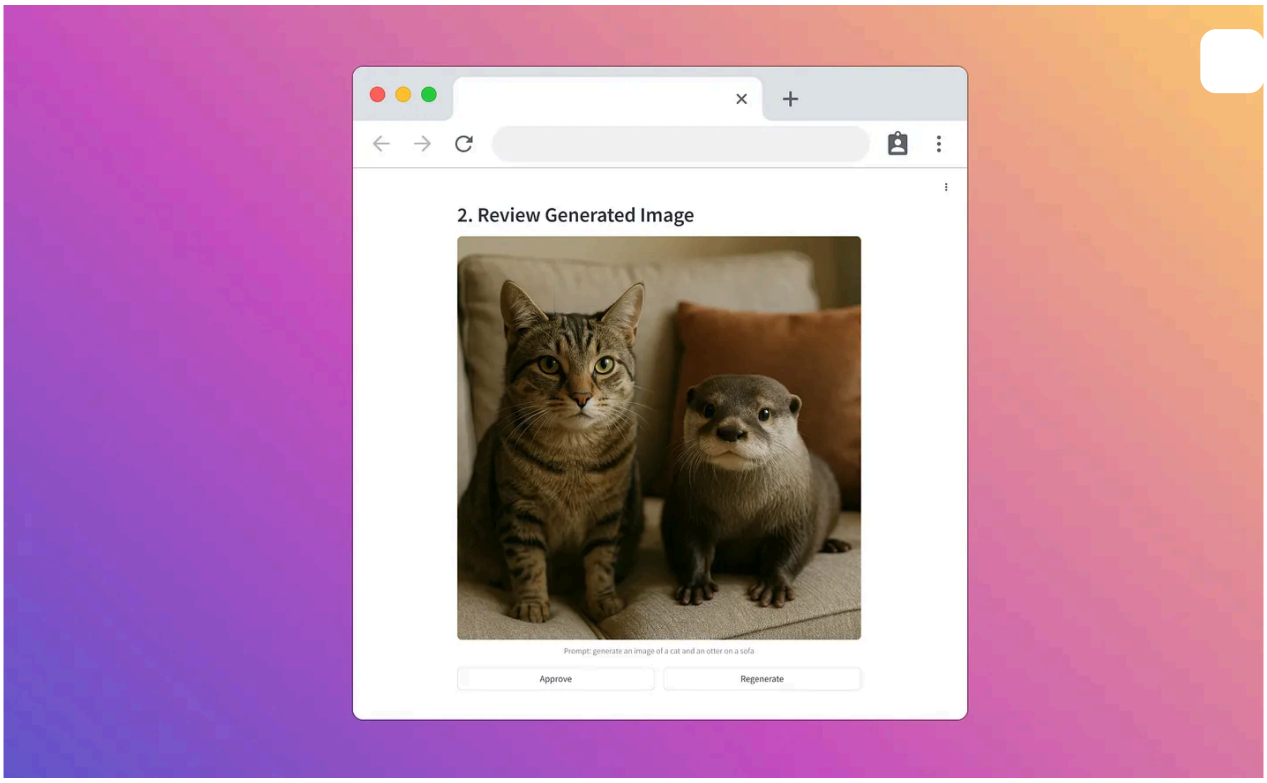
Build an LLM Web App in Python from Scratch Part 2 (Streamlit & FSM)

ZACHARY HUANG
JUN 05, 2025

5 4 1


2. Review Generated Image
Prompt: generate an image of a cat and an otter on a sofa
Approve
Regenerate

> *Ever wanted to create your own AI-powered image generator, where you call the shot the final masterpiece? That's exactly what we're building today! We'll craft an intera web application that lets users generate images from text prompts and then approve o regenerate them – all within a user-friendly interface. We'll use [PocketFlow](https://github.com/The-Pocket/PocketFlow) for workj management, and*

1. Want to E et Do It! 🎨

Imagine you're an artist, and you have a super-smart assistant (an AI) who can pa anything you describe. You tell it, "Paint a cat and an otter on a sofa!" The AI qu paints a picture. But maybe the cat looks a bit grumpy, or the otter is on the floo instead of the sofa. Wouldn't it be great if you could tell the assistant, "Make the look friendlier," or "Put the otter on the sofa next to the cat," and it would repair

That's where you come in as the director. You get to say "Nope, try again!" or "Perfect, I love it!" This back-and-forth between you and the AI is called **Human the-Loop** (**HITL**), and it's exactly what we're building today.

