Course Outline

1. LangGraph Orientation
2. LangGraph foundations
3. Build an Application

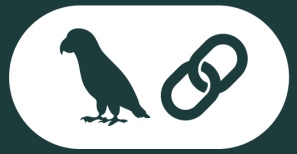
LangGraph Orientation

LangGraph

Agents and LLM applications have these challenges

- Latency in the seconds vs ms
  - Parallelization – to save actual latency
  - Streaming – to save perceived latency

https://blog.langchain.com/building-langgraph



LangChain

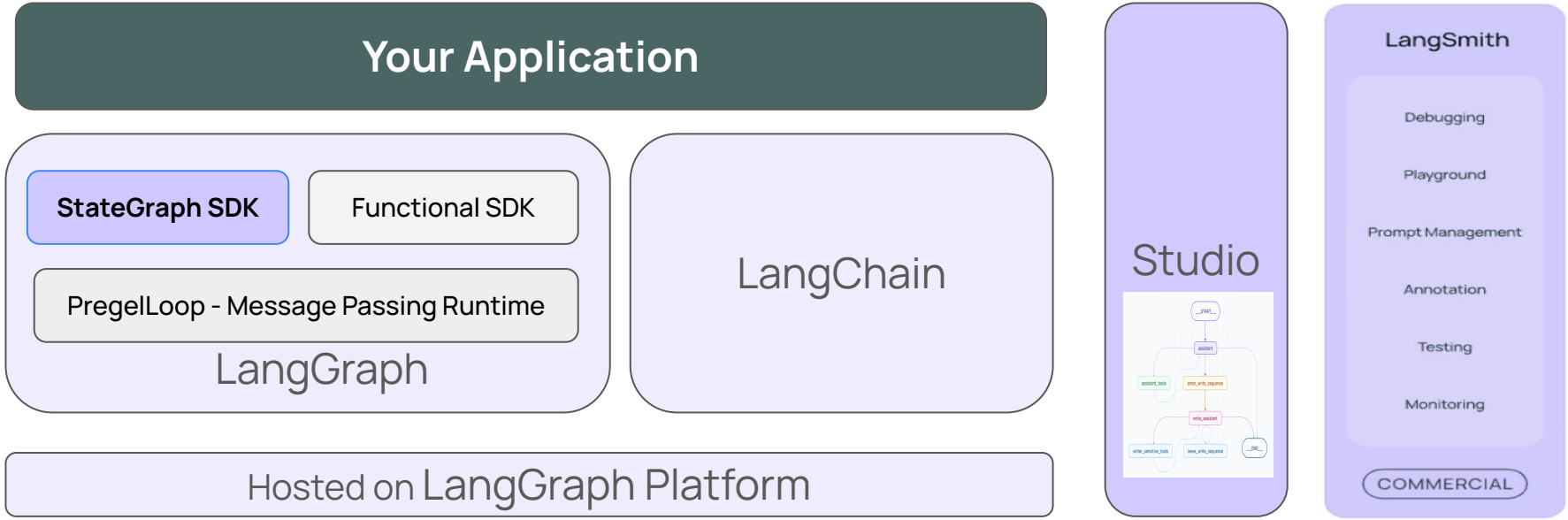
LangGraph: StateGraph Essentials

LangGraph

[NAME]
[TITLE], LangChain

- Parallelization – to save actual latency
- Streaming – to save perceived latency
- Task queue – to reduce number of retries (LangGraph Platform)
- Checkpointing – to reduce the cost of each retry
- Human-in-the-loop - to collaborate with the user
- Tracing - to learn how users use it (LangSmith)

Layer Diagram



Hidden

Execution algorithm

Once you make the choice to structure agents into multiple discrete steps, you need to choose some algorithm to organize its execution. Even if it's a naive one that feels like "no algorithm," which is where LangGraph started before launch. The problem with using "no algorithm" is, while it may seem simpler, you're making it up as you go along, and end up with unexpected results. (For instance, an early version of a precursor to LangGraph suffered from non-deterministic behavior with concurrent nodes). The usual DAG algorithms (topological sort and friends) are out of the picture, given we need loops. We ended up building on top of the **BSP/ Pregel** algorithm, because it provides deterministic concurrency, with full support for loops (cycles).

Our execution algorithm works like this:

Channels contain data (any Python/JS data type), and have a name and current version (a monotonically increasing string)
Nodes are functions to run, which subscribe to one or more channels, and run whenever they change
One or more channels are mapped to **input**, ie. the starting input to the agent is written to those channels, and therefore triggers any nodes subscribed to them
One or more channels are mapped to **output**, ie. the return value of the agent is the value of those channels when execution halts

The execution proceeds in a loop, where each iteration

Selects the 1 or more nodes to run, by comparing current channel versions and the last versions seen by each of their subscribers
Executes those nodes in parallel, with independent copies of the channel values (ie. the state, so they don't affect each other while running)
Nodes modify their local copy of the state while running
Once all nodes finish, the updates from each copy of the state are applied to their respective channels, in a deterministic order (this is what guarantees no data races), and the channel versions are bumped

The execution loop stops when there are no more nodes to run (ie. after comparing channels with their subscriptions we find all nodes have seen the most recent version of their subscribed channels), or when we run out of iteration steps (a constant the developer can set).

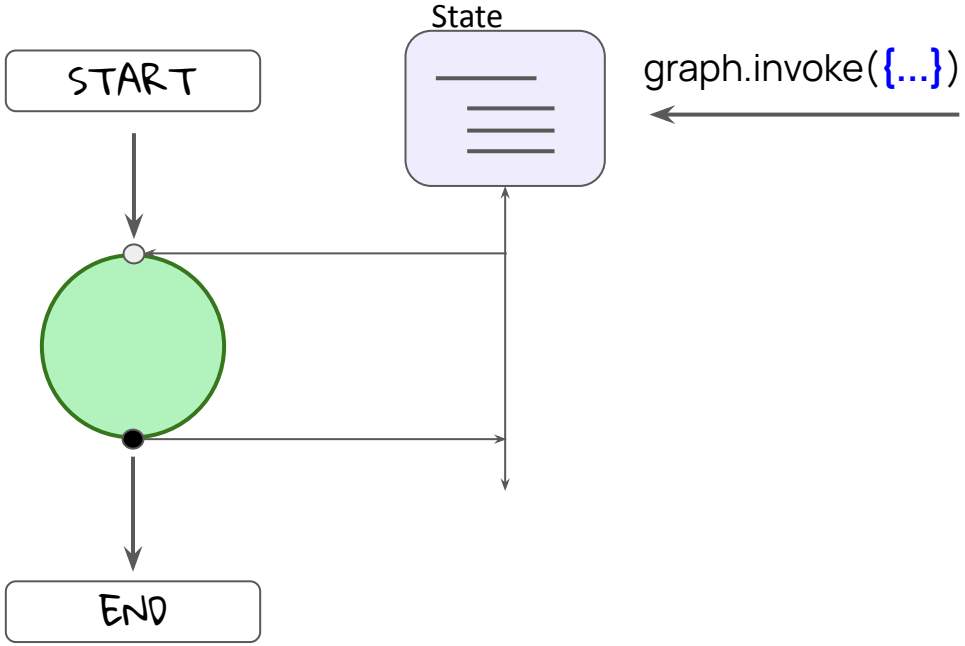
LangGraph: StateGraph Foundations

LangGraph:StateGraph

Components and Capabilities

- State: Data
- Node: Functions
- Edges: Control Flow
  - Serial, Parallel
  - Conditional
- Checkpointing/Memory
- Human In the Loop: Interrupts

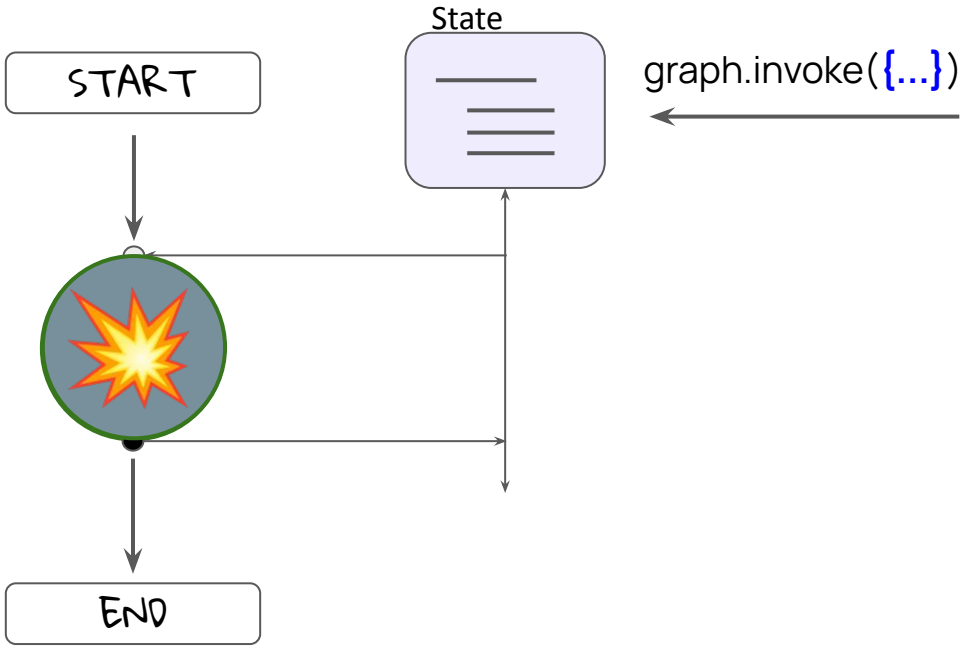
State, Nodes




```
class State(TypedDict):
    nlist : List[str]
```

```
def node_a (state: State):
    ...
    return({"nlist": [note]})
```

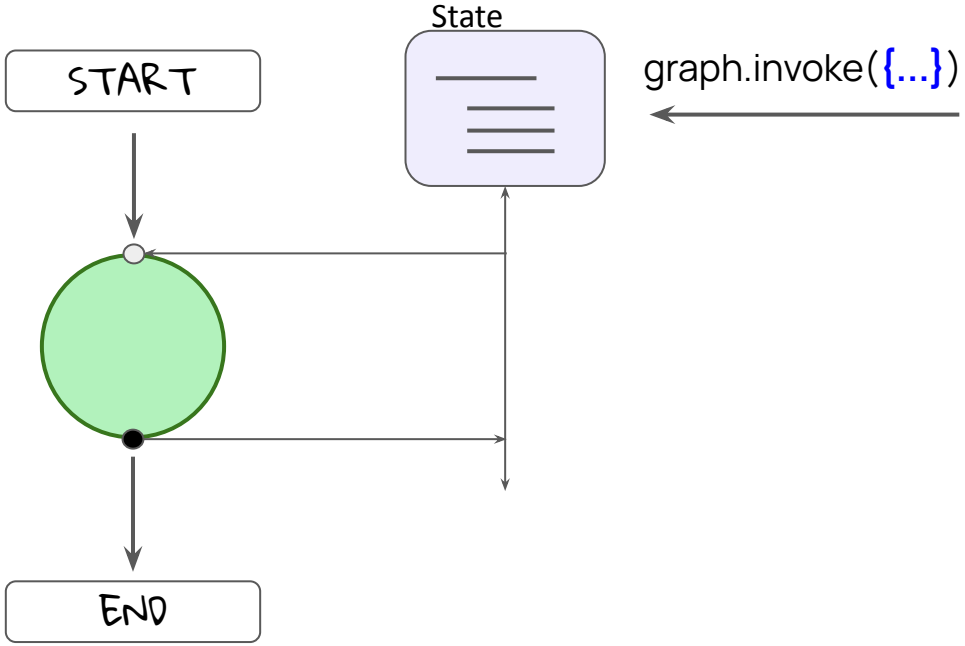
State, Nodes




```
class State(TypedDict):
    nlist : List[str]
```

```
def node_a (state: State):
    ...
    return({"nlist": [note]})
```

