Build an LLM Web App in Python from Scratch Part 1 (Local CLI)

ZACHARY HUANG
JUN 02, 2025

> *Ever thought about sprinkling some AI magic into your web app? This is Part 1 of ou journey from a basic command-line tool to a full-blown AI web app. Today, we're bu a "Human-in-the-Loop" (HITL) chatbot. Think of it as an AI that politely asks for y "OK" before it says anything. We'll use a tiny, super-simple tool called [PocketFlow](#) – just 100 lines of code!*

So, You Wa

Adding AI (especia ops make them super p or coding help. But hold on, it's not just "plug and play." You'll bump into ques

- Where does the AI brain actually live? In the user's browser? On your server
- How do you handle AI tasks that need multiple steps?
- How can users tell the AI what to do or give feedback?
- What's the deal with remembering conversation history?
- **Most importantly:** How do you stop the AI from saying something totally w wrong?

This tutorial series is your guide to building LLM web apps in Python, and we'll these questions head-on! **Here's our 4-part adventure:**

- **Part 1 (That's Today!):** We'll build a basic Human-in-the-Loop (HITL) chatb runs in your computer's command line. We'll use PocketFlow to nail down t core logic, no fancy UI to distract us.
- **Part 2:** Let's get this HITL bot on the web! We'll build a user interface using **Streamlit** (or maybe **Flask**), and learn how to manage user interactions smoo
- **Part 3:** Time for real-time action! We'll upgrade our web app with **WebSock** (using **FastAPI**) for instant messages and a slicker chat feel.
- **Part 4:** What about AI tasks that take a while? We'll set up **background processing** and use **Server-Sent Events** (**SSE**) with **FastAPI**. This lets us sho users live progress updates, making our app feel truly pro.

To make this journey smooth, we'll use [PocketFlow](). It's not just for simple AI ca PocketFlow helps you build complex AI workflows that you can actually control. because it's so smal wa know what's going

The secret sauce to (H With PocketFlow, H dra

stuff, but **you** (the human co-pilot) get the final say. Approve, edit, or reject – you control before anything goes live. Total quality control, phew!

You can try out the code for the command-line HITL bot discussed in this part at: [Pock Command-Line HITL Example](#).

Your First HITL App: The "Are You Sure?" Bot

Let's build the most basic HITL app: a chatbot where you approve every single A response. Imagine it's an AI trying to tell jokes. AI can be funny, but sometimes jokes might be... well, not funny, or maybe even a little weird or inappropriate. T where *you* come in!

The Basic Idea (It's Super Simple!)

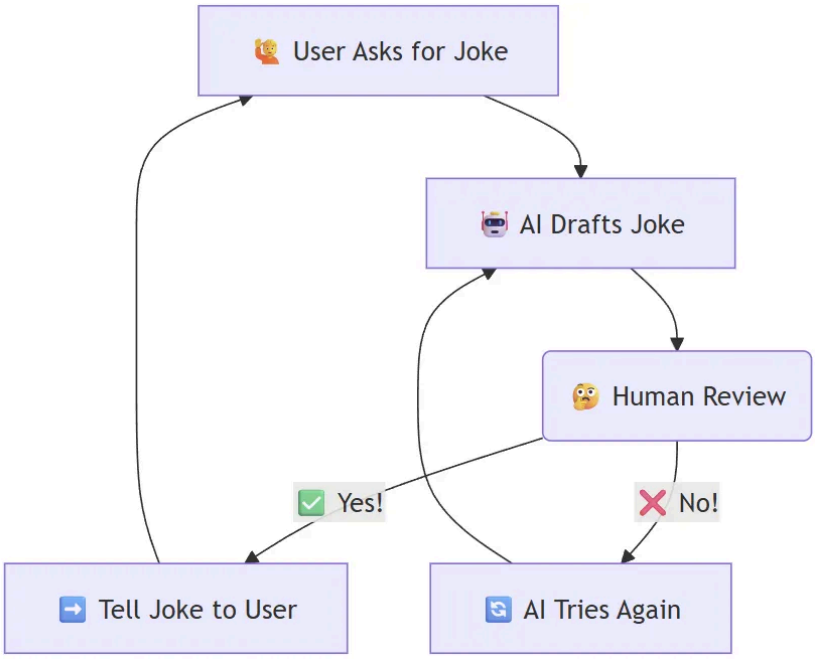
Think of a tiny script. The AI suggests a joke, and you, the human, give it a thum or thumbs-down before it's told. That's HITL in action! If the AI suggests, "Why the chicken cross the playground? To get to the other slide!" and you think it's a winner, you say "yes!" If it suggests something that makes no sense, or isn't right your audience, you say "nope!"

```
user_topic_request = "Tell me a joke about cats."
ai_suggested_joke = call_llm_for_joke(user_topic_request)

# You get to approve!
approval = input(f"AI suggests: '{ai_suggested_joke}'. Tell this jok
(y/n): ")

if approval.low
    print(f"To
else:
    print("Huma
```

