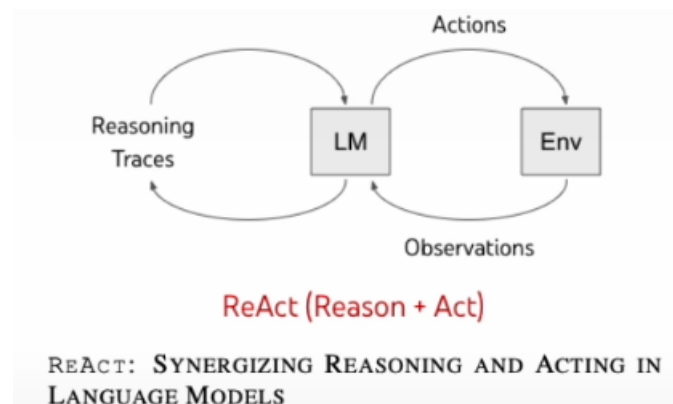


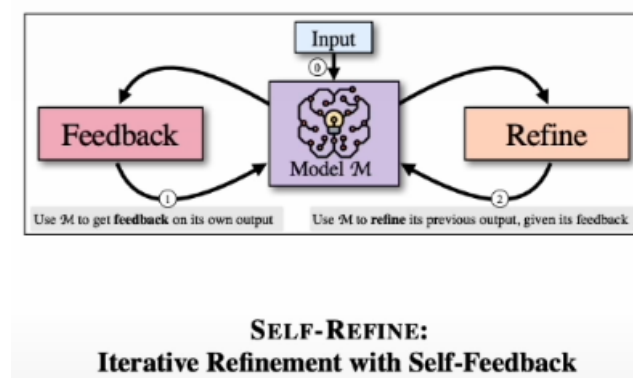
# Course 4 - AI Agents in LangGraph

## Cyclic graph based agents

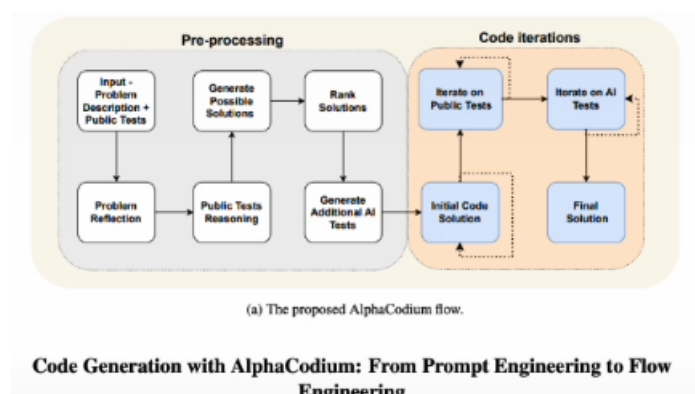
- ReAct - reasoning and actions



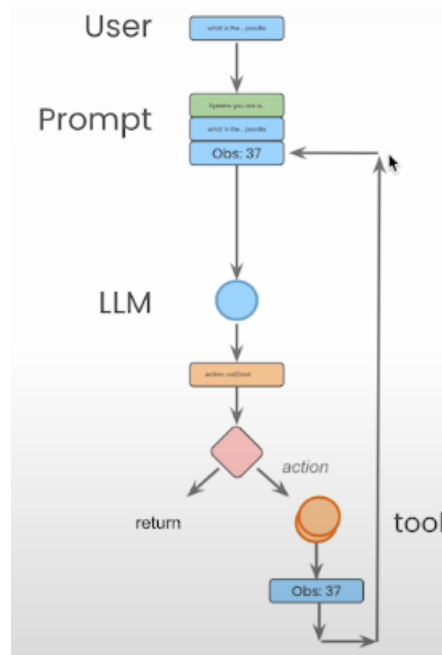
- iterative refinement



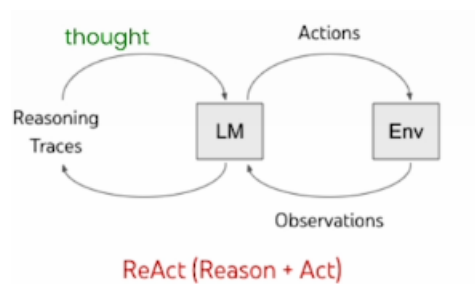
- Flow engineering



## Build an agent from scratch



- ReAct
  - Thought → action → observation



```
class Agent:
    def __init__(self, system=""):
        self.system = system
        self.messages = []
        if self.system:
            self.messages.append({"role": "system", "content": system})

    def __call__(self, message):
        self.messages.append({"role": "user", "content": message})
        result = self.execute()
        self.messages.append({"role": "assistant", "content": result})
        return result

    def execute(self):
        completion = client.chat.completions.create(
            model="gpt-4o",
            temperature=0,
```

```

        messages=self.messages)
    return completion.choices[0].message.content

action_re = re.compile('^Action: (\w+): (.*?)$')
def query(question, max_turns=5):
    i = 0
    bot = Agent(prompt)
    next_prompt = question
    while i < max_turns:
        i += 1
        result = bot(next_prompt)
        print(result)
        actions = [
            action_re.match(a)
            for a in result.split('\n')
            if action_re.match(a)
        ]
