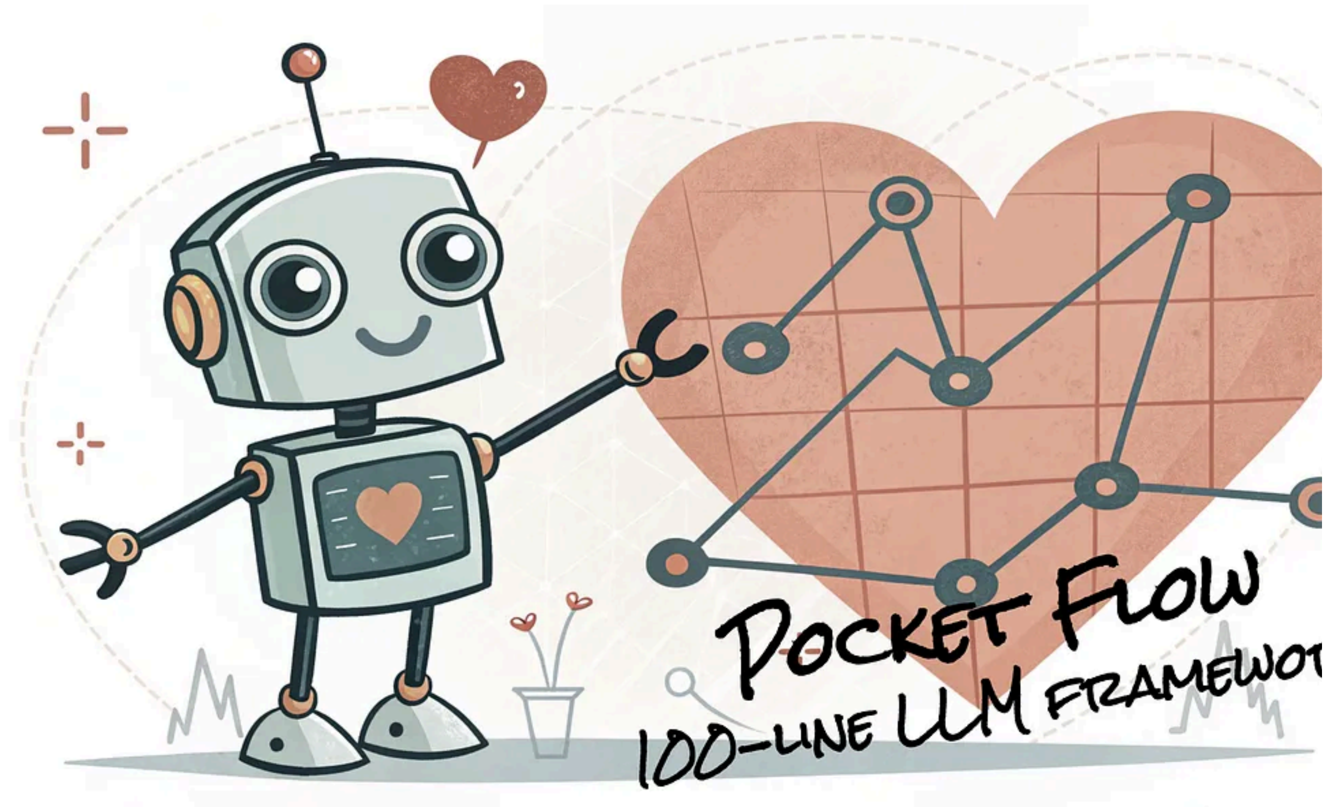
LLM Agents are simply Graph — Tutorial For Dummies

ZACHARY HUANG
MAR 18, 2025

100 7 2 Sh



Ever wondered how AI agents actually work behind the scenes? This guide breaks dow how agent systems are built as simple graphs - explained in the most beginner-friendly possible!

Note: This is a super-friendly, step-by-step version of the official PocketFlow Agent documentation. We explanations to ma

Looks like an article worth saving! Option Q

Hover over the brain icon or use hotkeys to save with Memex.