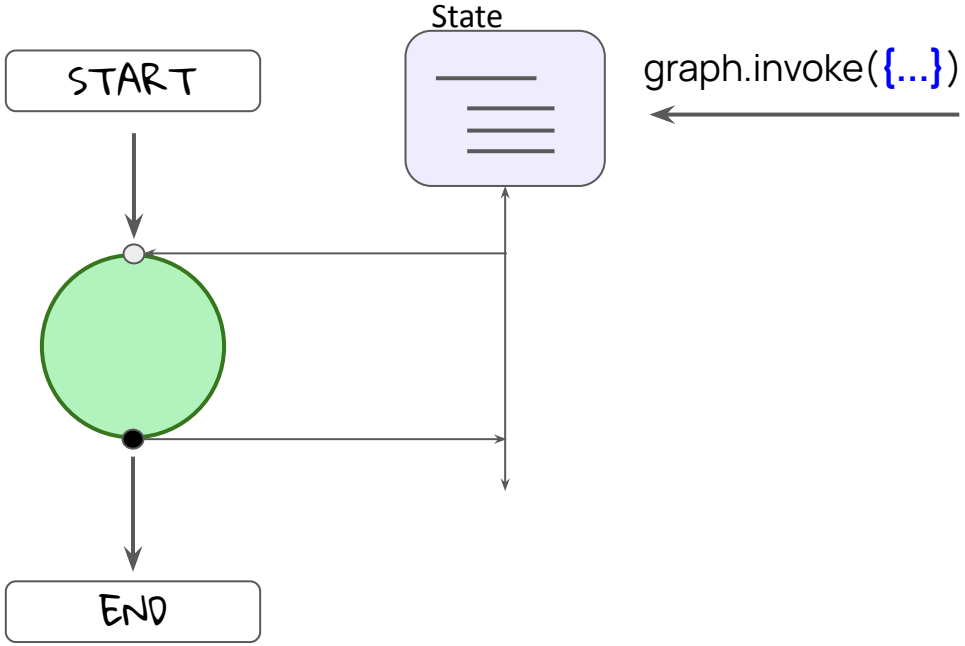
State, Nodes




```
class State(TypedDict):
    nlist : List[str]
```

```
def a (state: State):
    ...
    return({"nlist": [note]})
```

State, Nodes



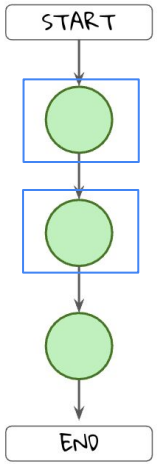

```
class State(TypedDict):
    nlist : List[str]
```

```
def a (state: State):
    ...
    return({"nlist": [note]})
```

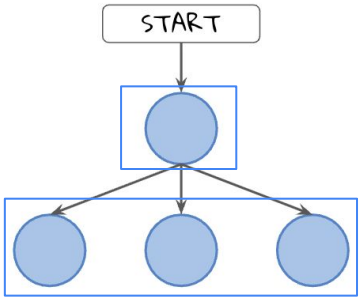
Edges: Control Flow

Edge

Serial

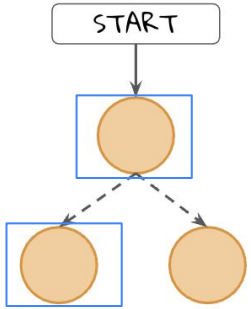


Parallel

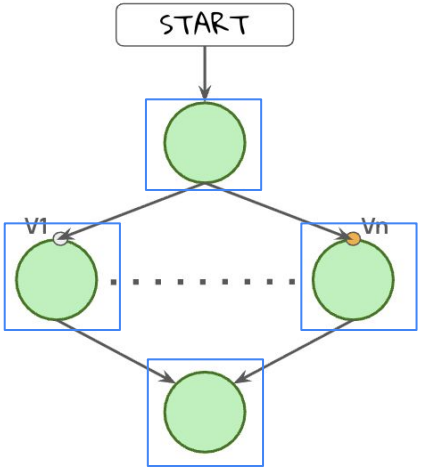


Conditional Edge

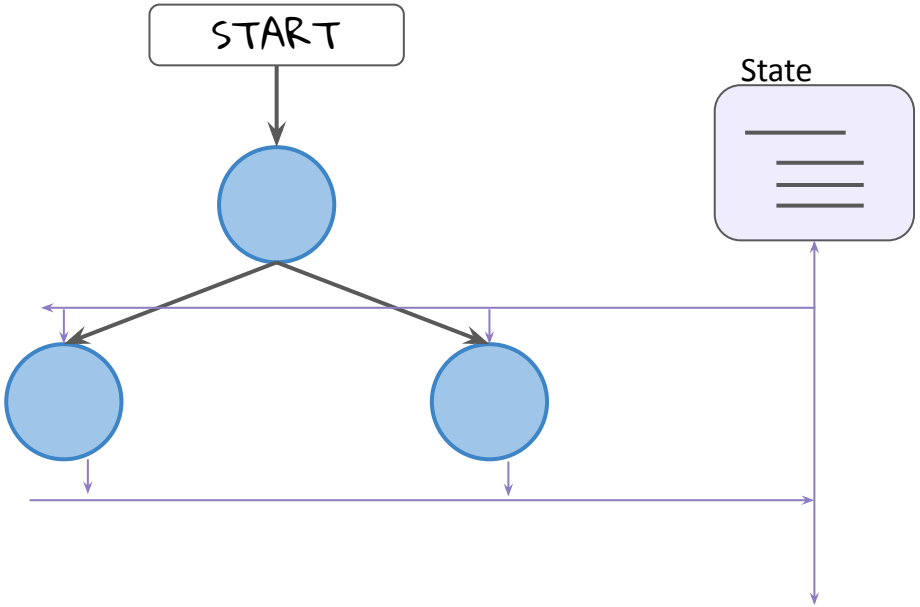
Conditional



Map-Reduce

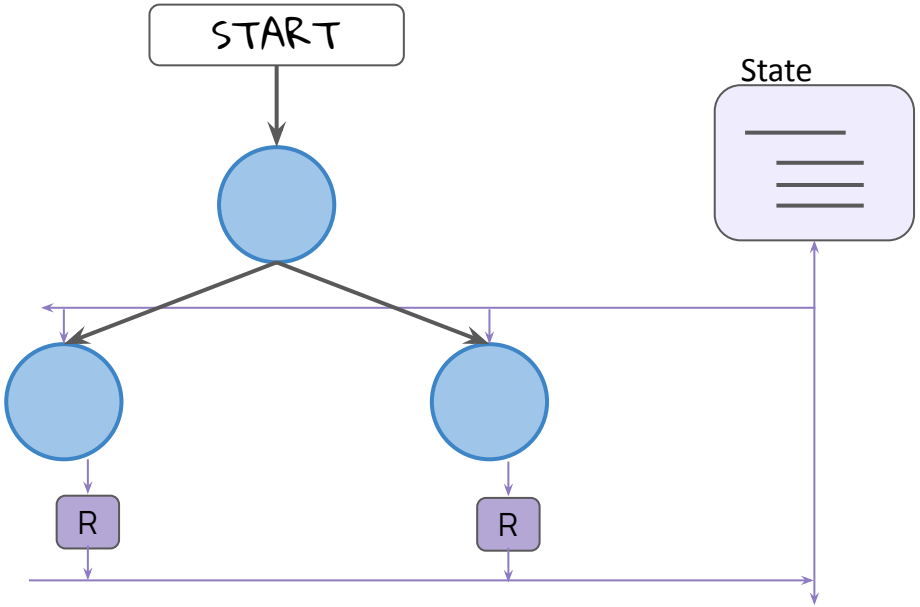


Reducers




```
class State(TypedDict):
    nlist : List[str]
```

Reducers



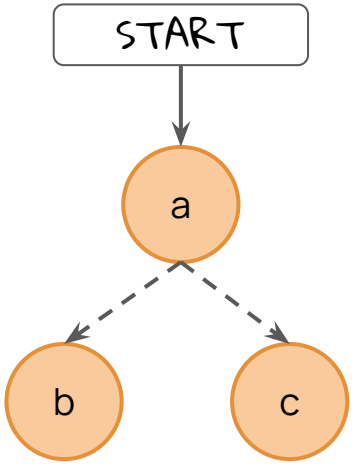

```
class State(TypedDict):
    nlist : Annotated[list[str], operator.add]
```