That's the core! AI suggests a joke, human (`input()`) approves, then it's (maybe) This simple check is crucial for joke quality and appropriateness. Everything else build is about making this core idea robust for more complex apps. Visually, it's a loop, especially for our joke bot:



Let's See It Go!

```
You: Tell me a programming joke.
AI suggests: 'Why do programmers prefer dark mode? Because light
attracts bugs!'
Approve? (y/n): y
To Audience: Why do programmers prefer dark mode? Because light attr
bugs!

You: Tell me a joke about vegetables.
AI suggests: 'Why did the tomato turn red? Because it saw the salad
dressing! But a
Approve?
Human said: Nop                                                    nt
about vegetable
Regenerating...
AI suggests: 'W
```

```
Approve? (y/n): y
To Audience: What do you call a sad strawberry? A blueberry!
```

See? The human caught that slightly off-kilter joke and the unnecessary commer
That's HITL making sure the AI comedian doesn't bomb too hard!

Why Just Plain Python Gets Tangled Fast

When you try to build something more than a quick demo with plain Python `wh`
loops and `if` statements, things can turn into a bowl of spaghetti code REAL qu
The main headaches are:

1. 🍝 **Spaghetti Code:** Add features like conversation history, letting users edit trying different AI prompts, and your simple loop becomes a monster of nes `if/else` blocks. It's tough to read and a nightmare to fix.
2. 🧱 **Not Very Mix-and-Match:** Your logic for getting input, calling the AI, ge approval, and sending the response all gets jumbled together. Want to test ju piece? Or reuse your "AI calling" bit in another app? Good luck untangling
3. 🕸️ **Hard to Change or Grow:** Want to add a new step, like checking for bad before the human sees it? Or offer *three* ways to react instead of just "yes/no' plain Python, these changes mean carefully rewiring everything, and you'll probably break something.

These problems make it super hard to build AI workflows that are robust and rea
real users.

PocketFlow

Trying to build complex AI steps with plain Python is like building a giant LEG castle without instructions – you'll end up with a wobbly mess. PocketFlow is yo friendly instruction manual and a set of perfectly fitting LEGO bricks. It helps y build AI workflows in just 100 lines of actual framework code!

Imagine PocketFlow as running a little workshop:

- You have **Nodes**: These are your specialist workers, each good at one job.
- You have a **Shared Store**: This is like a central whiteboard where everyone sl notes.
- You have a **Flow**: This is the manager who tells the workers what to do and i what order.

The Node Pattern: Your Specialist Workers

In PocketFlow, each main task is a **Node**. A Node is like a specialist worker who' pro at *one specific thing*.

Here's what a Node looks like in PocketFlow:

```
class Node:
    def prep(self, shared): pass
    def exec(self, prep_res): pass
    def post(self, shared, prep_res, exec_res): pass
    def run(self,shared): p=self.prep(shared); e=self.exec(p); retur
self.post(shared,p,e)
```

Don't worry if `__init__` or `self` look weird; they're just Python things! The important bit is the

1. `prep(shared` whiteboard?"
2. `exec(data_f` calling an AI).

3. `post(shared, prep_res, exec_res)`: "Job's done! I'll write my result to the `shared` whiteboard and tell the manager (the Flow) what happened by returning a simple signal (like a keyword, e.g., `"done"` or `"needs_approva`

The Shared Store: The Central Whiteboard (Just a Python Dictionary!)

This is just a plain old Python dictionary (let's call it `shared_store`). All our N workers can read from it and write to it. It's how they pass info—like the user's question, the AI's draft answer, or conversation history—to each other.

For a math problem, it might start like this:

```
shared_store = {
    "number_to_process": 0,
    "intermediate_result": None,
    "final_answer": None
}
```

As Nodes do their work, they'll update this `shared_store`.