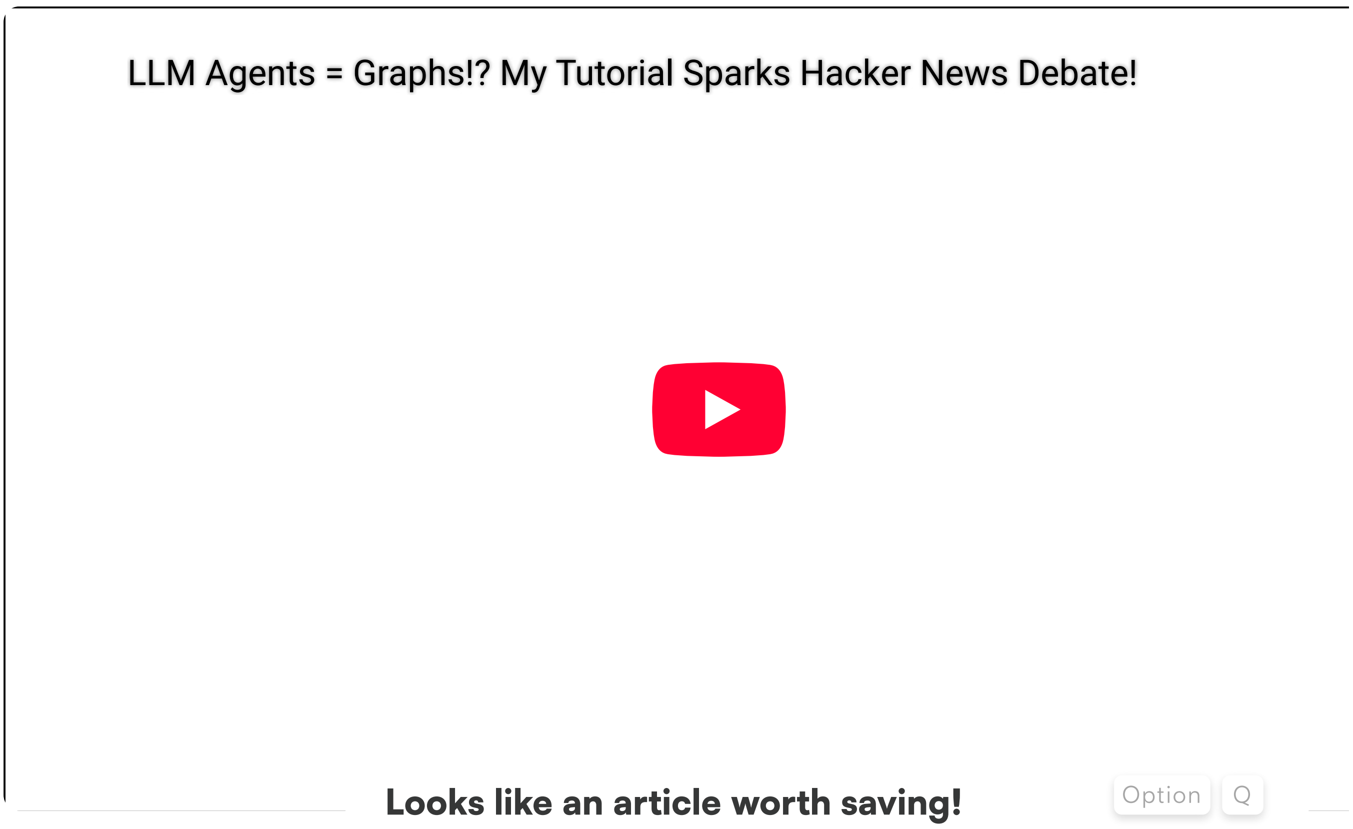
Remind me later Hide Forever

Have you been heari technical jargon? Yo sin complex AI agents like GitHub Copilot, PerplexityAI, and AutoGPT, most explanations make them sound like rocket science.

Good news: they're not. In this beginner-friendly guide, you'll learn:

- The surprisingly simple concept behind all AI agents
- How agents actually make decisions (in plain English!)
- How to build your own simple agent with just a few lines of code
- Why most frameworks overcomplicate what's actually happening

Check out my YouTube Video on this topic:


LLM Agents = Graphs!? My Tutorial Sparks Hacker News Debate!

In this tutorial, we'll ay
the complexity to sh er

frameworks that hide the important details, PocketFlow lets you see the entire sys at once.

Why Learn Agents with PocketFlow?

Most agent frameworks hide what's really happening behind complex abstraction: that look impressive but confuse beginners. PocketFlow takes a different approacl it's just 100 lines of code that lets you see exactly how agents work!

