## Sequential Workflow Basics (no LLM): BMI Calculator

**Goal:** Learn LangGraph syntax without LLMs.

**State design (TypedDict):**

- `weight: float (kg)`
- `height: float (m)`
- `bmi: float`

**Graph recipe (always this order):**

1. **Define State** (TypedDict).
2. **Create graph**: `graph = StateGraph(BMIState)` .
3. **Add nodes**: each node maps to a **Python function** taking `state` and returning updated `state` .
   - Node: `calculate_bmi(state) → state`
     - Read `weight` , `height`
     - Compute `bmi = weight / (height**2)` and round
     - Update and return `state`
4. **Add edges**: `START -> calculate_bmi -> END`
5. **Compile**: `workflow = graph.compile()`
6. **Run**: `final_state = workflow.invoke({"weight": 80, "height": 1.73})`

**Key ideas:**

- A node **always** receives the **current state** and **returns state**.
- The graph **input** and **final output** are **state** objects.

**Visualizing the graph (in notebooks only):**

- Use the snippet from LangGraph docs to render; you'll see `START → calculate_bmi → END` .

---

## 4) Extending the BMI Flow (still sequential)

**New feature:** Label BMI category based on computed BMI.

**Changes:**

- Add `category: str` to state.
- New node `label_bmi(state)` :
  - Read `bmi` and set `category` (e.g., `<18.5 = Underweight` , `18.5–24.9 = Normal` , etc.)
- New edge: `calculate_bmi -> label_bmi -> END`

**Takeaway:** Growing a linear pipeline = **add node + add edge + maybe extend state**.

---

## 5) Minimal LLM Workflow (Q&A)

**State:**

- `question: str , answer: str`

**Node:** `llm_qa(state)`

- Build a prompt from `state["question"]`
- Call an LLM (via `langchain-openai` ), set `state["answer"]`
- Return state

**Graph:** `START -> llm_qa -> END`

**Why show this:** To demonstrate how **LangGraph (flow)** + **LangChain (LLM/model)** work hand-in-hand.

**Note:** For a single LLM call, LangGraph is overkill—its power shows on **larger flows**.

---

## 6) Prompt Chaining Workflow (two LLM calls in sequence)

**Goal:** Provide topic → get **outline** → use topic+outline → generate **blog**.

**State (TypedDict):**

- `title: str , outline: str , content: str`

**Nodes:**

1. `create_outline(state)`
   - Prompt LLM: “Generate a detailed outline for a blog on: {title}”
   - Set `state["outline"]`
2. `create_blog(state)`
   - Prompt LLM: “Write a detailed blog on {title} using this outline:\n{outline}”
   - Set `state["content"]`

**Edges:** `START → create_outline → create_blog → END`

**Benefit of LangGraph here:** Because the **state persists**, your final result can include **all intermediate artifacts** (title, outline, content). In a plain LangChain “chain,” you typically only keep the last output unless you wire custom memory.

---

## 7) Why LangGraph (in practice)

- **Stateful by design:** Every node sees and updates the **same evolving state**.
- **Explicit control flow:** Nodes + edges make execution order crystal clear.
- **Scales from simple to complex:** Start linear; later add branches, conditions, parallelism, retries.
- **Great with LangChain:** Use LangChain for models/prompts/loaders; use LangGraph to **orchestrate**.

---

## 8) Typical Patterns (cheat-sheet)

- **Define state**

```
class MyState(TypedDict): a: str b: str
```

- **Make graph & nodes**

```
graph = StateGraph(MyState) def step1(state: MyState) -> MyState: ... def
step2(state: MyState) -> MyState: ... graph.add_node("step1", step1)
graph.add_node("step2", step2)
```

- **Edges & run**

```
from langgraph.graph import START, END graph.add_edge(START, "step1")
graph.add_edge("step1", "step2") graph.add_edge("step2", END) workflow =
graph.compile() out = workflow.invoke({"a": "x", "b": "y"})
```

---

## 9) Homework (practice idea)

- Extend the **prompt-chaining** flow with a **third node**: `evaluate_blog`
  - Prompt: "Based on this outline, rate the blog (integer score)."
  - Add `score: int` to state.
  - Add edge: `create_blog → evaluate_blog → END` .
  - Now your final state exposes `title` , `outline` , `content` , **and** `score` .

---

## 10) Key Takeaways

- **Sequential flows** in LangGraph = define **state** → **nodes** → **edges** → **compile** → **invoke**.
- Nodes are **plain Python functions** with signature `state_in → state_out` .
- Use **LangChain** for LLMs, prompts, tools; use **LangGraph** to **compose** them.
- The **real payoff** appears as flows get **bigger** (multi-step, branching, retries, tool calls).

If you'd like, I can turn this into a one-page printable cheat-sheet or add branching/guardrails examples