The Flow: The Workshop Manager

A `Flow` object is like the manager of your workshop. You tell it which Node kick things off. When you `run` a Flow:

1. It runs that first Node.
2. The Node finishes and returns a *signal* (just a text string, like `"user_appro` or `"try_agai`
3. The Flow checl : "] signal is `"user` `"try_again"`

You build this map with simple connections:

- `node_a >> node_b`: This is a shortcut. If `node_a` finishes and gives the u "all good" signal (PocketFlow calls this `"default"`), then `node_b` runs nex
- `node_a - "custom_signal" >> node_c`: This means if `node_a` finish and shouts `"custom_signal"`, then `node_c` is up.

The Flow keeps this going: run a Node, get its signal, find the next Node. If it ge signal and can't find a next step, the flow for that path just ends. Easy!
This lets you make workflows that branch off in different directions based on wh happens. Like a choose-your-own-adventure for your AI!

Here's how tiny the `Flow` manager class actually is in PocketFlow:

```
class Flow:
    def __init__(self, start_node): self.start_node = start_node
    def run(self, shared_store):
        current_node = self.start_node
        while current_node:
            signal = current_node.run(shared_store)
            current_node = current_node.successors.get(signal)
```

That's it! It just needs a `start_node` and then it keeps running nodes and follo their signals until there's no next step defined for a signal.

Tiny Math Example: PocketFlow in Action!

Let's build a super-tiny workflow: take a number, add 5, then multiply by 2.

Worker 1: The Add
This Node's job is t


```
class AddFive(N
    def prep(self, shared):
        return shared.get("number_to_process", 0)
```

```
    def exec(self, current_number):
        return current_number + 5

    def post(self, shared, prep_res, addition_result):
        shared["intermediate_result"] = addition_result
        print(f"AddFive Node: Added 5, result is {addition_result}")
        return "default" # Signal "all good, continue"
```

1. `prep`: Grabs `"number_to_process"` from `shared_store`.
2. `exec`: Adds 5.
3. `post`: Saves the new number as `"intermediate_result"` and says `"default"` (meaning "continue to the next step in line").

Worker 2: The Multiplier Node

This Node's job is to multiply by 2.

```
class MultiplyByTwo(Node):
    def prep(self, shared):
        return shared["intermediate_result"]

    def exec(self, current_number):
        return current_number * 2

    def post(self, shared, prep_res, multiplication_result):
        shared["final_answer"] = multiplication_result
        print(f"MultiplyByTwo Node: Multiplied, final answer is
{multiplication_result}")
        return "done" # Signal "all finished with this path"
```

1. `prep`: Grabs "
2. `exec`: Multipli
3. `post`: Saves it

Connecting the Workers (The Assembly Line):

Now, let's tell PocketFlow how these Nodes connect.

```
# First, make our specialist worker Nodes
adder_node = AddFive()
multiplier_node = MultiplyByTwo()

adder_node >> multiplier_node

# Create the Flow manager
math_flow = Flow(start_node=adder_node)

# Let's get some data for them to work on
shared_store_for_math = {"number_to_process": 10}
print(f"\nStarting math game with: {shared_store_for_math}")

# Run the flow!
math_flow.run(shared_store_for_math)

print(f"Math game finished. Whiteboard looks like:
{shared_store_for_math}")
```

And if you run this, you'd see:

```
Starting math game with: {'number_to_process': 10}
AddFive Node: Added 5, result is 15
MultiplyByTwo Node: Multiplied, final answer is 30
Math game finished. Whiteboard looks like: {'number_to_process': 10,
'intermediate_result': 15, 'final_answer': 30}
```

See? Each Node (wo :he number through. T `MultiplyByTwo`, l ot signals for differen

Now we know how PocketFlow uses Nodes, a Shared Store, and a Flow to handle steps pass data. Let's use these exact same ideas for our HITL approval chatbot!

Building Your HITL "Approval Bot" Workflow

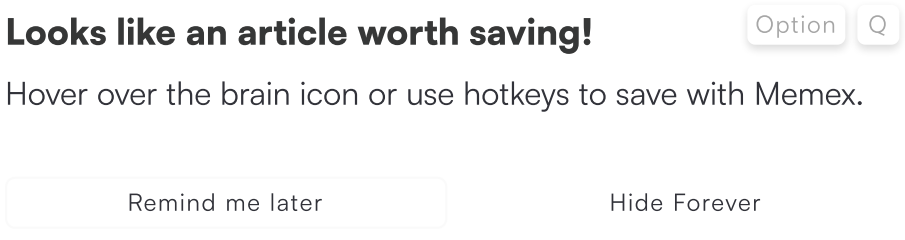
Alright, enough theory! Let's get our hands dirty and build that HITL workflow w PocketFlow for our joke bot. This time, we'll start by thinking about what inform our Nodes will need to share.