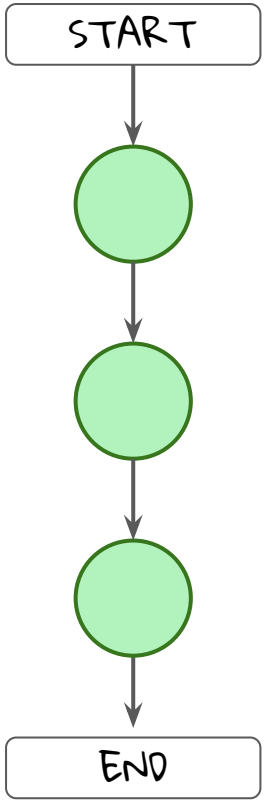
type

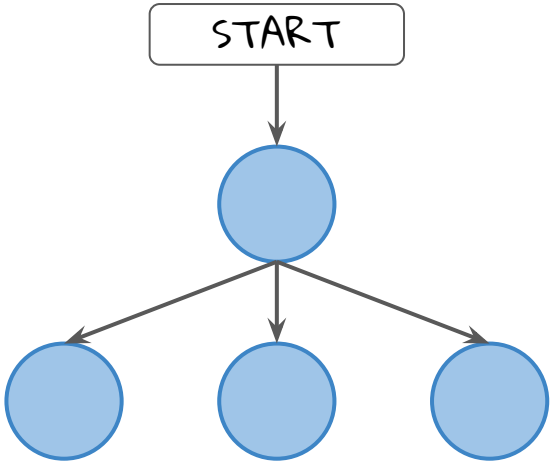
reducer function

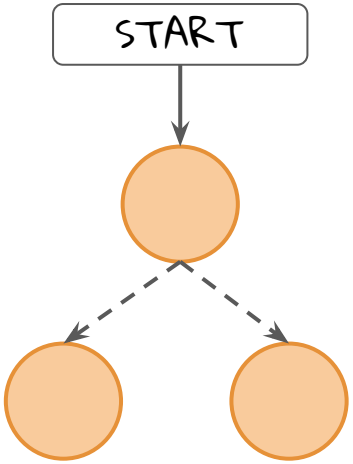
Lab - Parallel/Conditional Edges

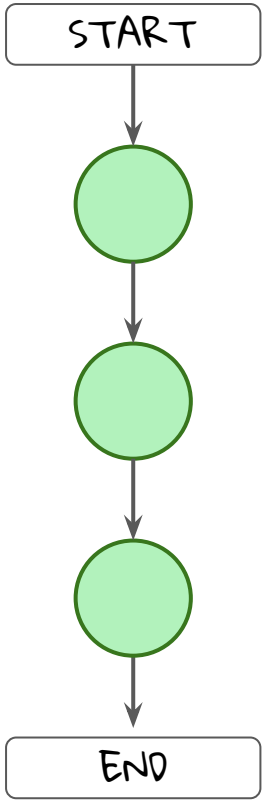
Conditional Edge

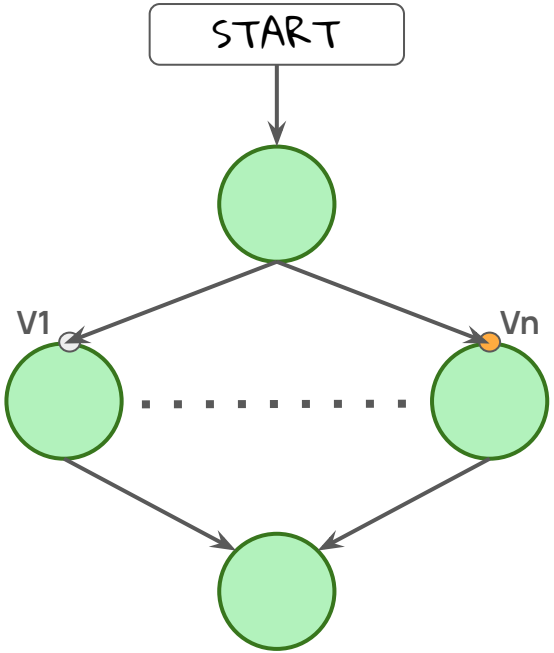


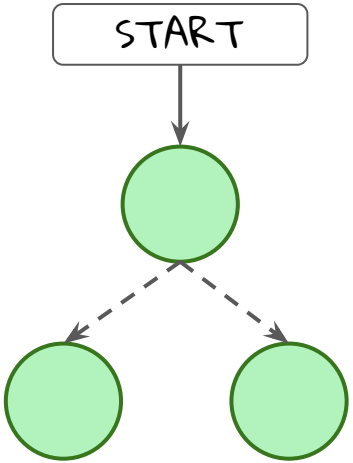

START
END


START

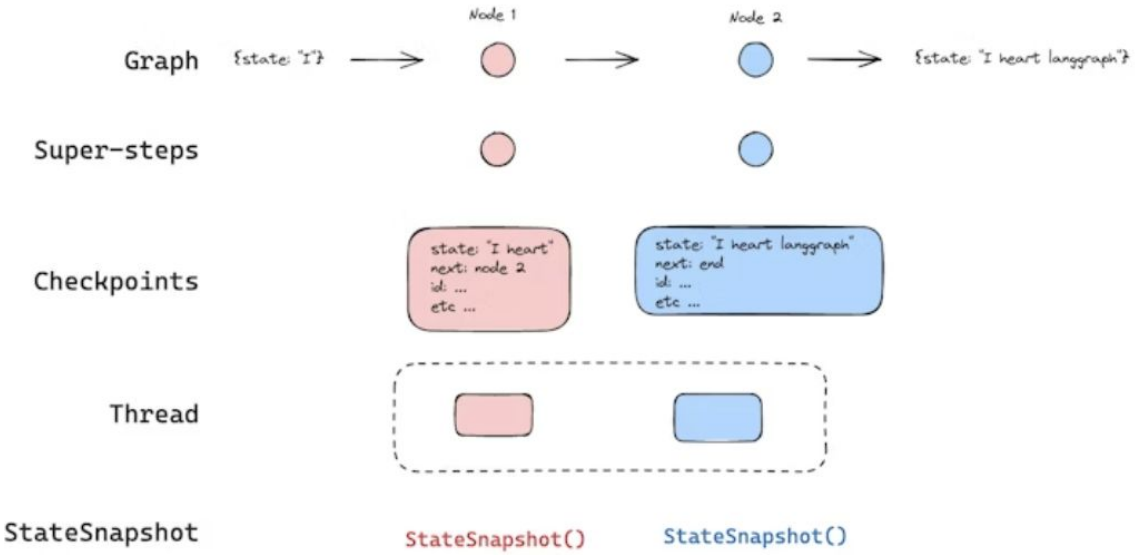

START


START
END


START
V1
Vn


START

Memory



Control flow of nodes, edges

Each sequential node is a separate super-step, while parallel nodes share the same super-step

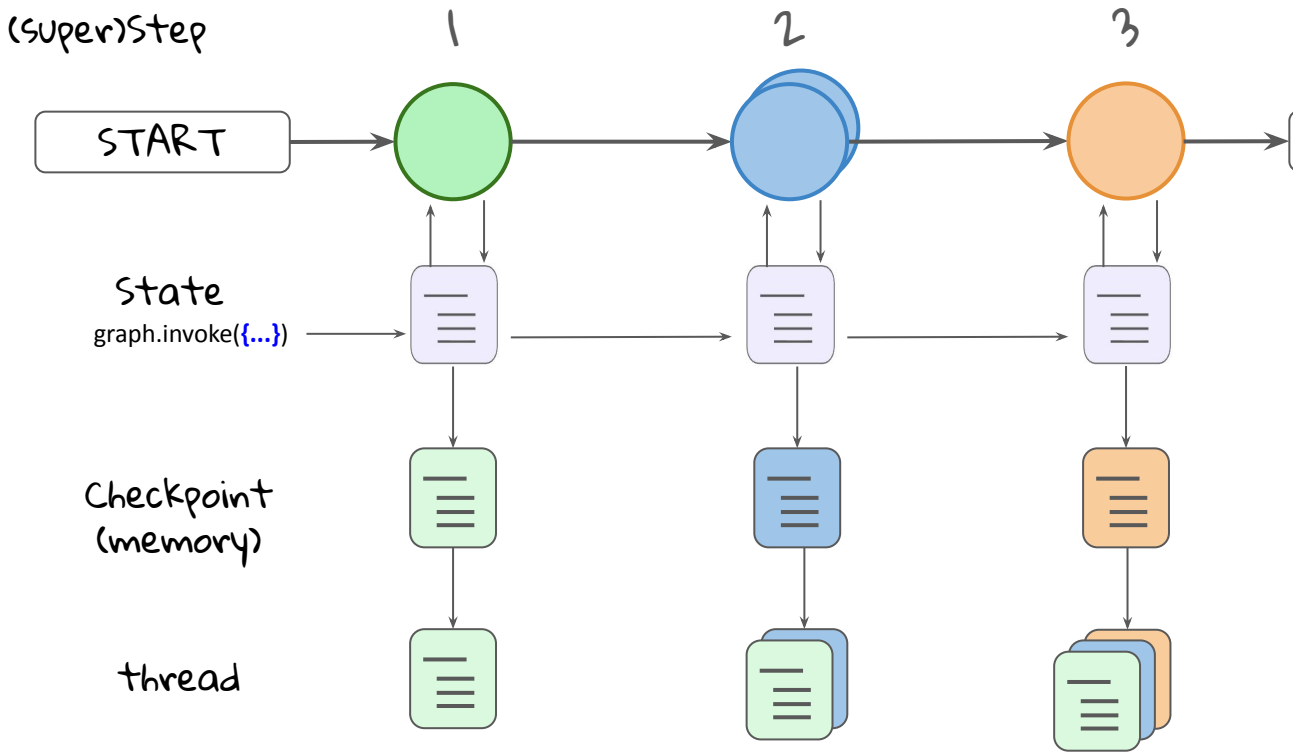
State and relevant metadata packaged at every super-step

Collection of checkpoints

Type for checkpoints

Back from Lab Memory

Memory / Checkpointers



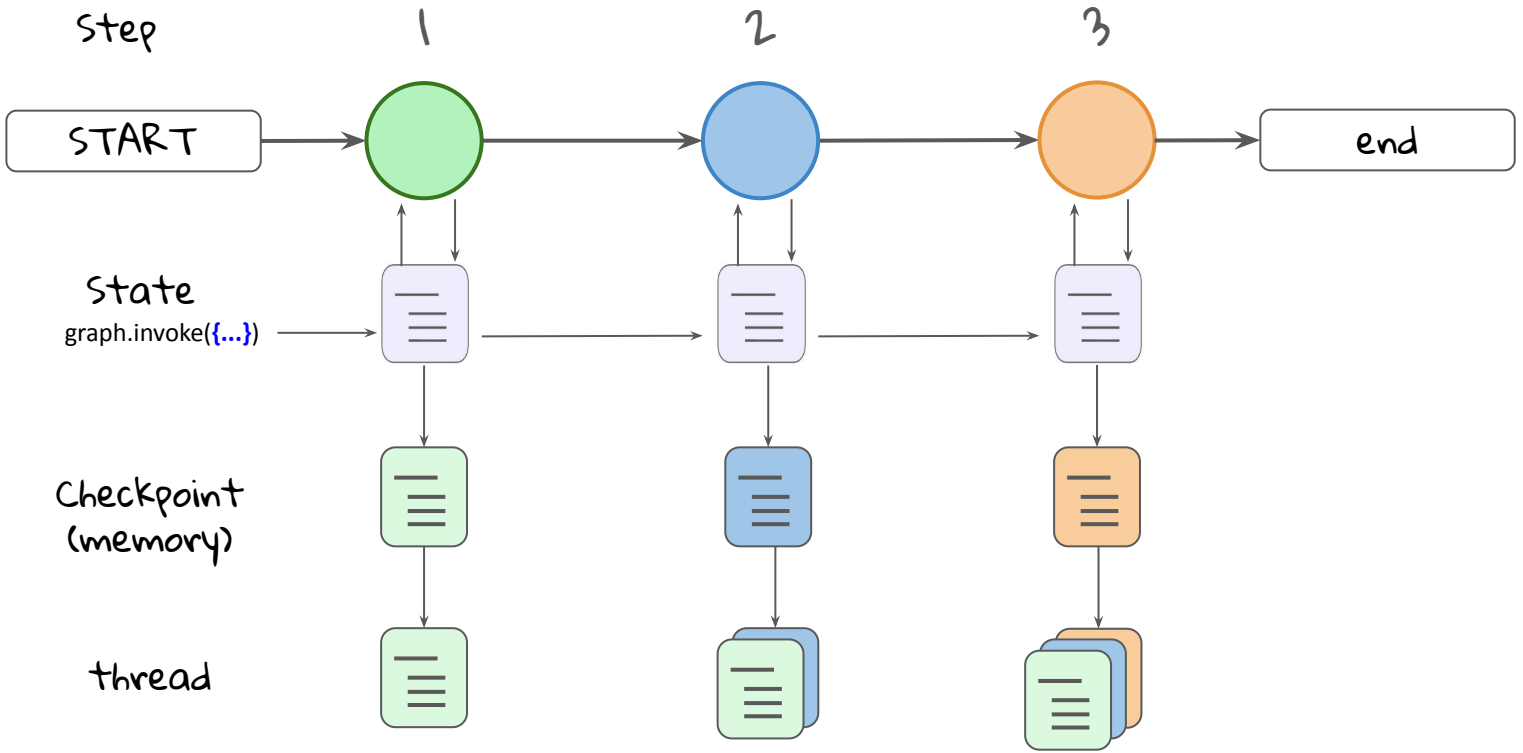
Nodes execute in sequence, per step

State is provided to the node at the start of a step and updated at the end of a step

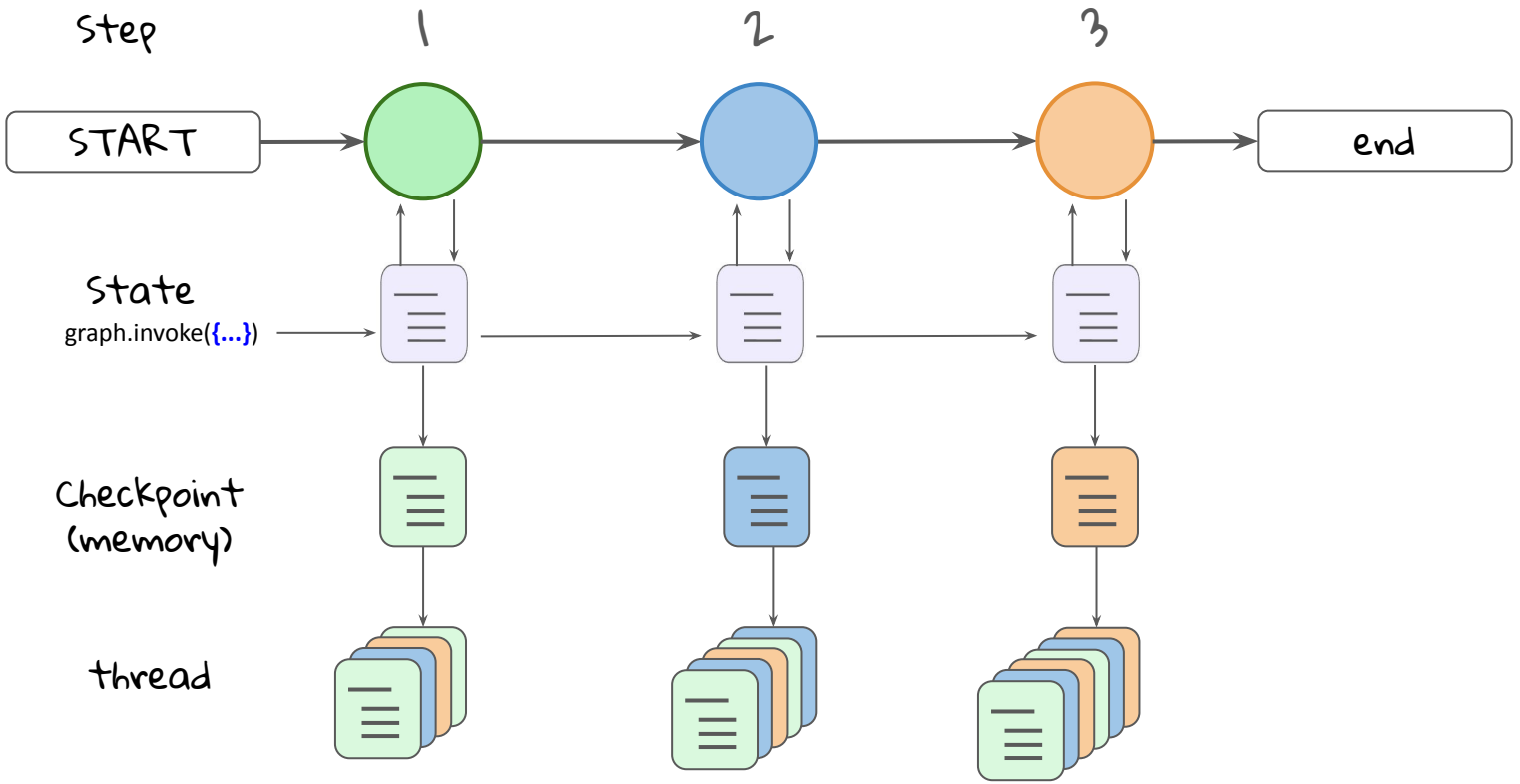
State is stored to a **checkpoint** at the end of a step