We're creating a web app where you can: (1) Type what you want (like "a robot ea pizza on Mars"), (2) See what the AI creates, (3) Approve it or ask for a do-over, a Keep the final masterpiece.

Don't worry – we're keeping it simple with just a few Python tools:

- 🔧 **Streamlit** – Turns your Python code into a web app. No HTML/CSS need
- 🔧 PocketFlow – Organizes our AI tasks like a recipe (we used this in Part 1
- 🔧 **Finite State Machine** (**FSM**) – Keeps track of where we are in the process (typing → generating → reviewing → done)

This is part 2 of our 4-part journey of LLM web app tutorial:

- **Part 1:** Built the basic HITL system in the command line ✅
- **Part 2** (**You are here!**): Making it a real web app with Streamlit 🚀
- **Part 3:** Adding
- **Part 4:** Handlir

Ready to turn your

Want to see the final result? Check out the complete code example we're building

2. Streamlit 101: A Web App in a Single Pytho File!

So we've picked **Streamlit** to build our AI Image Generator. Why is this perfect f Python folks who don't want to mess with HTML and CSS? Streamlit gives you a package of UI components (buttons, text boxes, sliders, charts) right out of the b no web design skills required! Plus, it has this brilliantly simple approach called "rerun" model.

Streamlit's Big Idea: Just Rerun It! 🔄

Here's the wild part – and it's dumb simple: Every time a user does something (a button, types text), Streamlit runs your *entire Python script* from top to bottom. Every single time.

"Wait, the whole script? Isn't that... slow?" you might think. Nope! Streamlit is c fast at this, and it actually makes everything simpler. Think of it like a chef who your entire meal fresh every time you ask for extra salt – except this chef works a lightning speed.

Why this "rerun everything" approach rocks:

- **Feels natural:** You write normal Python code, step by step
- **No complicated UI updates:** Streamlit automatically redraws everything bas your latest scri
- **Easy to unders** xp

