.py
    memory.py  In-memory check…
    sqlite.py  SQLite persiste…
  requirements.txt
  .env.example
  .gitignore
```

Why This Structure?

LangGraph builds cyclic, stateful agent workflows as directed graphs. Unlike simple chains, it supports loops, conditional branching, and persistent state via checkpointers. The `graph/` folder separates state schema, node logic, and edge routing.

Key Directories

app/graph/state.py - TypedDict defining workflow state

app/graph/nodes.py - Functions that process state

app/graph/edges.py - Conditional routing logic

app/graph/builder.py - StateGraph construction

app/checkpointers/ - State persistence options

Graph Definition

```python
from langgraph.graph import StateGraph

graph = StateGraph(AgentState)
graph.add_node("plan", plan_node)
graph.add_node("execute", execute_node)
graph.add_conditional_edges("plan", should_continue)
app = graph.compile(checkpointer=MemorySaver())
```

When To Use This

- Complex agent workflows with loops
- Human-in-the-loop patterns
- Workflows requiring state persistence
- Multi-step planning and execution

Trade-offs

Learning curve - Graph concepts take time to master

Debugging - Graph execution harder to trace than chains

Overhead - Simpler tasks don't need full graph