        if actions:
            # There is an action to run
            action, action_input = actions[0].groups()
            if action not in known_actions:
                raise Exception("Unknown action: {}: {}".format(action, action_input))
            print(" -- running {} {}".format(action, action_input))
            observation = known_actions[action](action_input)
            print("Observation:", observation)
            next_prompt = "Observation: {}".format(observation)
        else:
            return

```

```

prompt = """
You run in a loop of Thought, Action, PAUSE, Observation.
At the end of the loop you output an Answer
Use Thought to describe your thoughts about the question you have been asked.
Use Action to run one of the actions available to you - then return PAUSE.
Observation will be the result of running those actions.

```

Your available actions are:

calculate:

e.g. calculate: 4 \* 7 / 3

Runs a calculation and returns the number - uses Python so be sure to use float

average\_dog\_weight:

e.g. average\_dog\_weight: Collie

returns average weight of a dog when given the breed

Example session:

Question: How much does a Bulldog weigh?  
Thought: I should look the dogs weight using average\_dog\_weight  
Action: average\_dog\_weight: Bulldog  
PAUSE


You will be called again with this:

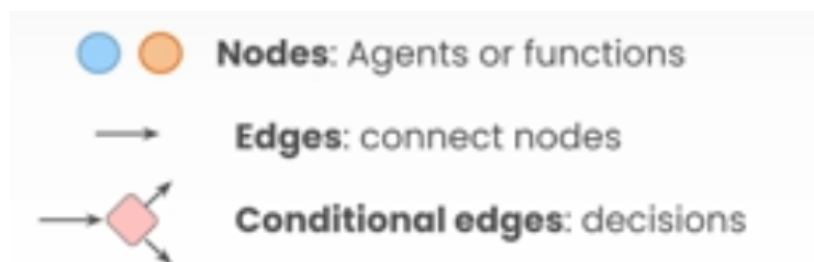
Observation: A Bulldog weights 51 lbs

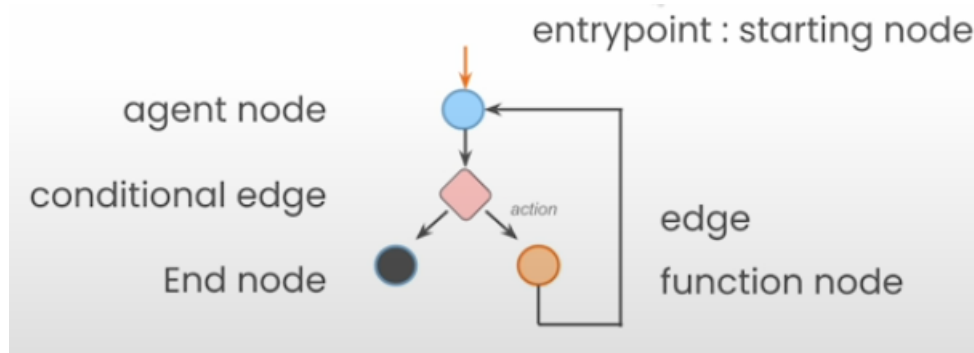
You then output:

Answer: A bulldog weights 51 lbs  
"".strip()

## Langgraph components

- prompt hub → [LangSmith \(langchain.com\)](https://langchain.com).
- Tools → <https://python.langchain.com/v0.1/docs/integrations/tools/>
- Tool kits → [Toolkits](#) |  [LangChain](#)
- langgraph concepts
  - cyclic graphs are created
  - also comes with build in persistence
    - having multiple conversation at a time
    - or remembering previous conversations
  - hum-in-the-loop
- Basic components in a lang graph





- Agent state → tracked over time
  - accessible at all parts of the graph, at each node and at each edge
  - it is local to the graph and can be stored in a persist layer
  - simple state → only have the sequence of messages

```
class AgentState(TypedDict):
    messages: Annotated[list[AnyMessage], operator.add]
```

- complex state → input, chat\_history, agent\_outcome, and intermediate\_steps

```
from langgraph.graph import StateGraph, END
from langchain_core.messages import AnyMessage, SystemMessage, HumanMessage,

class Agent:

    def __init__(self, model, tools, system=""):
        self.system = system
        graph = StateGraph(AgentState)
        graph.add_node("llm", self.call_openai) -> llm node
        graph.add_node("action", self.take_action) -> action node
        graph.add_conditional_edges(
            "llm", -> edge starts node
            self.exists_action, -> function that will
            {True: "action", False: END} -> map the r
            ) -> conditional node

        graph.add_edge("action", "llm") -> connecting from action back to llm
        graph.set_entry_point("llm") -> entry point of the graph
        self.graph = graph.compile() -> to create a langchain runnable
        self.tools = {t.name: t for t in tools}
        self.model = model.bind_tools(tools) -> letting the model know we hav

    def exists_action(self, state: AgentState):
        ...

    def take_action(self, state: AgentState):
```

```

...

def call_openai(self, state: AgentState):
    ...

```

- Draw the graph

```

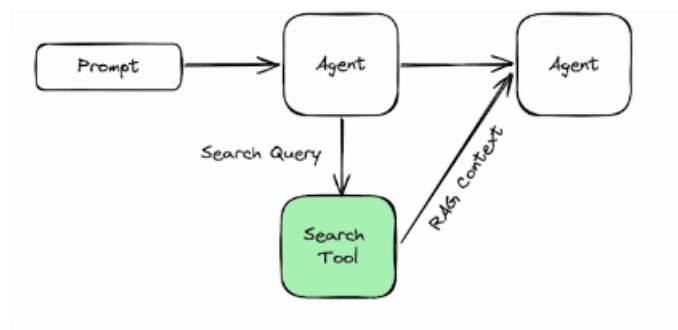
from IPython.display import Image

Image(abot.graph.get_graph().draw_png())

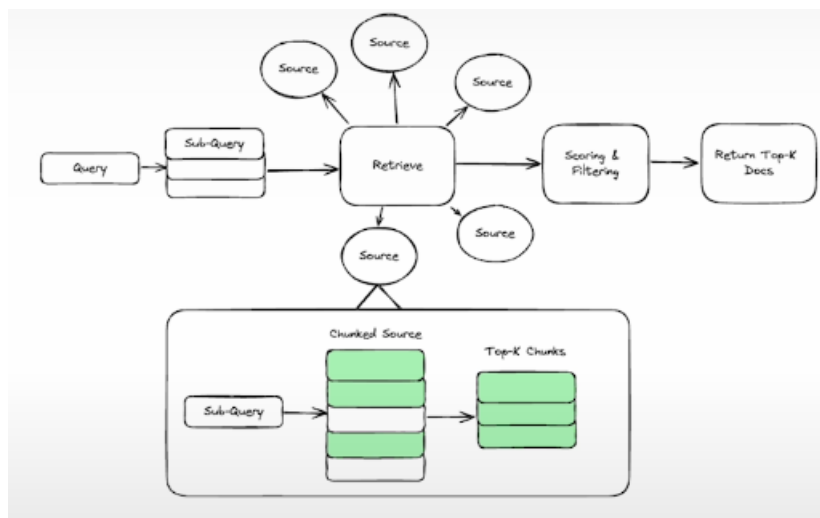
```