Benefits for beginners:

- **Crystal clear**: No mysterious black boxes or complex abstractions
- **See everything**: The entire framework fits in one readable file
- **Learn fundamentals**: Perfect for understanding how agents really operate
- **No baggage**: No massive dependencies or vendor lock-in

Instead of trying to understand a gigantic framework with thousands of files, PocketFlow gives you the fundamentals so you can build your own understanding the ground up.

The Simple Building Blocks

Imagine our agent system like a kitchen:

- Nodes are like different cooking stations (chopping station, cooking station, plating station)
- Flow is like the recipe that tells you which station to go to next
- Shared store is like the big countertop where everyone can see and use the ingredients

In our kitchen (agent

1. Each station (No
    - **Prep**: Grab what you need from the countertop (like getting ingredier

- **Exec**: Do your special job (like cooking the ingredients)
- **Post**: Put your results back on the countertop and tell everyone where to g next (like serving the dish and deciding what to make next)

2. The recipe (Flow) just tells you which station to visit based on decisions:
   - "If the vegetables are chopped, go to the cooking station"
   - "If the meal is cooked, go to the plating station"

Let's see how this works with our research helper!

What's an LLM Agent (In Human Terms)?

An LLM (Large Language Model) agent is basically a smart assistant (like ChatGP but with the ability to take actions) that can:

1. Think about what to do next
2. Choose from a menu of actions
3. Actually do something in the real world
4. See what happened
5. Think again...

Think of it like having a personal assistant managing your tasks:

1. They review your inbox and calendar to understand the situation
2. They decide what needs attention first (reply to urgent email? schedule a meeting?)
3. They take action (draft a response, book a conference room)
4. They observe the
5. They plan the ne

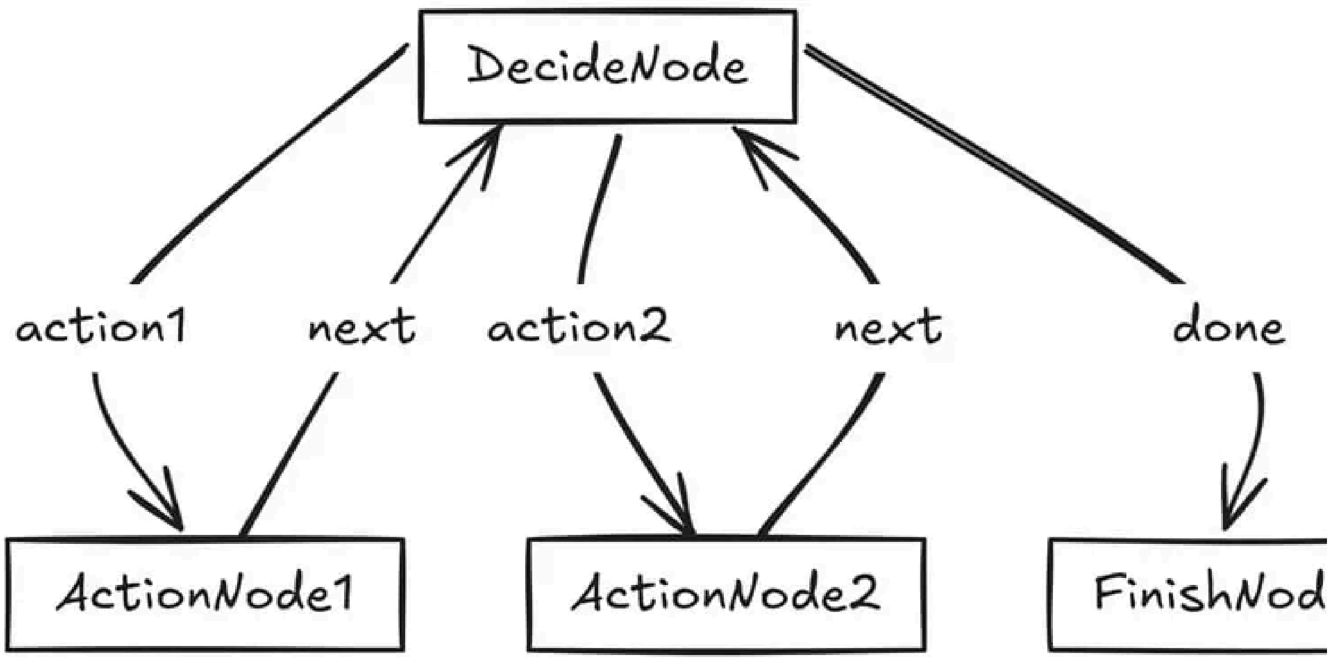
Looks like an article worth saving! Option Q

Hover over the brain icon or use hotkeys to save with Memex.