A **thread** identifies a collection of checkpoints

Memory


Step
1
2
3
START
end
State
graph.invoke({...})
Checkpoint
(memory)
thread

Memory


Step
1
2
3
START
end
State
graph.invoke({...})
Checkpoint
(memory)
thread

Memory

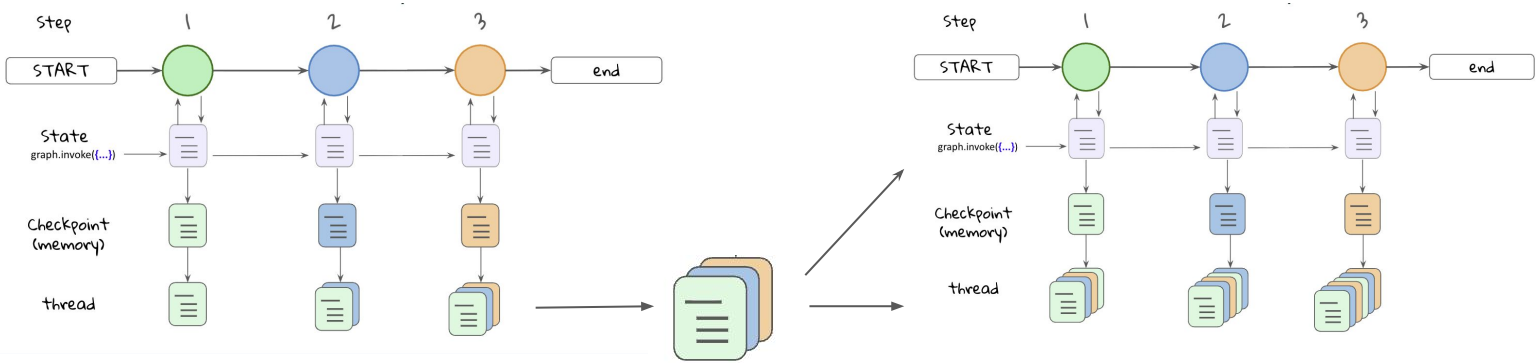
Benefits

- **Recover gracefully from failures** — resume without losing progress.
- **Time travel** — roll back to a known good point and continue forward.

Memory / Checkpointers

Benefits

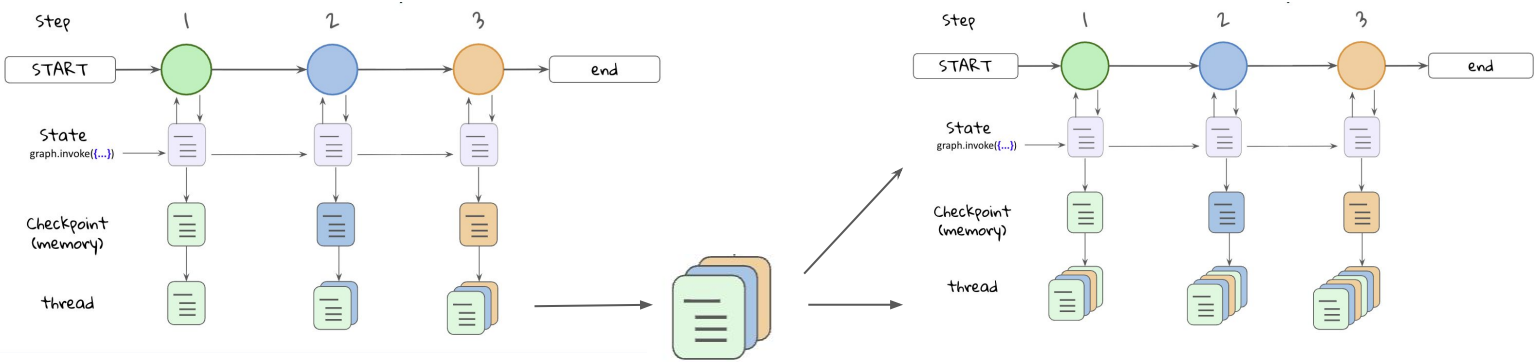
- **Recover gracefully from failures** — resume without losing progress.
- **Time travel** — roll back to a known good point and continue forward.
- **Persistent state** — data is preserved even when the graph is not running.



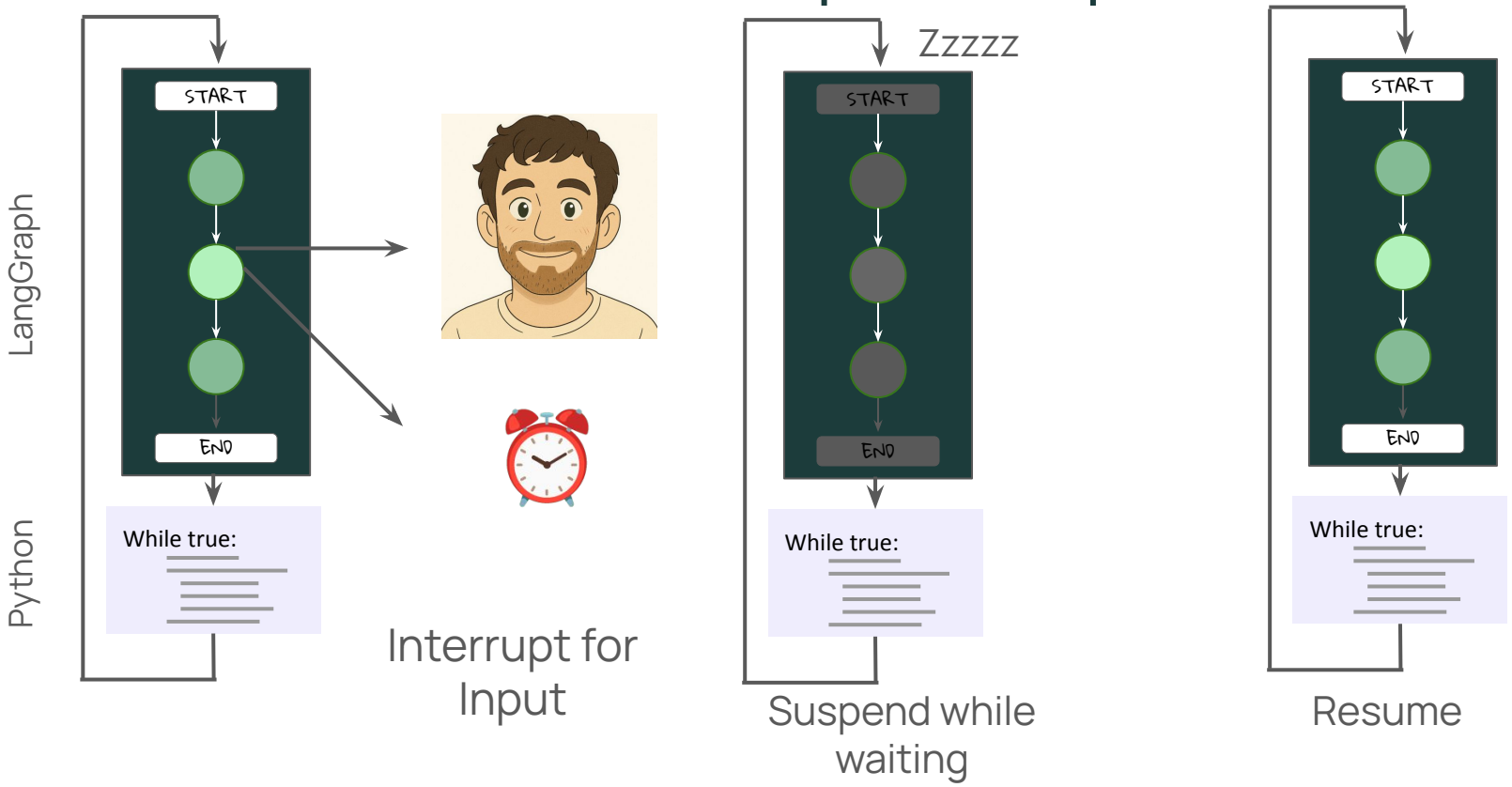
Memory / Checkpointers

Benefits

- **Recover gracefully from failures** — resume without losing progress.
- **Time travel** — roll back to a known good point and continue forward.
- **Persistent state** — data is preserved even when the graph is not running.
- **Restore state at any step** — pick up execution from where you left off.



Human In the Loop: Interrupt



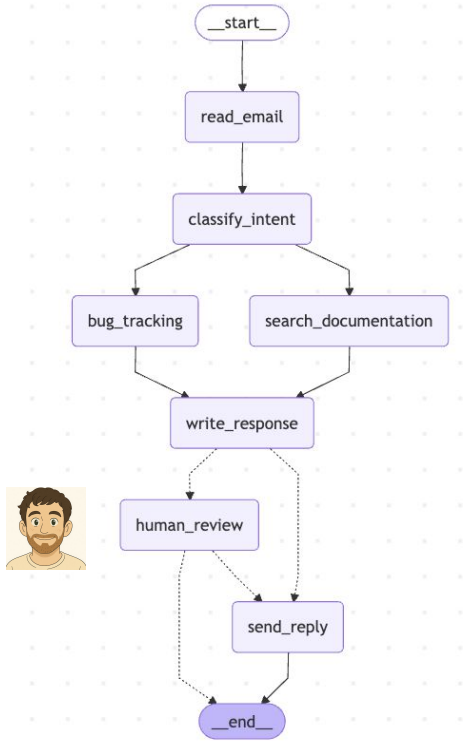
Lab Interrupts

LangGraph:StateGraph

Components and Capabilities

- State: Data
- Node: Functions
- Edges: Control Flow
  - Serial, Parallel
  - Conditional
- Checkpointing/Memory
- Human In the Loop: Interrupts

Email Support Workflow

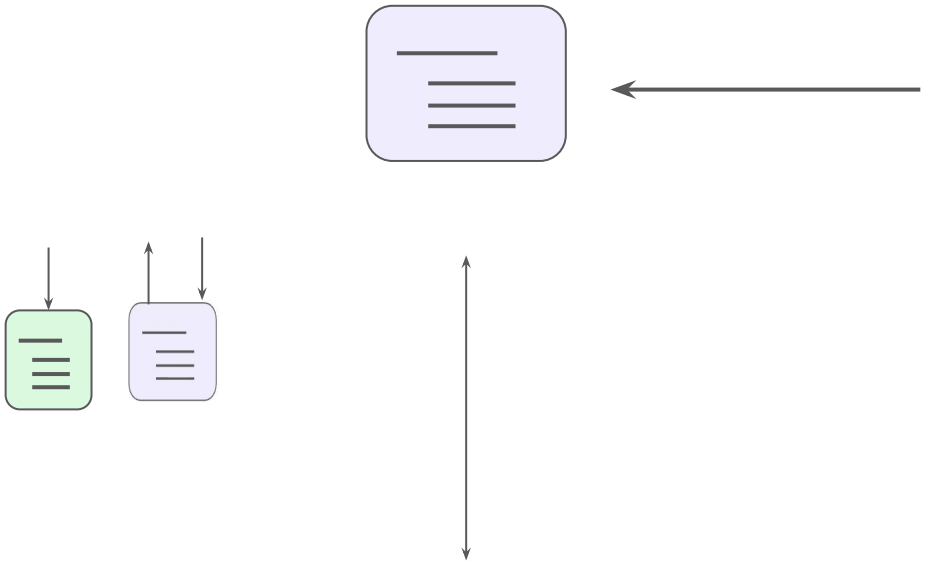


Simulate a email customer support workflow

Focus on LangGraph aspects:

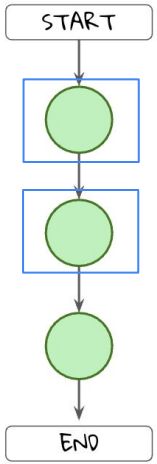
- State, Nodes,
- Edges
  - Serial
  - Parallel
  - Conditional
- Memory
- Interrupt

Memory

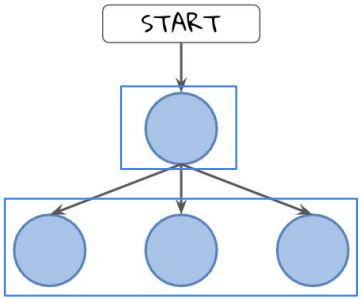


Edge

Serial

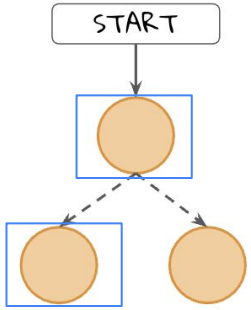


Parallel



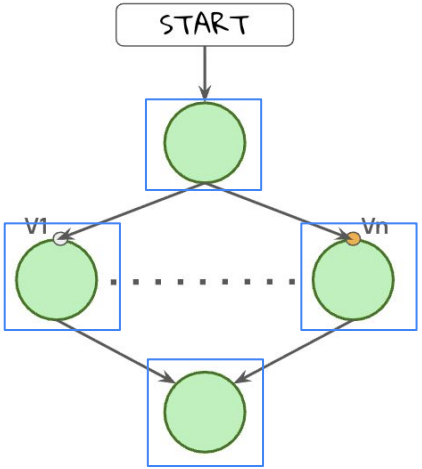
Conditional Edge

Conditional

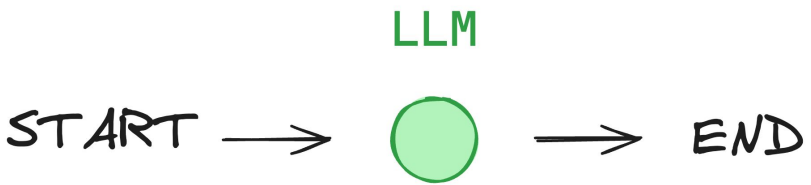


.add_condition_edge()
Command()

Map-Reduce

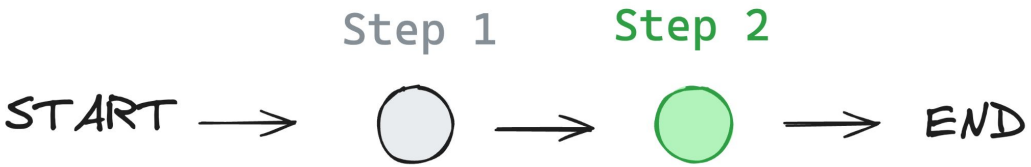


A solitary language model is fairly limited...



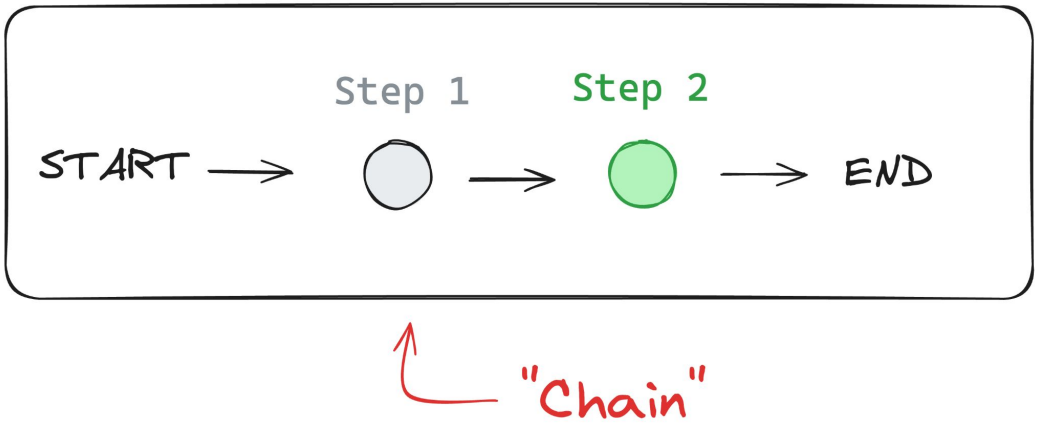
... it lacks relevant context.

Good LLM applications follow a control flow.



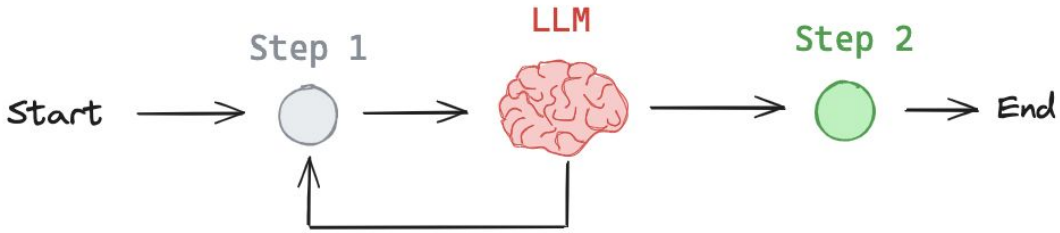
Reference: https://en.wikipedia.org/wiki/Control_flow

This control flow forms a “chain”



Example: search → LLM (RAG) chain

Agent ~= control flow defined by an LLM

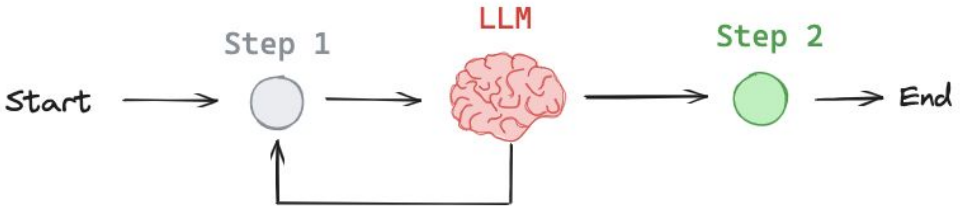


Chain

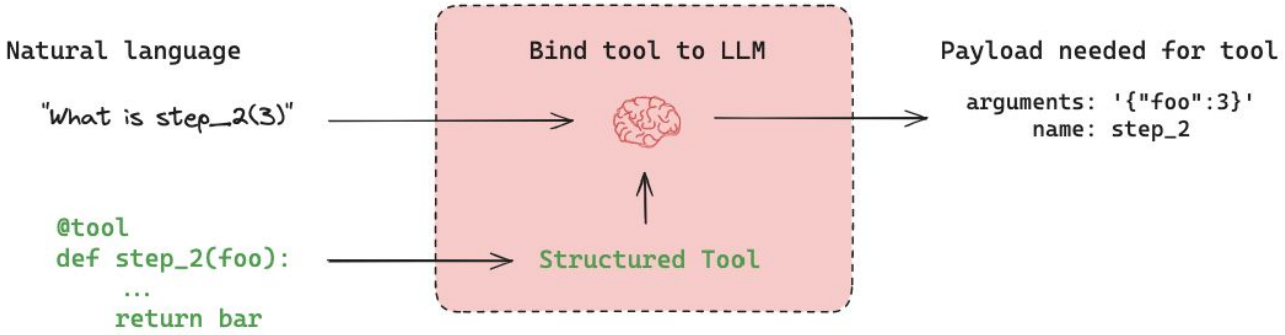
Developer defined control flow

Agent

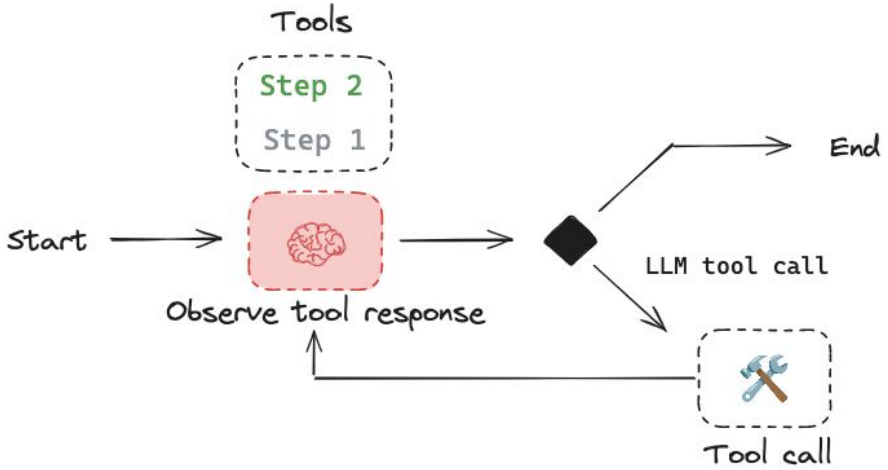
LLM defined control flow



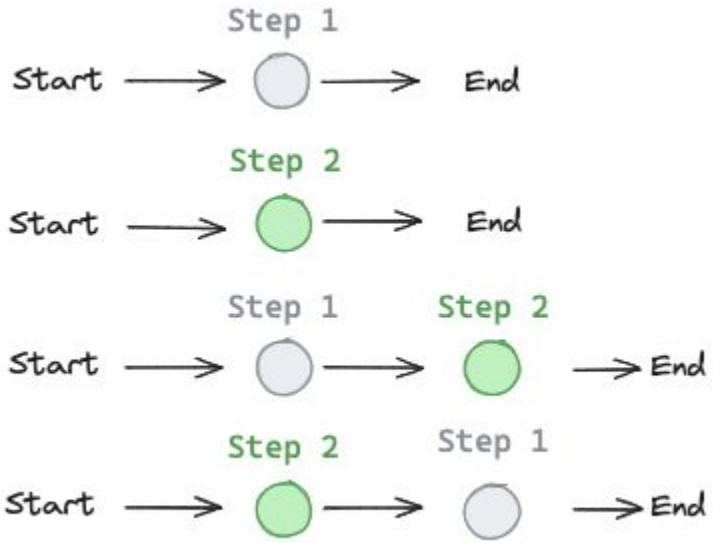
Agents typically use tools-calling to execute steps



Simplest design is a `for loop` (ReAct)



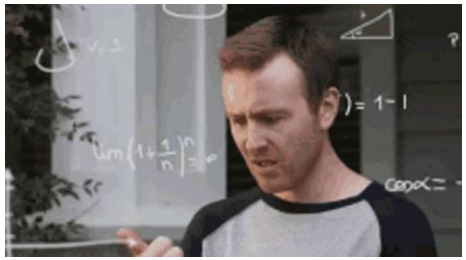
ReAct agents are flexible: any state possible!

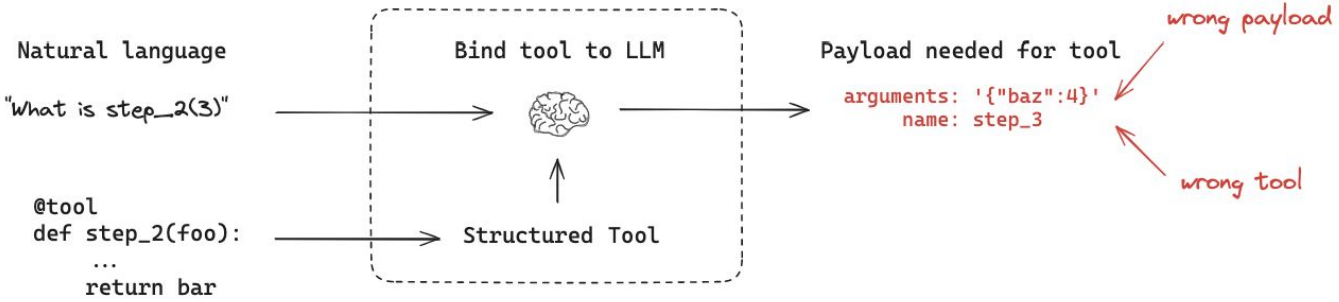


… but they suffer from poor reliability

… often caused by

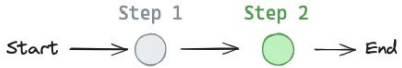
- Task ambiguity
- LLM non-determinism
- Tool misuse
- Tool dependencies
- … and more!