## Agentic search tools

- why search tool?
- Agent level diagram



- Search tool breakdown



- Queries are converted to sub-queries if we require to break down a complex task
- then for each sub-query the search tool will have find the best source
- the search tool select only the relevant information by selecting the top-k chunks, based on a vector search (can be different approach)
- the search tool then score the results and filter out the less relevant information

- Tavily agents

```
from tavily import TavilyClient
client = TavilyClient(api_key=os.environ.get("TAVILY_API_KEY"))
result = client.search(query, max_results=1)
data = result["results"][0]["content"]
```

- print json as pretty thing

```
import json
from pygments import highlight, lexers, formatters
# parse JSON
parsed_json = json.loads(data.replace("'", '"'))

# pretty print JSON with syntax highlighting
formatted_json = json.dumps(parsed_json, indent=4)
colorful_json = highlight(formatted_json,
                           lexers.JsonLexer(),
                           formatters.TerminalFormatter())

print(colorful_json)
```

## persistence and streaming

- longer running tasks
  - persistence → keep the state of an agent to resume in future iterations
  - streaming → emit list of signals of what is going on in that exact moment
- persistence (in memory persistence for now)
  - Using sql server for now
  - redis or postgre can also be used.
  - we can also connect this to an external database

```
from langgraph.checkpoint.sqlite import SqliteSaver
memory = SqliteSaver.from_conn_string(":memory:")

class Agent:
    def __init__(self, model, tools, checkpointer, system=""):
        ...
        self.graph = graph.compile(checkpointer=checkpointer)
        ...

abot = Agent(model, [tool], system=prompt, checkpointer=memory)
```

- Streaming

```
for event in abot.graph.stream({"messages": messages}, thread):
    for v in event.values():
        print(v['messages'])
```

- individual messages → AI message, what action to take
- observation message → results of taking that action
- stream the token that are generated

```
from langgraph.checkpoint.aiosqlite import AsyncSqliteSaver

memory = AsyncSqliteSaver.from_conn_string(":memory:")
abot = Agent(model, [tool], system=prompt, checkpoint=memory)

messages = [HumanMessage(content="What is the weather in SF?")]
thread = {"configurable": {"thread_id": "4"}}
async for event in abot.graph.astream_events({"messages": messages}, th
    kind = event["event"]
    if kind == "on_chat_model_stream":
        content = event["data"]["chunk"].content
        if content:
            # Empty content in the context of OpenAI means
            # that the model is asking for a tool to be invoked.
            # So we only print non-empty content
            print(content, end="|")
```

- thread config

```
thread = {"configurable": {"thread_id": "any_string"}}
```

- used to track of different threads inside the persistence checkpoint
- allow us to have multiple conversation going on at the same time

## Human in loop

- We add an interrupt at before the actions node to take in input

```
def reduce_messages(left: list[AnyMessage], right: list[AnyMessage]) -> li
    # assign ids to messages that don't have them
    for message in right:
        if not message.id:
            message.id = str(uuid4())
    # merge the new messages with the existing messages
    merged = left.copy()
    for message in right:
        for i, existing in enumerate(merged):
            # replace any existing messages with the same id
            if existing.id == message.id:
```



```

        merged[i] = message
        break
    else:
        # append any new messages to the end
        merged.append(message)
    return merged

class AgentState(TypedDict):
    messages: Annotated[list[AnyMessage], reduce_messages]