Remind me later Hide Forever

The Big Secr s!

Here's the mind-blowing truth about agents that frameworks overcomplicate



That's it! Every agent is just a graph with:

1. A **decision node** that branches to different actions
2. **Action nodes** that do specific tasks
3. A **finish node** that ends the process
4. **Edges** that connect everything together
5. **Loops** that bring execution back to the decision node

No complex math, no mysterious algorithms - just nodes and arrows! Everything e is just details. If you dig deeper, you'll uncover these hidden graphs in overcompli frameworks:

- **OpenAI Agents**: run.py#L119 for a workflow in graph.
- **Pydantic Agents**: _agent_graph.py#L779 organizes steps in a graph.
- **Langchain**: agent iterator.py#L174 demonstrates the loop structure
- **LangGraph**: age

Let's see how the gra

Looks like an article worth saving!

Option Q

Hover over the brain icon or use hotkeys to save with Memex.

Remind me later Hide Forever

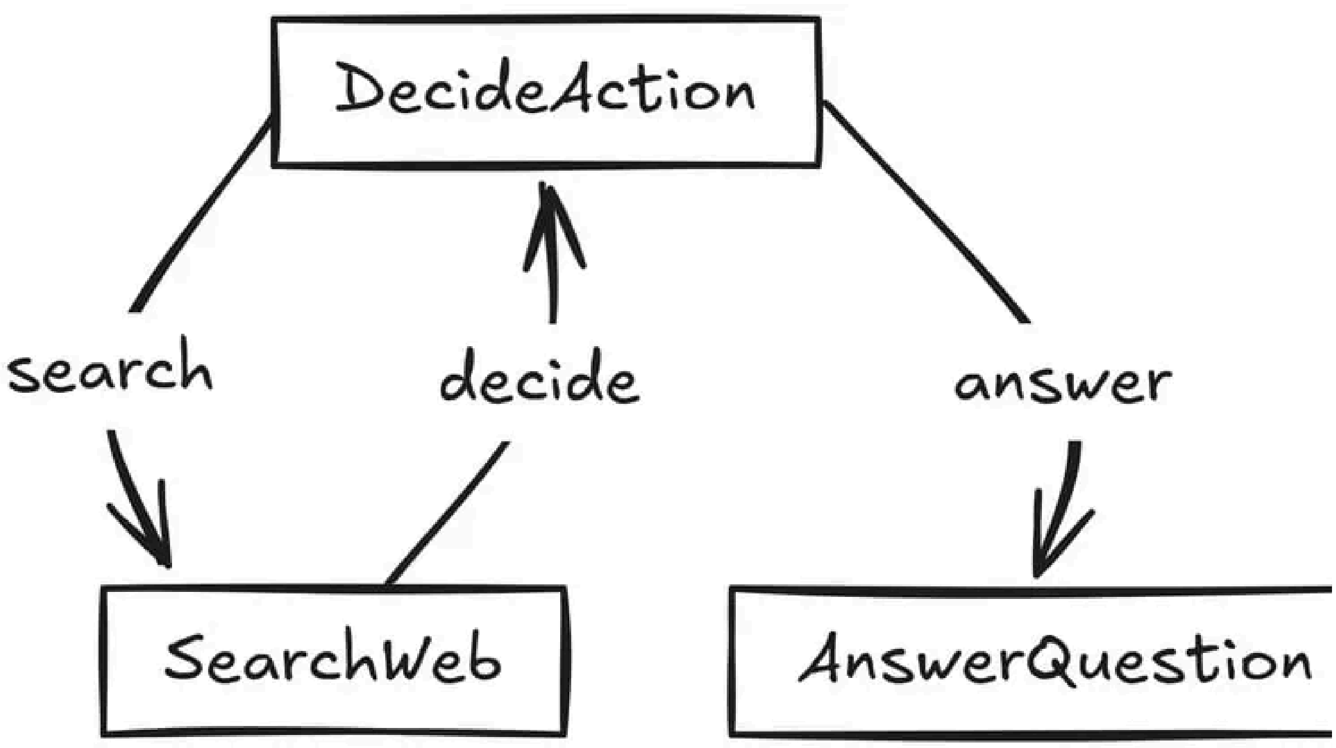
Let's Build a Super Simple Research Agent