Can we have both?

Chain



Not flexible

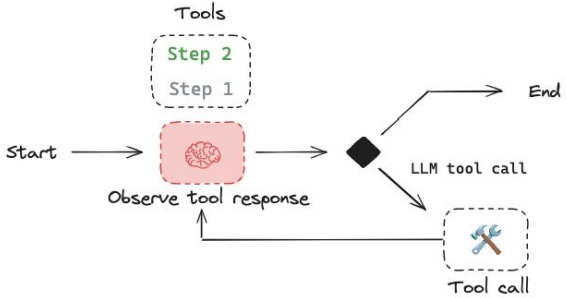
More reliable

?

Flexible

Reliable

Agent (for loop)



Flexible

Less reliable

2 Introducing  LangGraph

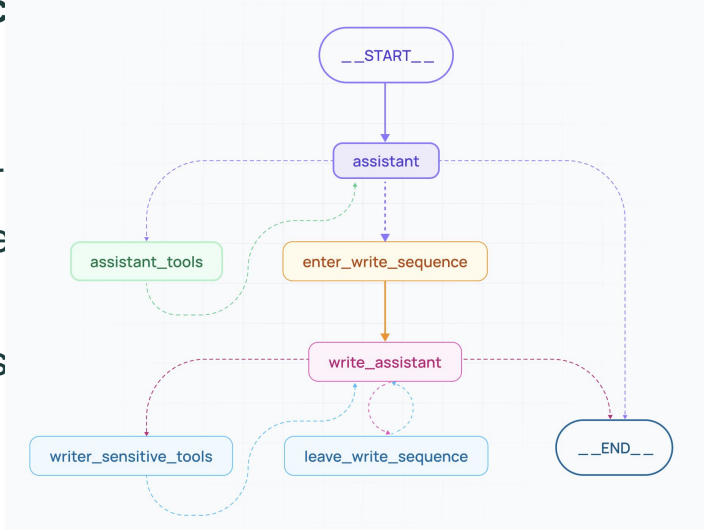
What is LangGraph?

LangGraph applications balance agent control with agenc

Its core pillars support:

A
B
C
D

- **Controllability:** to define both explicit and implicit wor
- **Persistence:** to enable human-agent/multi-agent inte
- **Human-in-the-loop:** to facilitate human guidance
- **Streaming:** to expose any event (or token) as it occurs

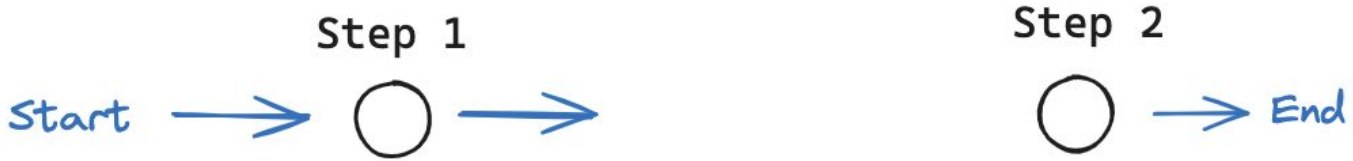


LangGraph also:

- Works with or without LangChain
- Integrates with LangSmith

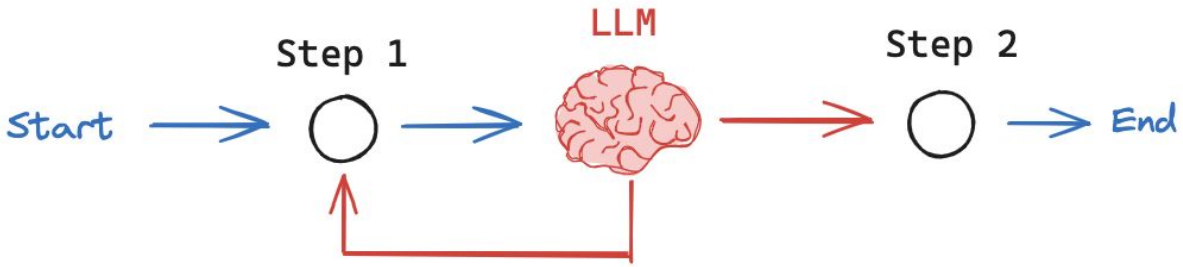
A Controllability

Intuition: Let developer set parts of control flow (reliable)



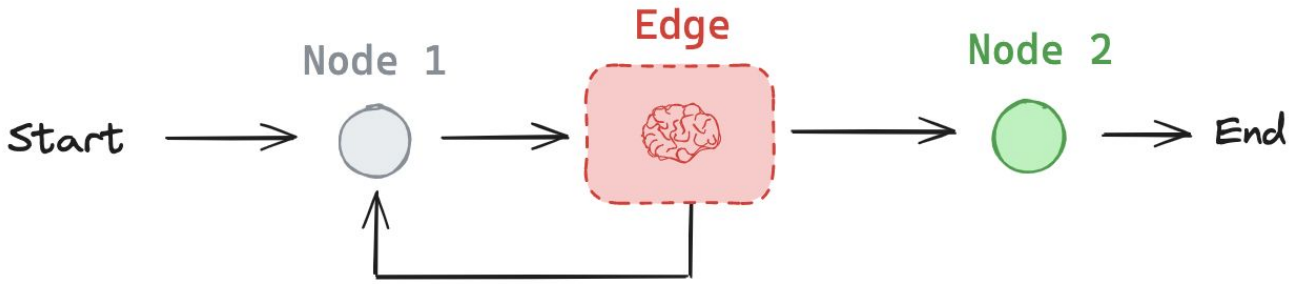
A Controllability

Intuition: Inject LLM to make it an agent (flexible)



A Controllability

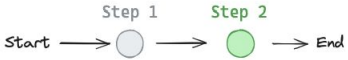
LangGraph: Express control flows as graphs



A Controllability

We can have <u>both</u>!

Chain



Not flexible

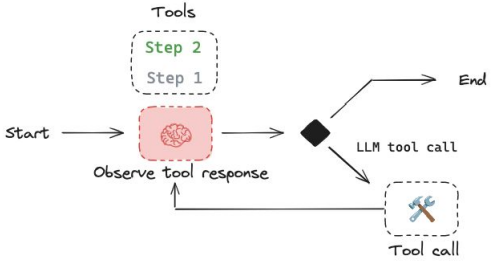
More reliable

LangGraph

Flexible

Reliable

Agent (for loop)