First, Design the Shared Store (Our Whiteboard)

For our interactive joke generator, the `shared_store` dictionary (our central whiteboard) needs to keep track of a few key things as the conversation flows:

- `topic`: What kind_of_joke the user wants (e.g., "cats", "programming"). Thi be filled in by our first Node.
- `current_joke`: The latest joke the AI cooked up. This will be updated by t joke generation Node.
- `disliked_jokes`: A list of jokes about the current `topic` that the user alr said "no" to. This helps the AI avoid telling the same bad joke twice. It will l updated by our feedback Node.
- `user_feedback`: The user's latest decision (e.g., `"approve"` or `"disapprove"`). Also updated by the feedback Node.

Here's a peek at what it might look like while the bot is running:




```
shared_store =
    "topic": "d
    "current_jo
labracadabrador
    "disliked_jokes": ["Why did the dog cross the road? To get to th
```

```
barking lot!"],
    "user_feedback": None
}
```

Each Node we design will read the information it needs from this `shared_stor` write its results back here for other Nodes to use. This way, everyone's on the sa page!

The Three Core Nodes: Our Specialist Joke Crafters

We'll use three main Nodes for our joke-making machine:

1. **GetTopicNode**: Asks the user what they want a joke about.
2. **GenerateJokeNode**: Cooks up a joke using the AI.
3. **GetFeedbackNode**: Asks the user if the joke was a hit or a miss.

Let's build them one by one!

1. GetTopicNode - The Idea Catcher 🎤

This Node's only job is to ask the user for a joke topic.

```
class GetTopicNode(Node):
    def prep(self, shared):
        shared["topic"] = None
        shared["current_joke"] = None
        shared["disliked_jokes"] = []
        shared["user_feedback"] = None
        return None
```

First, the `prep` met : cl up any old informat sta fresh for the new to

```
    def exec(self, _prep_res):
        return input("What topic shall I jest about today? ")
```

Next, `exec` does the main work: it simply asks the user for a topic using `input(` returns whatever they type.

```
    def post(self, shared, _prep_res, topic_input):
        shared["topic"] = topic_input
        print(f"Alright, a joke about '{topic_input}'! Coming right
up...")
        return "generate_joke"
```

Finally, `post` takes the `topic_input` from `exec`, saves it to our `shared` whitel under the key `"topic"`, prints a little message, and then returns the signal `"generate_joke"`. This signal tells the Flow manager, "Okay, we have a topic, go to the node that generates jokes!"

2. GenerateJokeNode - The AI Comedy Chef 🤖

This Node grabs the topic (and any jokes the user *didn't* like about it) and tells th to whip up a new joke.

```
class GenerateJokeNode(Node):
    def prep(self, shared):
        topic = shared.get("topic", "something random")
        disliked = shared.get("disliked_jokes", [])

        prompt = f"Tell me a short, funny joke about: {topic}."
        if disl
            avo
            pro                                                    e:
[{avoid_these}]
        return
```

In `prep`, this Node looks at the `shared` whiteboard for the `"topic"` and any `"disliked_jokes"`. It then crafts a `prompt` (a set of instructions) for our AI. I there *are* disliked jokes, it cleverly tells the AI, "Hey, avoid jokes like these ones t user didn't like!"

```
    def exec(self, joke_prompt):
        return call_llm(joke_prompt)
```

Then, `exec` is where the AI magic would happen. We take the `joke_prompt` fr `prep` and send it to our `call_llm` function. This function would be responsible talking to a real AI service (like OpenAI, Anthropic, etc.) and returning its respo

```
    def post(self, shared, _prep_res, ai_joke):
        shared["current_joke"] = ai_joke
        print(f"🤖 AI Suggests: {ai_joke}")
        return "get_feedback"
```

And in `post`, we save the `ai_joke` to our `shared` whiteboard as `"current_j` print it out for the user to see, and then return the signal `"get_feedback"`. Th the Flow manager, "Joke's ready! Go to the node that gets the user's opinion."