```

- can also be done for a specific tool call, given in documentation.

```

self.graph = graph.compile(
    checkpoint=checkpointer,
    interrupt_before=["action"]) #the interrupt happend before action

```

- Graph state
  - Current state

```

abot.graph.get_state(thread) # current state of the graph
abot.graph.get_state(thread).next # the next node in the graph

```

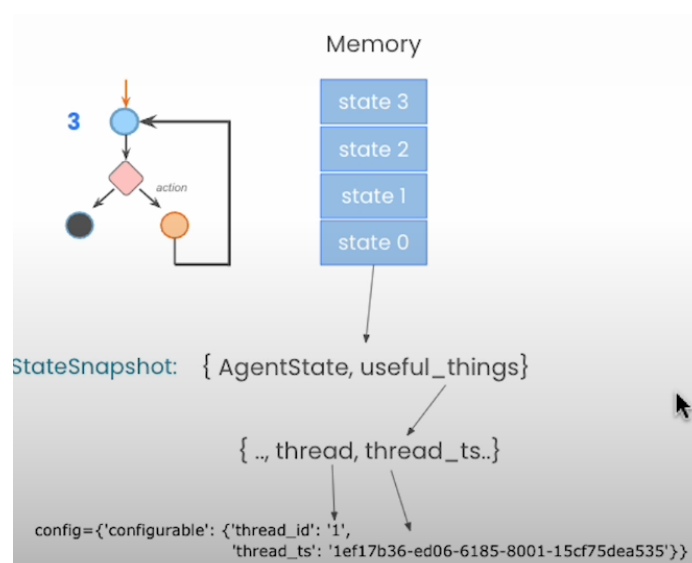
- To run the next step in graph after the interrupt

```

for event in abot.graph.stream(None, thread):
    for v in event.values():
        print(v)

```

- State memory



- As the graph is executing the snapshot of each state is stored in memory.

```
graph.get_state(thread) # by default give the current running state using
graph.get_state_history(thread)
```

- Modify state

- we change the user query in this particular case of modification

```
thread = {"configurable": {"thread_id": "3"}}
current_values = abot.graph.get_state(thread)
_id = current_values.values['messages'][-1].tool_calls[0]['id']
current_values.values['messages'][-1].tool_calls = [
    {'name': 'tavily_search_results_json',
     'args': {'query': 'current weather in Louisiana'},
     'id': _id}
]
abot.graph.update_state(thread, current_values.values)
```

- Time Travel

- get all the states present in the memory
- and we try to use an older state that we had.
- So each state is preserved with in the history of that thread and we can go to that state or use the older states to modify and start the graph from there.

```
states = []
for state in abot.graph.get_state_history(thread):
    print(state)
    print('--')
    states.append(state)

# states[-3] is the oldest state present in the system
for event in abot.graph.stream(None, states[-3].config):
    for k, v in event.items():
        print(v)
```