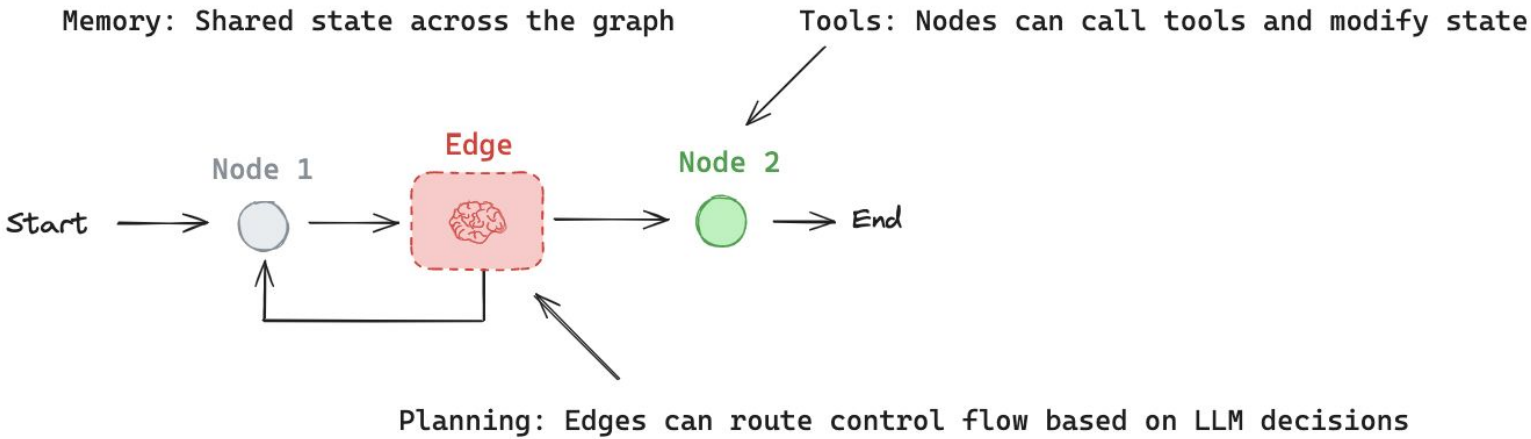


Flexible

Less reliable

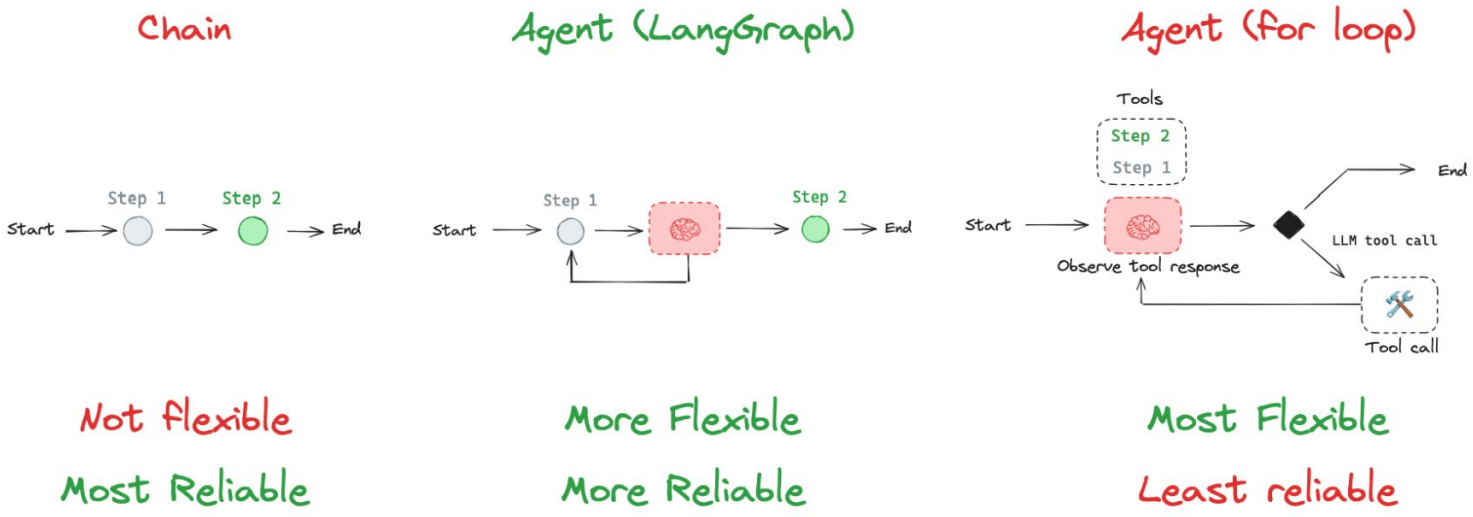
A Controllability

LangGraph Agent



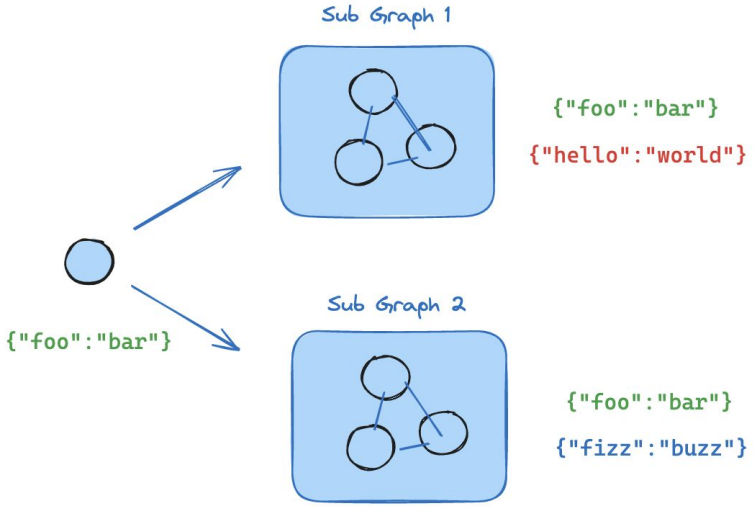
A Controllability

LangGraph allows for developer + LLM-defined control flows



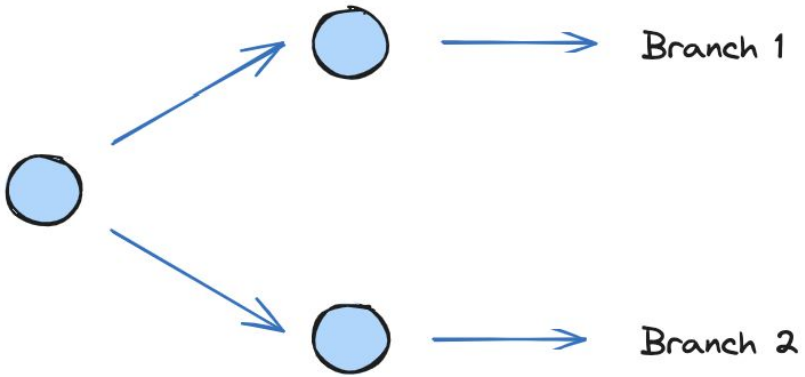
A Controllability

Subgraphs enable complex system design by managing states separately



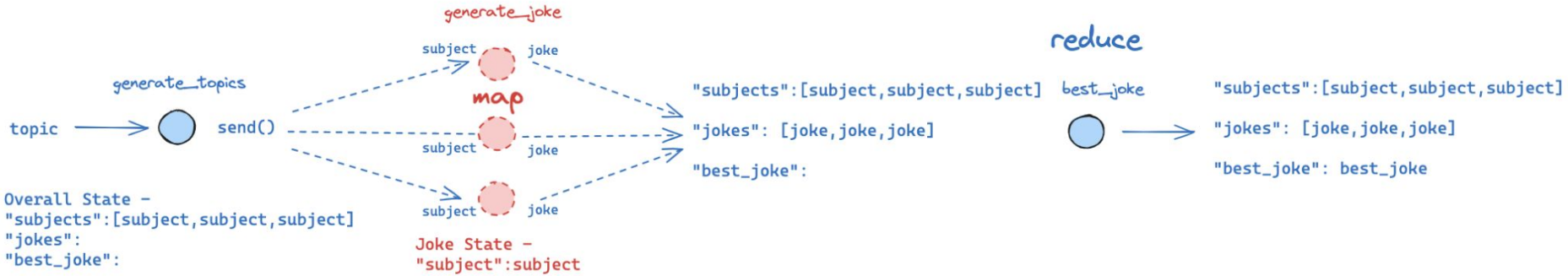
A Controllability

Branches enable parallel execution of nodes to speed up overall graph operation



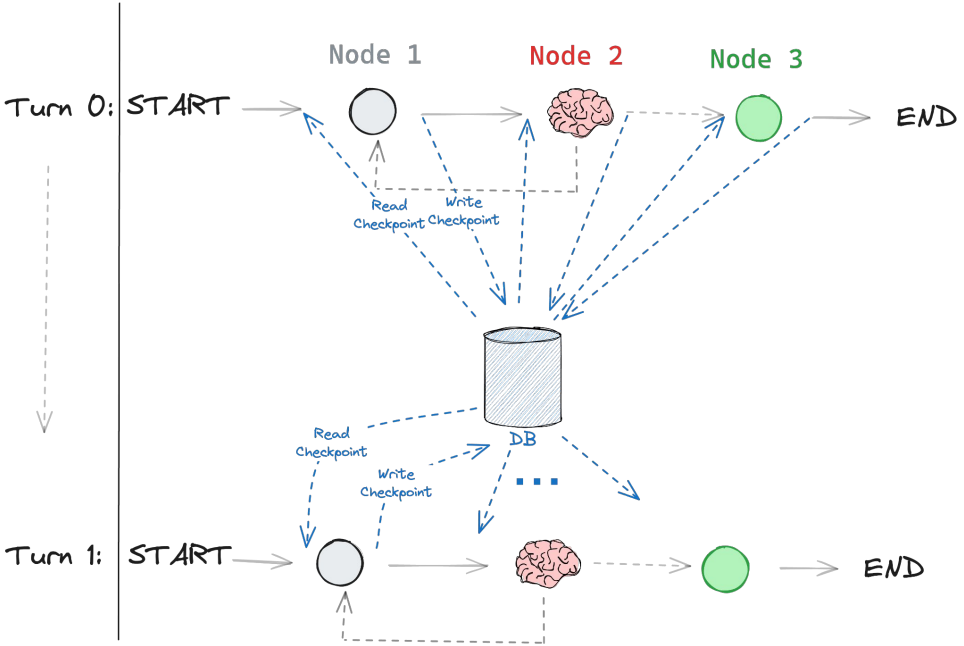
A Controllability

Map-reduce branches enable efficient parallel processing and flexible execution



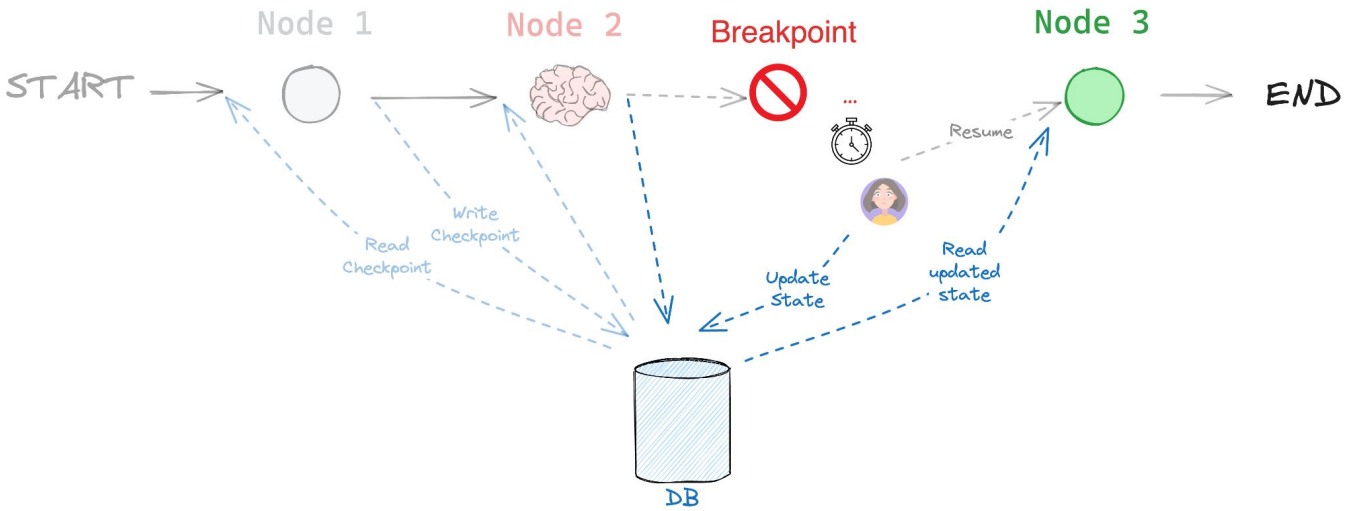
B Persistence

Provides “Memory”