3. GetFeedbackNode - The Joke Judge 🤔

This Node shows the joke and asks the user: thumbs up or thumbs down? Based their answer, it decides if we should try another joke on the same topic or if we'r done.

```
class GetFeedba
    def prep(se
        return l
```

`prep` is super simple here. `GenerateJokeNode` already showed the joke, so the nothing to set up. We just pass `None`.

```
    def exec(self, _prep_res):
        while True:
            decision = input("Did you like this joke? (yes/no): 
").strip().lower()
            if decision in ["yes", "y", "no", "n"]:
                return decision
            print("Hmm, I didn't catch that. Please type 'yes' or 
'no'.")
```

`exec` asks the user if they liked the joke. It waits for a clear "yes" (or "y") or "no" "n") before moving on.

```
    def post(self, shared, _prep_res, user_decision):
        if user_decision in ["yes", "y"]:
            print("🎉 Hooray! Glad you liked it!")
            shared["user_feedback"] = "approve"
            return "joke_approved"
        else:
            print("😅 Oops! My circuits must be crossed. Let me try 
again...")
            shared["user_feedback"] = "disapprove"
            current_joke = shared.get("current_joke")
            if current_joke:
                shared.setdefault("disliked_jokes", 
[]).append(current_joke)
            return "regenerate_joke"
```

Finally, `post` looks

- If it's a "yes", w ac
  and return the is j
  topic.

- If it's a "no", we apologize, store `"disapprove"`, add the failed `current_` to our `shared["disliked_jokes"]` list (so `GenerateJokeNode` knows repeat it), and return the signal `"regenerate_joke"`. This tells the Flow manager: "Back to the joke drawing board (the `GenerateJokeNode`) for the topic!"

Connecting the Flow: Drawing the Joke Map & Runnin It! 🗺️

Now we tell PocketFlow how to get from one Node to another using the signals v just defined, and then we'll run it.

```
# 1. Make our specialist Node workers
get_topic_node = GetTopicNode()
generate_joke_node = GenerateJokeNode()
get_feedback_node = GetFeedbackNode()

# 2. Draw the paths for the Flow manager
get_topic_node - "generate_joke" >> generate_joke_node
generate_joke_node - "get_feedback" >> get_feedback_node
get_feedback_node - "regenerate_joke" >> generate_joke_node

# 3. Let's Run Our HITL Joke Bot!
shared = {
    "topic": None, "current_joke": None,
    "disliked_jokes": [], "user_feedback": None
}
hitl_joke_flow = Flow(start_node=get_topic_node)
hitl_joke_flow.run(shared)

print(f"\n🎤 Joke session over! Here's what happened: {shared}")
```


This is like drawing e journey:

1. We create our t

2. We use the `node - "signal" >> next_node` pattern to define the path
3. We set up our `shared` whiteboard.
4. We create our `Flow` manager (using the `Flow` class we saw earlier), telling it kick things off with `get_topic_node`.
5. We call `hitl_joke_flow.run(shared)`. The Flow manager now takes ov runs the Nodes, listens for their signals, and follows the map. The `shared` dictionary gets updated live.
6. When the flow naturally ends (because `"joke_approved"` has no next step `run` method finishes, and we print out the final state of our whiteboard.

And that's it! You've built a Human-in-the-Loop chatbot using PocketFlow. Eacl is small, understandable, and they all work together to create a flexible workflow where the human is always in control.

From CLI to Web App: What's Next & Key Takeaways

You've built a cool command-line bot where you're the boss! But most people do hang out in command prompts, right? They use web apps! The great news is that HITL logic you've built with PocketFlow is the *engine* that can power a web UI t

The Journey Ahead:

- **Part 2 (Web UI):** We'd take this exact HITL flow and hook it up to somethin **Streamlit** or **Flask**. Instead of `input()`, users would click buttons (`st.button("👍 Approve")`). The underlying PocketFlow logic remains largely the sam
- **Part 3 (Real-Ti WebSockets** (u HITL decision wi Th

- **Part 4 (Background Tasks & Progress Updates):** What about AI tasks that ta while? We'll set up **background processing** and use **Server-Sent Events (SSE FastAPI**. This lets us show users live progress updates, making our app feel pro. Your PocketFlow `Flow` is a key part of each user's session.

In Part 2, we'll take our HITL bot to the web using Streamlit! This means buildir proper user interface where users can interact with buttons and see responses. W even explore how such a UI could handle more than just text, like displaying ima making our AI interactions richer.

Want to try out this exact command-line HITL bot yourself? You can find the compl runnable code in the PocketFlow cookbook here: [PocketFlow Command-Line HITL Ex](#) Go ahead, build your own HITL apps!

← Previous Ne

Discussion about this post