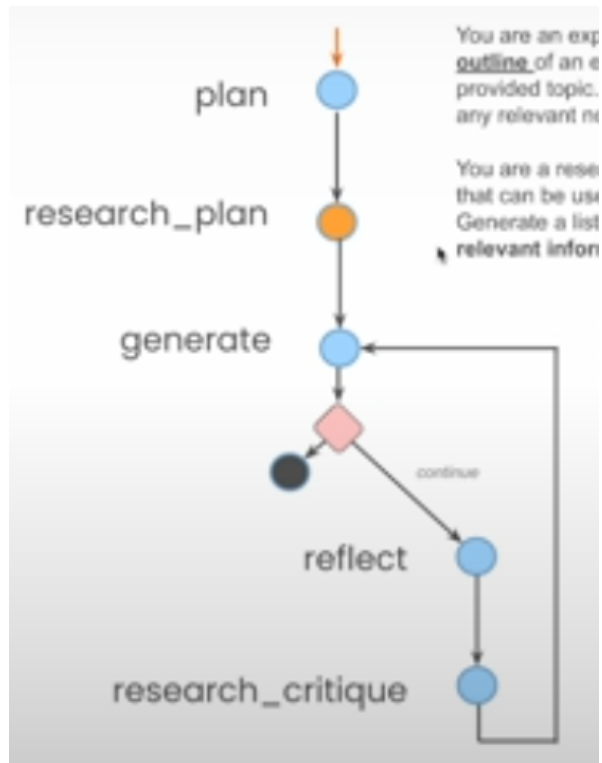
- Add messages

- Add custom messages with in the state
- Here `as_node="action"` to let the graph know this is acting as a proxy action

```
_id = to_replay.values['messages'][-1].tool_calls[0]['id']
state_update = {"messages": [ToolMessage(
    tool_call_id=_id,
    name="tavily_search_results_json",
    content="54 degree celcius",
)]]
branch_and_add = abot.graph.update_state(
    to_replay.config,
```

```
state_update,  
as_node="action")
```