Imagine we want to build an AI assistant that can search the web and answer questions - similar to tools like Perplexity AI, but much simpler. We want our ager be able to:

1. Read a question from a user
2. Decide if it needs to search for information
3. Look things up on the web if needed
4. Provide an answer once it has enough information

Let's break down our agent into individual "stations" that each handle one specifi job. Think of these stations like workers on an assembly line - each with their owr specific task.

Here's a simple diagram of our research agent:



In this diagram:

1. **DecideAction** is ne
2. **SearchWeb** is ou

3. **AnswerQuestion** is our "response station" where the agent creates the final answer

Before We Code: Let's Walk Through an Example

Imagine you asked our agent: "Who won the 2023 Super Bowl?"

Here's what would happen step-by-step:

1. **DecideAction station**:
   - LOOKS AT: Your question and what we know so far (nothing yet)
   - THINKS: "I don't know who won the 2023 Super Bowl, I need to search"
   - DECIDES: Search for "2023 Super Bowl winner"
   - PASSES TO: SearchWeb station
2. **SearchWeb station**:
   - LOOKS AT: The search query "2023 Super Bowl winner"
   - DOES: Searches the internet (imagine it finds "The Kansas City Chiefs wo
   - SAVES: The search results to our shared countertop
   - PASSES TO: Back to DecideAction station
3. **DecideAction station (second time)**:
   - LOOKS AT: Your question and what we know now (search results)
   - THINKS: "Great, now I know the Chiefs won the 2023 Super Bowl"
   - DECIDES: We have enough info to answer
   - PASSES TO: AnswerQuestion station
4. **AnswerQuestion**
   - LOOKS AT:
   - DOES: Creat
   - SAVES: The
   - FINISHES: The task is complete!

Looks like an article worth saving! Option Q

Hover over the brain icon or use hotkeys to save with Memex.

Remind me later Hide Forever

This is exactly what our code will do - just expressed in programming language.

Let's build each of these stations one by one and then connect them together!

Step 1: Building Our First Node: DecideAction 🤔

The **DecideAction** node is like the "brain" of our agent. Its job is simple:

1. Look at the question and any information we've gathered so far
2. Decide whether we need to search for more information or if we can answer n

Let's build it step by step:

```
class DecideAction(Node):
    def prep(self, shared):
        # Think of "shared" as a big notebook that everyone can read a
write in
        # It's where we store everything our agent knows

        # Look for any previous research we've done (if we haven't
searched yet, just note that)
        context = shared.get("context", "No previous search")

        # Get the question we're trying to answer
        question = shared["question"]

        # Return both pieces of information for the next step
        return question, context
```

First, we created a `prep` method that gathers information. Think of `prep` like a cl gathering ingredients before cooking. All it does is look at what we already know "context") and what c

Now, let's build the "

Looks like an article worth saving!

Hover over the brain icon or use hotkeys to save with Memex.

Option Q

Remind me later Hide Forever

```
    def exec(sel
        # This is where the magic happens - the LLM "thinks" about
```

~~~
to do
        question, context = inputs

        # We ask the LLM to decide what to do next with this prompt:
        prompt = f"""
### CONTEXT
You are a research assistant that can search the web.
Question: {question}
Previous Research: {context}

### ACTION SPACE
[1] search
  Description: Look up more information on the web
  Parameters:
    - query (str): What to search for

[2] answer
  Description: Answer the question with current knowledge
  Parameters:
    - answer (str): Final answer to the question

## NEXT ACTION
Decide the next action based on the context and available actions.
Return your response in this format:

```yaml
thinking: |
    <your step-by-step reasoning process>
action: search OR answer
reason: <why you chose this action>
search_query: <specific search query if action is search>
```"""

        # Call the LLM to make a decision
        response = call_llm(prompt)

        # Pull ou                                                's
answer
        # (This i                                               ide
        yaml_str                                                ip(
        decision                                                o a
format our progra
~~~

```
        return decision
```

The `exec` method is where the actual "thinking" happens. Think of it like the che cooking the ingredients. Here, we:

1. Create a prompt that explains the current situation to the LLM
2. Ask the LLM to decide whether to search or answer
3. Parse the response to get a clear decision

Finally, let's save the decision and tell the flow what to do next:

```
    def post(self, shared, prep_res, exec_res):
        # If the LLM decided to search, save the search query
        if exec_res["action"] == "search":
            shared["search_query"] = exec_res["search_query"]

        # Return which action to take - this tells our flow where to g
next!
        return exec_res["action"]  # Will be either "search" or "answe
```

The `post` method is where we save results and decide what happens next. Think o like the chef serving the food and deciding what dish to prepare next. We save the search query if needed, and then return either "search" or "answer" to tell our flov which node to visit next.

Step 2: Building Our Second Node: SearchWeb 🔍

The **SearchWeb** node is our "researcher." Its only job is to:

1. Take a search qu
2. Look it up (in th
3. Save what it find
4. Tell the agent to decide what to do with this new information

Let's build it:

```
class SearchWeb(Node):
    def prep(self, shared):
        # Simply get the search query we saved earlier
        return shared["search_query"]
```

The `prep` method here just grabs the search query that was saved by the DecideA node.

```
    def exec(self, search_query):
        # This is where we'd connect to Google to search the internet
        # Set up our connection to Google
        search_client = GoogleSearchAPI(api_key="GOOGLE_API_KEY")

        # Set search parameters
        search_params = {
            "query": search_query,
            "num_results": 3,
            "language": "en"
        }

        # Make the API request to Google
        results = search_client.search(search_params)

        # Format the results into readable text
        formatted_results = f"Results for: {search_query}\n"

        # Process each search result
        for result in results:
            # Extract the title and snippet from each result
            formatted_results += f"- {result.title}: {result.snippet}\

        return fo
```

Looks like an article worth saving!

Option Q

Hover over the brain icon or use hotkeys to save with Memex.

Remind me later Hide Forever

The `exec` method cc esu

In a real implementat ke

Google Custom Search API and get your own API key.

```
    def post(self, shared, prep_res, exec_res):
        # Store the search results in our shared whiteboard
        previous = shared.get("context", "")
        shared["context"] = previous + "\n\nSEARCH: " +
shared["search_query"] + "\nRESULTS: " + exec_res

        # Always go back to the decision node after searching
        return "decide"
```

The `post` method saves our search results by adding them to our shared context. '
it returns "decide" to tell our flow to go back to the DecideAction node so it can tl
about what to do with this new information.

Step 3: Building Our Third Node: AnswerQuestion 💬

The **AnswerQuestion** node is our "responder." Its job is simply to:

1. Take all the information we've gathered
2. Create a friendly, helpful answer
3. Save that answer for the user

Let's build it:

```
class AnswerQuestion(Node):
    def prep(self, shared):
        # Get both the original question and all the research we've do
        return shared["question"], shared.get("context", "")
```

The `prep` method ga
collected.

```
    def exec(self, inputs):
        question, context = inputs
        # Ask the LLM to create a helpful answer based on our researcl
        prompt = f"""
### CONTEXT
Based on the following information, answer the question.
Question: {question}
Research: {context}

## YOUR ANSWER:
Provide a comprehensive answer using the research results.
"""
        return call_llm(prompt)
```

The `exec` method asks the LLM to create a helpful answer based on our research.

```
    def post(self, shared, prep_res, exec_res):
        # Save the answer in our shared whiteboard
        shared["answer"] = exec_res

        # We're done! No need to continue the flow.
        return "done"
```

The `post` method saves the final answer and returns "done" to indicate that our fl
is complete.

Step 4: Connecting Everything Together! 🔄

Now for the fun part - we need to connect all our nodes together to create a worki
agent!

```
# First, create :
decide = DecideA(
search = SearchWe
answer = AnswerQu

# Now connect them together - this is where the magic happens!
# Each connection defines where to go based on the action returned by
post()

# If DecideAction returns "search", go to SearchWeb
decide - "search" >> search

# If DecideAction returns "answer", go to AnswerQuestion
decide - "answer" >> answer

# After SearchWeb completes, go back to DecideAction
search - "decide" >> decide

# Create our flow, starting with the DecideAction node
flow = Flow(start=decide)
```

That's it! We've connected our nodes into a complete flow that can:

1. Start by deciding what to do
2. Search for information if needed
3. Loop back to decide if we need more information
4. Answer the question when we're ready

Think of it like a flowchart where each box is a node, and the arrows show where
next based on decisions made along the way.