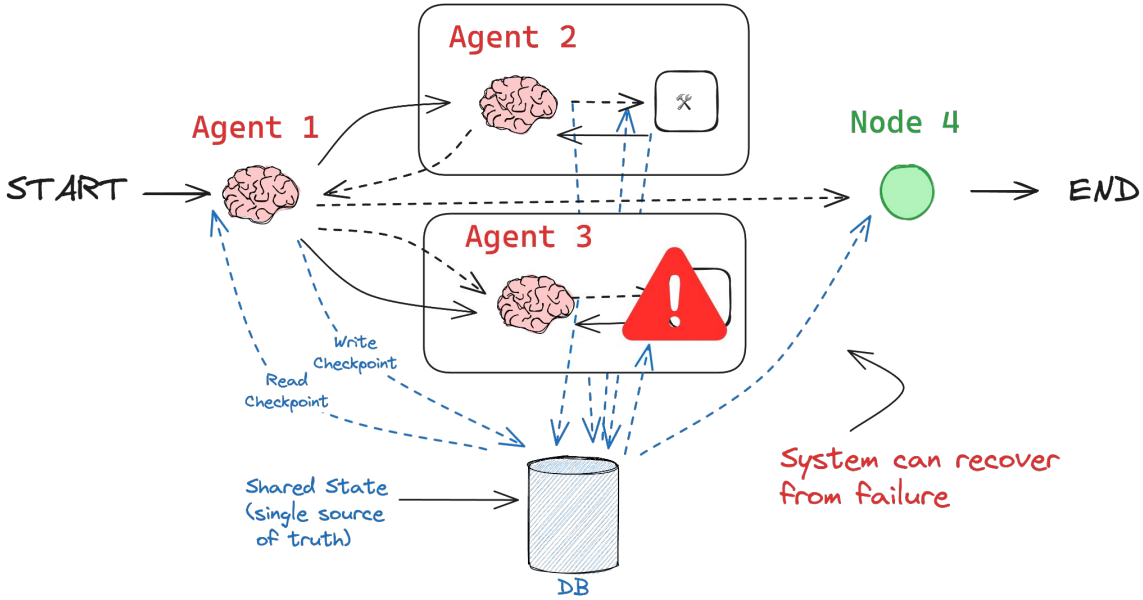
B Persistence

Enables human-agent interactions



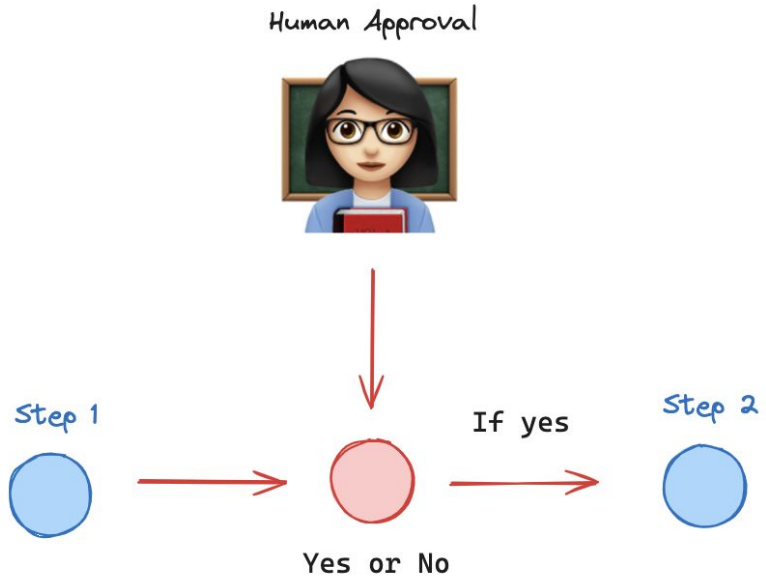
B Persistence

Enables fault-tolerant multi-agent interactions



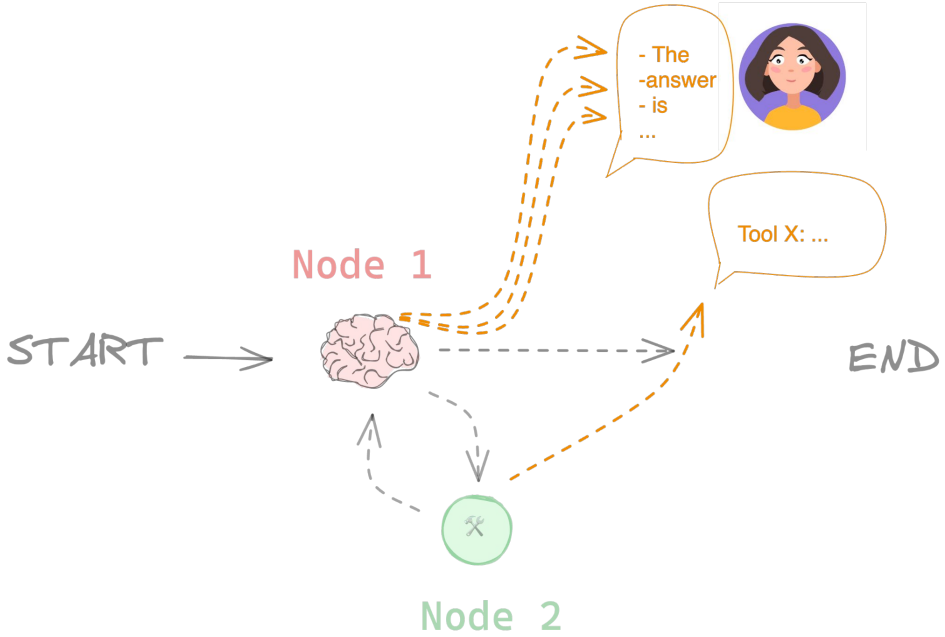
C Human-in-the-loop

Breakpoints enable human-in-the-loop interactions

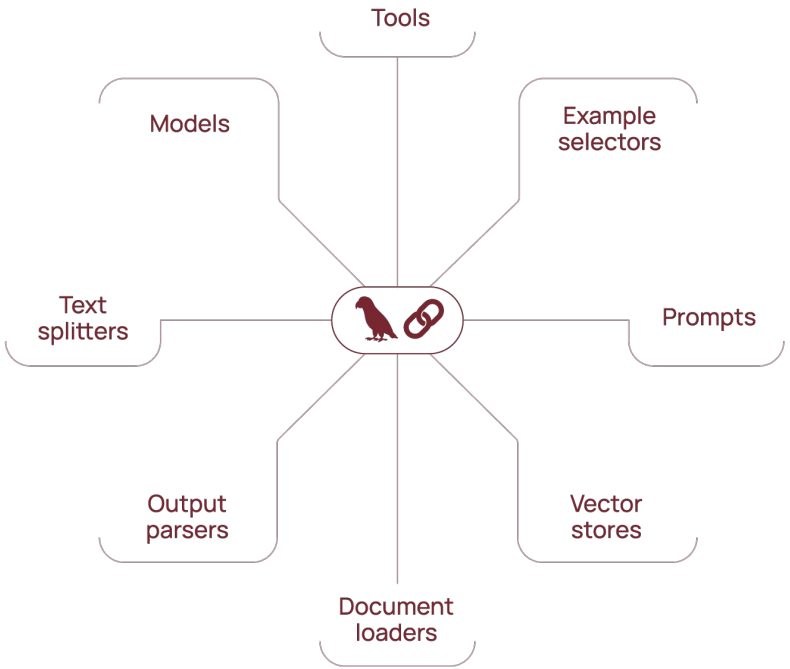


D Streaming

First-class support for token and event-level streaming



3 LangGraph within LangChain's Ecosystem



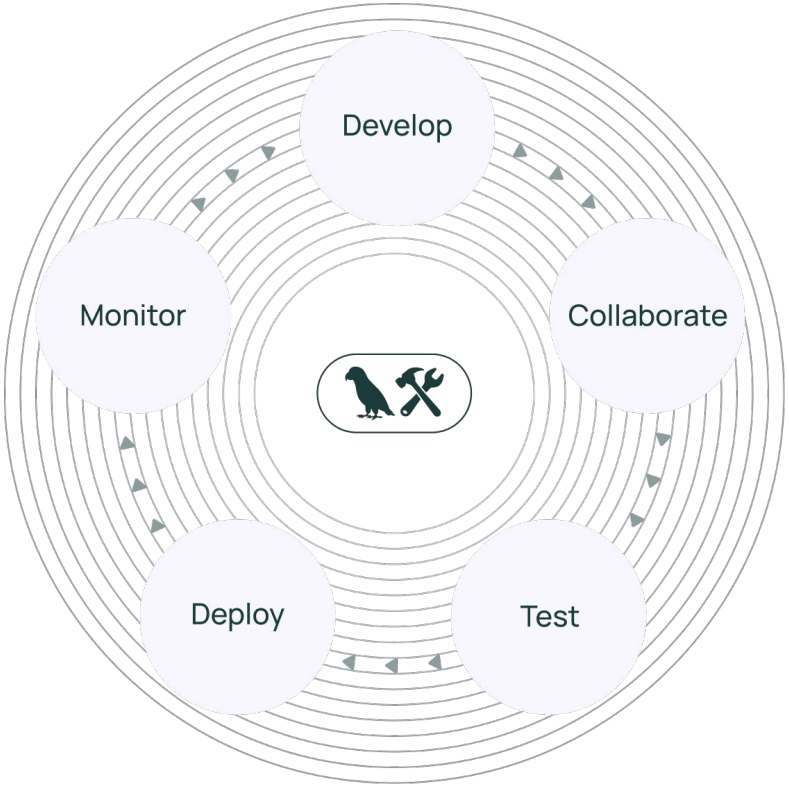
LangChain

LangChain's AI abstractions and integrations make it the #1 choice for developers when building with GenAI

15M+ Monthly Downloads

100K+ Apps Powered

3K+ Contributors



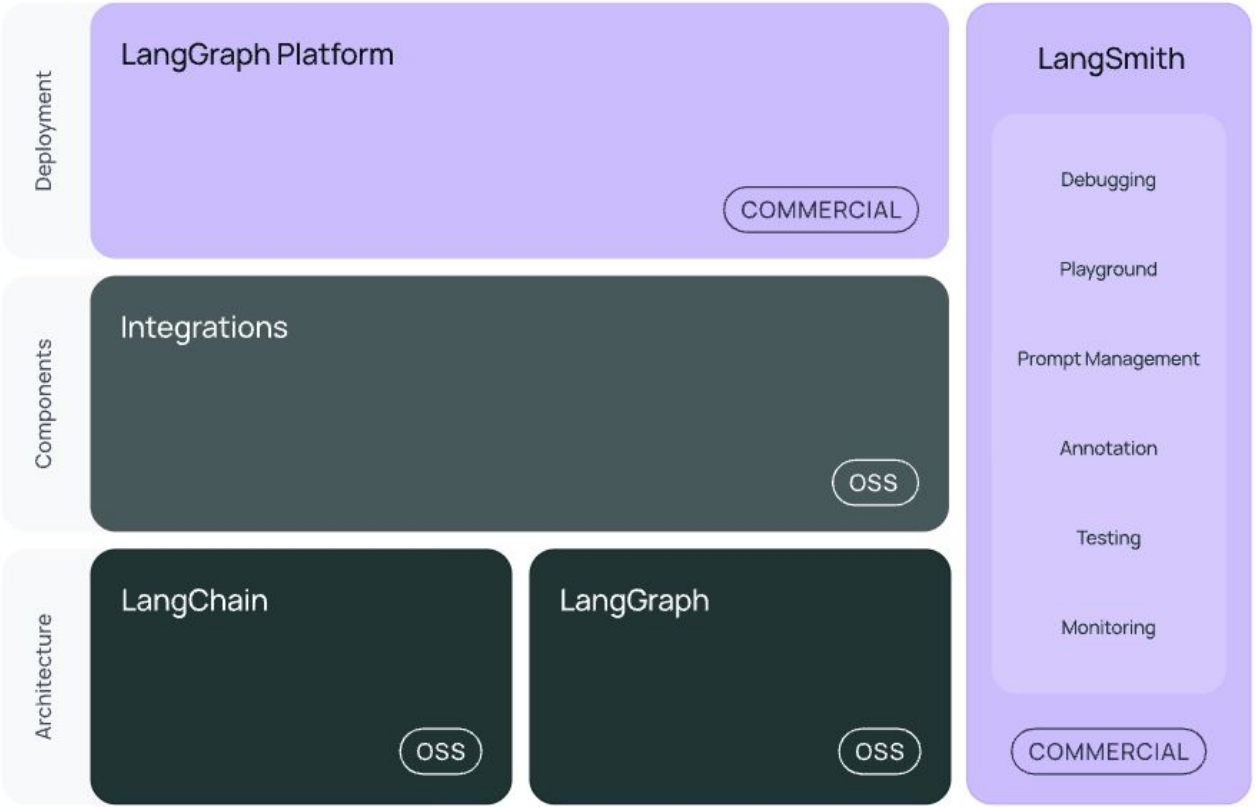
LangSmith

LangSmith is a unified DevOps platform, purpose-built for LLM applications

100K+
Users signed up

300M+
Traces logged

30K+
Monthly active teams

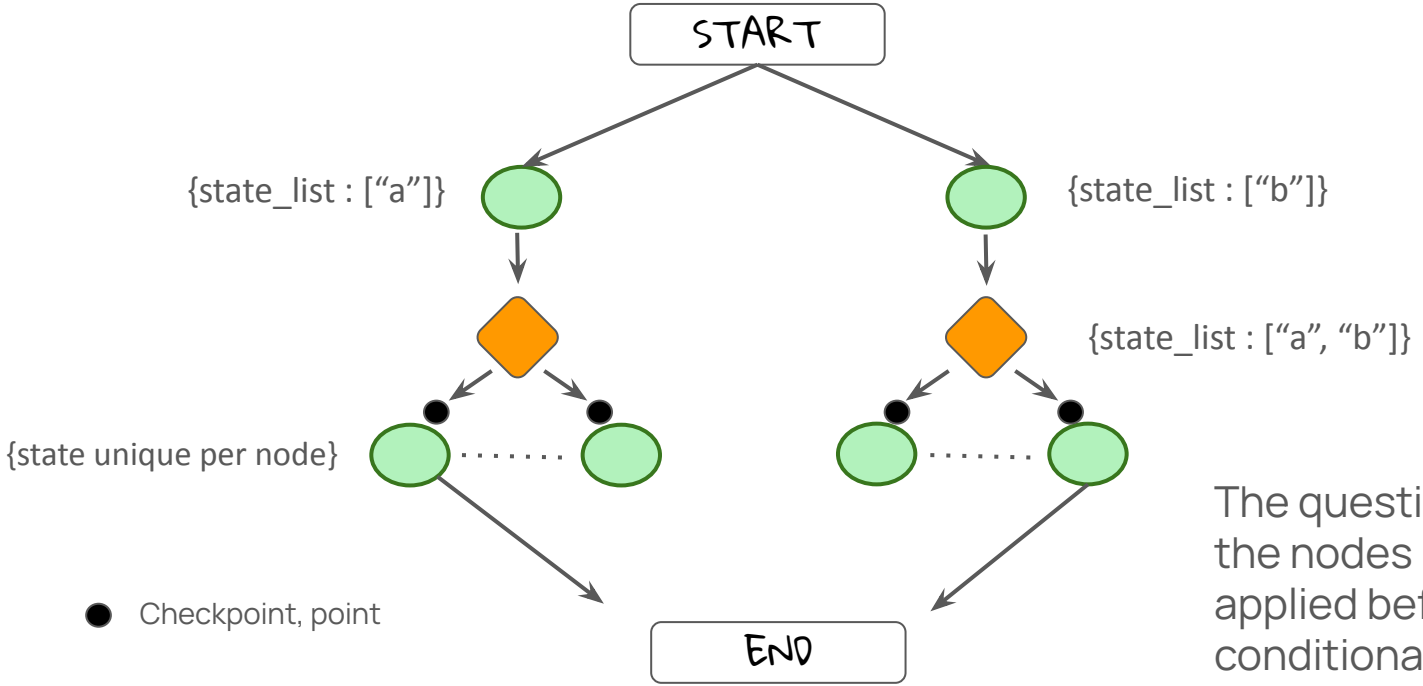

Deployment
LangGraph Platform
COMMERCIAL
Components
Integrations
OSS
Architecture
LangChain
OSS
LangGraph
OSS
LangSmith
Debugging
Playground
Prompt Management
Annotation
Testing
Monitoring
COMMERCIAL

REMOVE IF YOU DO NOT WANT TO INCLUDE A CODE DEMO - THIS IS LIKELY +15 MINUTES

Code Demo

https://github.com/vbarda/pandas-rag-langgraph/blob/6ebabb3c87fd4a494da7bf5c62e62aabadee4146/demo.ipynb

Q&A



The question: the reducer on the nodes in the superstep is applied before considering the conditional edge