## essay writer

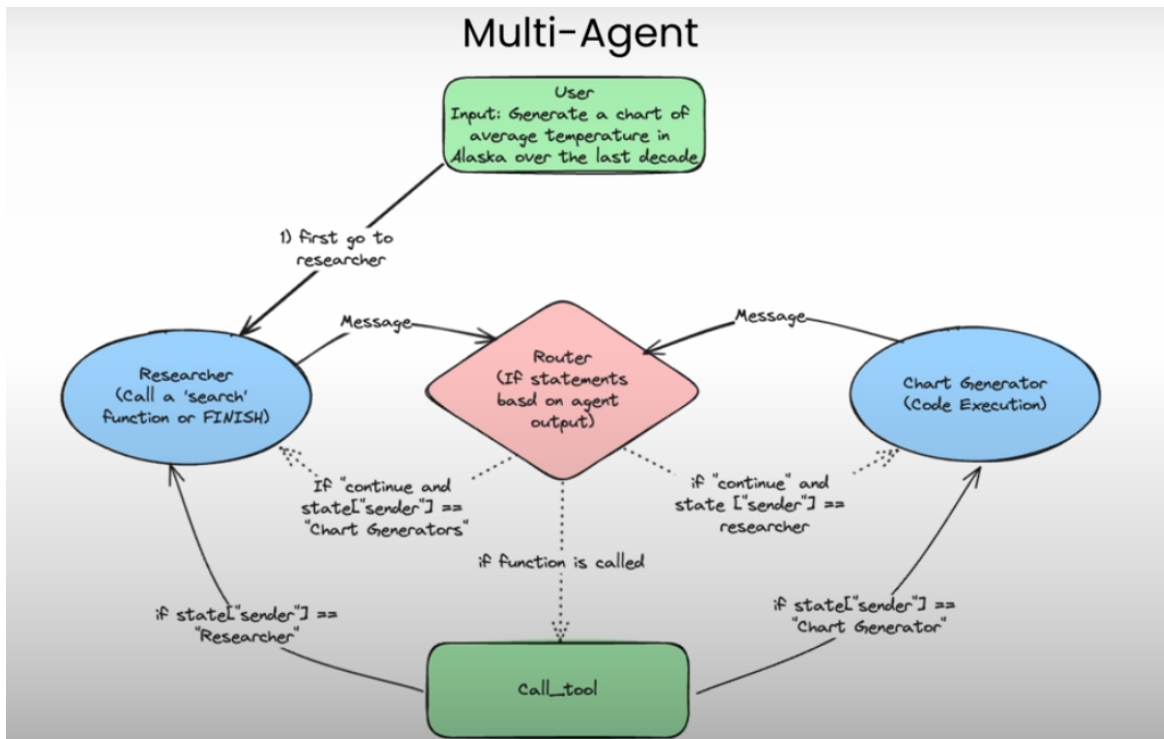


## Recourses

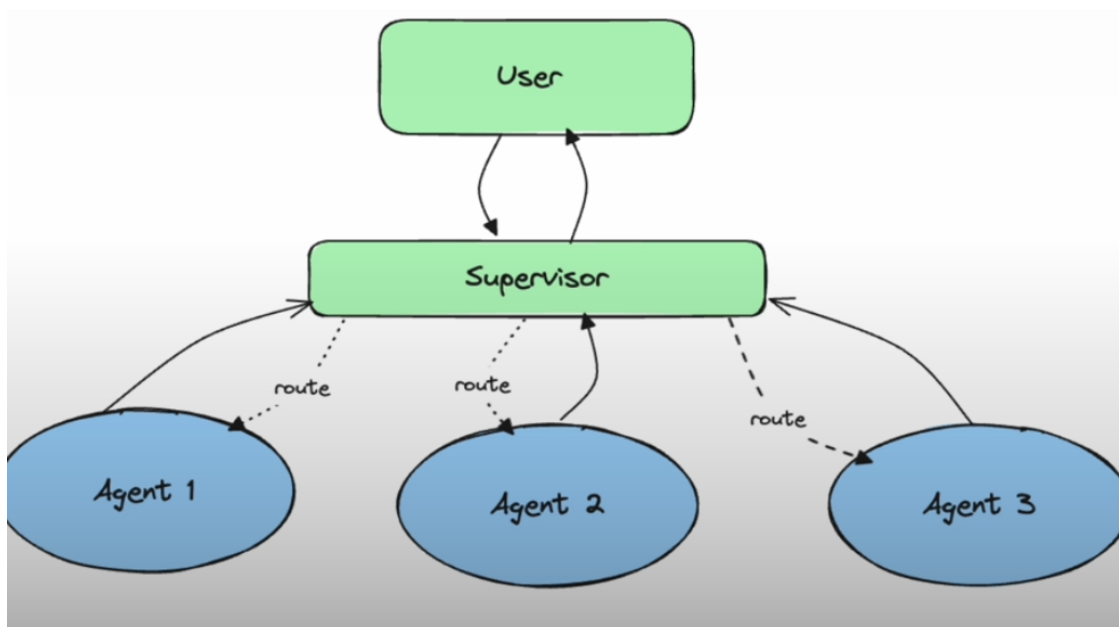
- This is only the basics ok.
- Streaming
  - [Streaming](#) | [LangChain](#)
- <https://python.langchain.com/v0.2/docs/introduction/>
  - <https://github.com/langchain-ai/langchain>
- <https://langchain-ai.github.io/langgraph/>
  - <https://github.com/langchain-ai/langgraph>
- prompt hub
  - [LangSmith \(langchain.com\)](#)
- bedrock support
  - <https://github.com/langchain-ai/langchain-aws>