Let's Run Our Agent! 🚀

Now let's see our agent in action:

```
# Create a shared whiteboard with just a question
shared = {"question": "What is the capital of France?"}

# Run our flow!
flow.run(shared)

# Print the fina
print(shared["ans
```

Here's what happens when our agent runs:

First Round: The Initial Decision 🤔

First, our agent thinks about the question:

```
thinking: |
  The question is asking about the capital of France. I don't have an
prior search results to work with.
  To answer this question accurately, I should search for information
about France's capital.
action: search
reason: I need to look up information about the capital of France
search_query: capital of France
```

Our agent decides it doesn't know enough yet, so it needs to search for informatio

Second Round: Searching the Web 🔍

The SearchWeb node now searches for "capital of France" and gets these results:

```
Results for: capital of France
- The capital of France is Paris
- Paris is known as the City of Light
```

It saves these results to our shared context.

Third Round: The Final Decision 🤔

Our agent looks at the question again, but now it has some search results:

```
thinking: |
  Now I have sea                                                    ce
Paris".
  This directly a                                                   er.
action: answer
reason: I now have the information needed to answer the question
```

This time, our agent decides it has enough information to answer!

Fourth Round: Answering the Question 💬

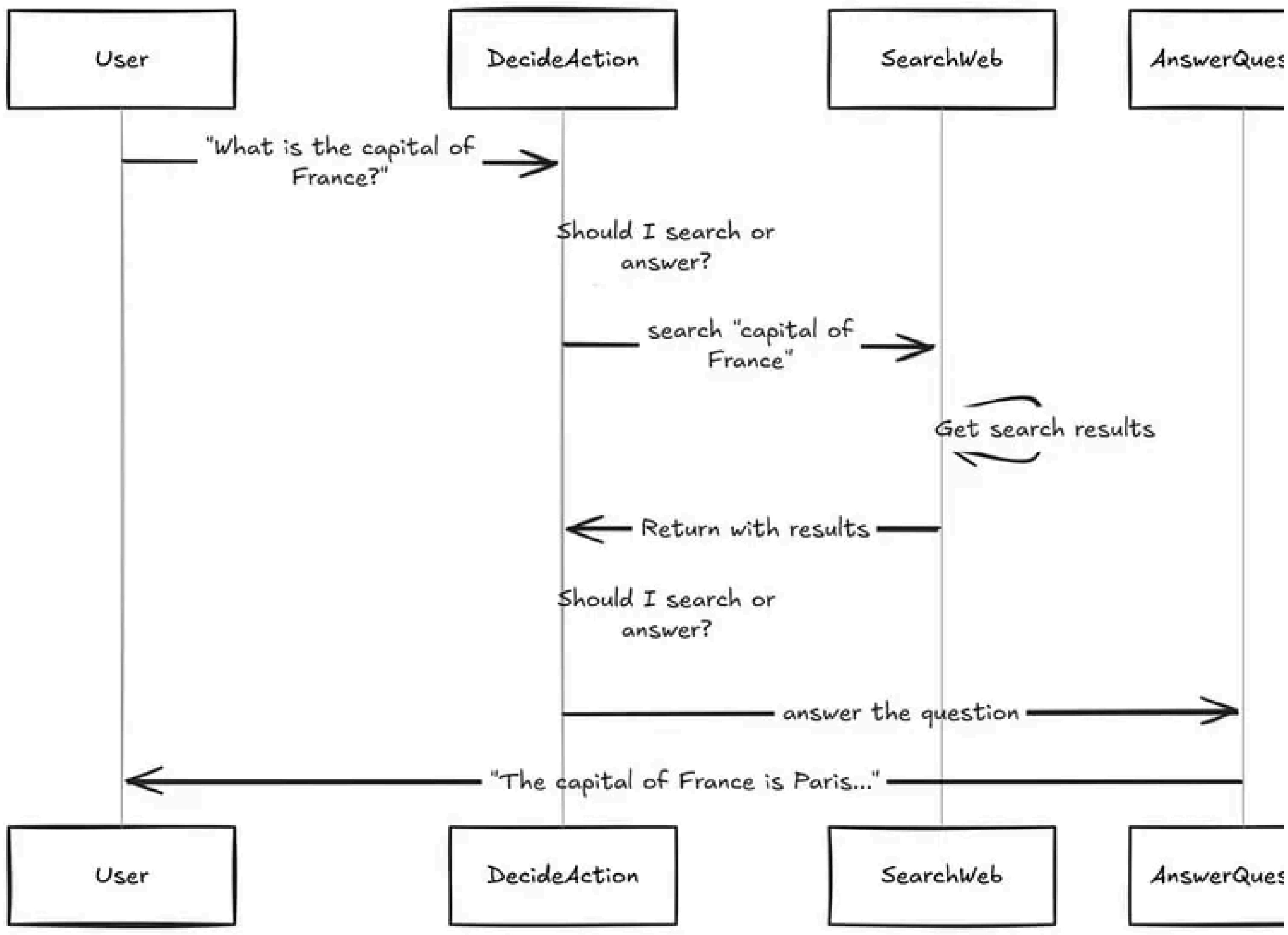
Finally, the AnswerQuestion node generates a helpful response:

```
The capital of France is Paris, which is also known as the City of
Light.
```

