LangGraph Agentic IDE Backend – Classroom Exercises

The code implements a **LangGraph-powered agentic IDE backend** for Python development, integrating:

- **Codebase indexing and semantic search** (via Chroma + HuggingFace embeddings)
- **Static code analysis** (AST-based dependency graph)
- **Language Server Protocol (LSP)** integration (hover, go-to-definition, symbols)
- **Tool-augmented agents** with a planner–executor loop
- **Checkpointed stateful workflows** (SQLite)

Below are three hands-on classroom exercises that progressively build trainee understanding.

Exercise 1: Debug the Planner to Fix Hallucinated Actions

Goal: Understand LangGraph node design and strict role separation.

Context: The `planner_node` is meant to only output a plan, but the LLM sometimes hallucinates actions such as "I created api.py".

Task:

- Modify `planner_node` to validate its output.
- Add a post-processing check: if the plan contains phrases such as "I created", "written", or "you can run", re-prompt the LLM (max 2 retries).
- Track retries in LangGraph state.

Learning Outcomes:

- Enforcing agent role boundaries
- State mutation in LangGraph
- Self-correction loop inside a node

Hint:

```
def planner_node(state: AgentState):
    retries = 0
    while retries < 2:
        response = llm.invoke([...])
        content = response.content
        if any(bad in content.lower() for bad in
               ["i␣created", "written", "saved", "you␣can␣run"]):
            messages = state["messages"] + [
                response,
                SystemMessage(content=
                    "ERROR:␣Your␣response␣contained␣action␣claims.␣"
                    "Only␣output␣a␣numbered␣plan.␣NO␣actions.␣NO␣outcomes."
                )
            ]
            response = llm.invoke(messages)
            retries += 1
        else:
            break
    return {"plan": response.content}
```

Exercise 2: Extend the Agent with Reference-Aware Editing

Goal: Use LSP `get_references` to enable safe, context-aware code updates.

Context: The agent can write code but does not know where a function is used. Changing a function signature without updating callers breaks the code.

Task:

- Add a tool: `lsp_references(file_path, line, column) -> List[str]`
- Update the executor node's system prompt to instruct: "Before modifying a function, use `lsp_references` to find all call sites. If references exist, update them too."
- Test by asking the agent to rename `greet` to `say_hello` in `demo_app.py`.

Expected Behavior:

1. Locate the function definition
2. Find references such as `print(greet(...))`
3. Update both definition and call site

Learning Outcomes:

- Tool integration
- Safe refactoring workflows
- Combining LSP data with agent reasoning

Bonus: Add a validation step running `python demo_app.py`.

Exercise 3: Build a "Fix My Code" Human-in-the-Loop Workflow

Goal: Add a human approval stage to the LangGraph workflow.

Context: Real IDEs involve human review before changes are written.

Task:

- Extend `AgentState` with a field `awaiting_approval: bool`.
- Add a `reviewer_node` that:
  - Detects when a `write_file` tool call is pending
  - Pauses execution and prints: "Proposed change to {file}. Approve? (y/n)"
  - Continues on approval or requests corrections using a HumanMessage
- Use LangGraph's interrupt/resume capabilities.

Demo Prompt: "Add type hints to the greet function in demo_app.py."
Expected proposal:

```
def greet(name: str) -> str:
    ...
```