## Good to know flow

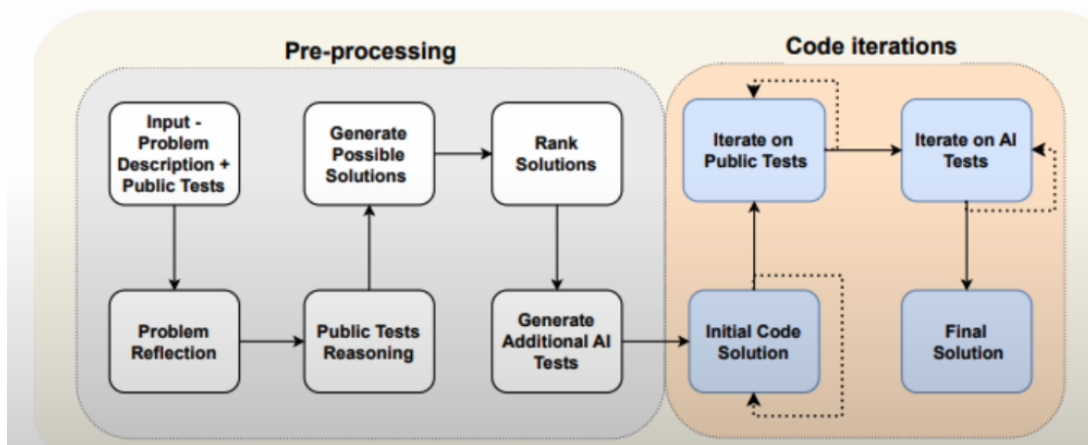
- multi agent architecture → multiple agents work on the same shared state



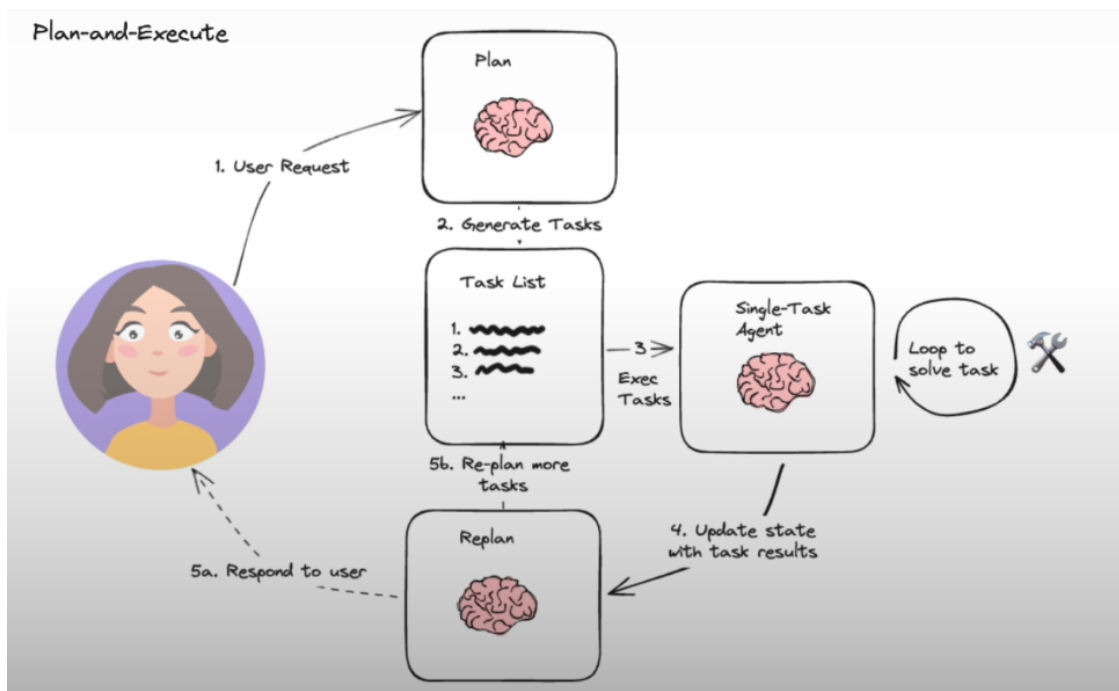
- Supervisor based flows → different states are present in each agent and each is a separate graph. Supervisor is responsible to route the agents



- Flow engineering → extends more broadly and generally → refers to thinking what is the right information flow for your agents to take action and think



- its a pipeline with key nodes having loops
- graph with a singular flow upto a point and then we have iterations
- Plan and execute ([langgraph/examples/plan-and-execute](https://github.com/langchain-ai/langgraph/blob/main/examples/plan-and-execute.py) at main · langchain-ai/langgraph (github.com)).



- Language agent tree search