And we're done! Our agent has successfully:

1. Realized it needed to search for information
2. Performed a search
3. Decided it had enough information
4. Generated a helpful answer

The Whole Process Visualized:


User
DecideAction
SearchWeb
AnswerQues
"What is the capital of France?"
Should I search or answer?
search "capital of France"
Get search results
Return with results
Should I search or answer?
answer the question
"The capital of France is Paris..."
User
DecideAction
SearchWeb
AnswerQues

Dive Deeper: Explore the Code & Beyond

If you're technically inclined and want to read or experiment with the complete cc for this tutorial, you can find it here!

You can also see how this same loop-and-branch pattern appears in larger framew by checking out these snippets:

- **OpenAI Agents:** run.py#L119 for a workflow in graph.
- **Pydantic Agents:** _agent_graph.py#L779 organizes steps in a graph.
- **Langchain:** agent iterator.py#L174 demonstrates the loop structure.
- **LangGraph:** age

Looks like an article worth saving!

Option Q

Hover over the brain icon or use hotkeys to save with Memex.

Conclusion: it

Remind me later Hide Forever

Now you know the secret - LLM agents are just **loops with branches**:

1. **Think** about the current state
2. **Branch** by choosing one action from multiple options
3. **Do** the chosen action
4. **Get results** from that action
5. **Loop back** to think again

The "thinking" happens in the prompt (what we ask the LLM), the "branching" is when the agent chooses between available tools, and the "doing" happens when w call external functions. Everything else is just plumbing!

Next time you see a complex agent framework with thousands of lines of code, remember this simple pattern and ask yourself: "Where are the decision branches loops in this system?"

Armed with this knowledge, you'll be able to understand any agent system, no ma how complex it may seem on the surface.

Want to learn more about building simple agents? Check out PocketFlow on GitHub!

Thanks for reading Pocket Flow! Subscribe for free to receive new posts and support my work.

100 Likes · 2 Restacks

Previous Next

Looks like an article worth saving! Option Q

Hover over the brain icon or use hotkeys to save with Memex.

Remind me later Hide Forever

Discussion about t

Comments Restacks

Write a comment...

Solaris Mar 22 *Edited*

Liked by Zachary Huang

Pocket Flow examples design documents are really detailed. As a software engineer, writing an important best practice that helps understand and consider the implementation thorougl this approach is especially needed for critical thinking / productive reviews of what the LLM developing, so its great to see them as part of Pocket's workflow. Thank you for the exceller 🙏

LIKE (2) REPLY

1 reply by Zachary Huang

G K May 11

Liked by Zachary Huang

Hey, great post and framework!

I just started to work with pocketflow and it seems to be really easy to maintain consistency.

I do have a quick question, though:

How does Pocketflow differ from LangGraph, aside from LangGraph having extra features lil LangGraph Platform?

Other than being easy to maintain, what are some other advantages of using Pocketflow ove LangGraph?

LIKE (1) REPLY

2 replies by Zachary Huang and others

5 more comments...

Looks like an article worth saving!

Option Q

Hover over the brain icon or use hotkeys to save with Memex.

Remind me later Hide Forever