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

Your First St

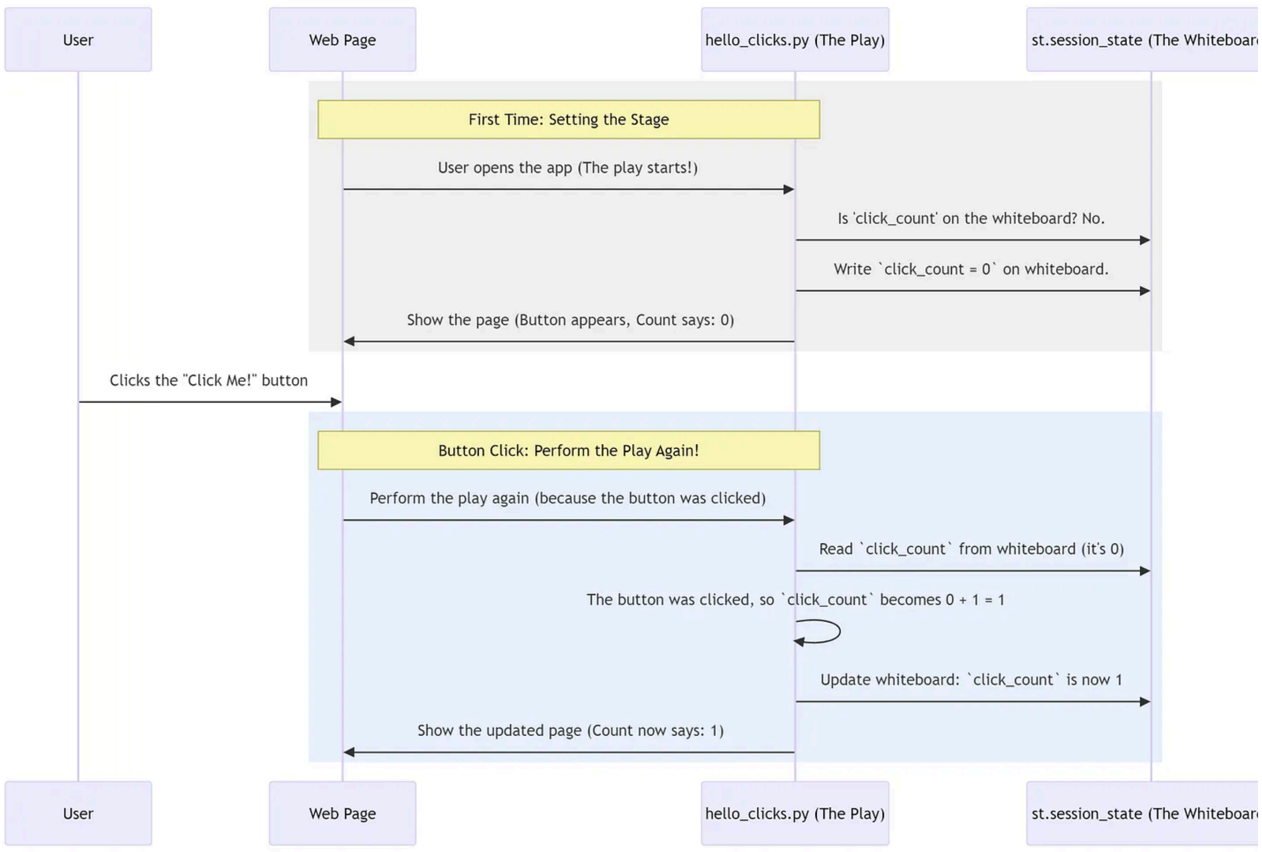
Let's see this in action with a simple click counter. Create a file called `hello_clicks.py`:

```
import streamlit as st

st.title("Hello, Clicks!")

# Think of st.session_state as Streamlit's memory notebook
if 'click_count' not in st.session_state:
    st.session_state.click_count = 0

if st.button("Click Me!"):
    st.session_state.click_count += 1

st.write(f"Button has been clicked: {st.session_state.click_count}
times")
```

Run it with: `streamlit run hello_clicks.py`

What Happens When You Click? (The Rerun Dance!)

Let's follow what happens when you click the button for the first time, like steps dance:


User
Web Page
hello_clicks.py (The Play)
st.session_state (The Whiteboard
First Time: Setting the Stage
User opens the app (The play starts!)
Is 'click_count' on the whiteboard? No.
Write `click_count = 0` on whiteboard.
Show the page (Button appears, Count says: 0)
Clicks the "Click Me!" button
Button Click: Perform the Play Again!
Perform the play again (because the button was clicked)
Read `click_count` from whiteboard (it's 0)
The button was clicked, so `click_count` becomes 0 + 1 = 1
Update whiteboard: `click_count` is now 1
Show the updated page (Count now says: 1)

First Time: Setting the Stage (Grey Box):

1. You open the app in your **Web Page** (Browser)
2. The Python **Script** (`hello_clicks.py` – our play) runs for the very first tin
3. It looks at the **SessionState** (our whiteboard) and sees `'click_count'` isn'
4. So, it writes `st.session_state.click_count = 0` on the whiteboard
5. The script then tells the **Web Page** to show the button and the text "Count: (

User Clicks Button:

1. The **User** clicks the "Click Me!" button on the web page

Button Click: Perfc

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

1. Streamlit sees t fr
   the first line to
2. This time, whe.
   `'click_count'` (its value is `0`)

3. The line `if st.button("Click Me!"):` is true (because *this specific* butt click is what caused the rerun). So, the script updates `st.session_state.click_count` to `1` (0 + 1)
4. The script finishes, and the **Web Page** shows the updated UI with "Count: 1

Every time you click again, the steps in the "Button Click Rerun" (blue box) repe The `st.session_state.click_count` keeps going up because **SessionState** whiteboard) remembers its value between each rerun of the play.

`st.session_state`: Your App's Memory Bank 🧠

Since your script reruns from scratch each time, regular Python variables would everything. That's where `st.session_state` comes in – it's like a personal no that Streamlit gives each user to remember important stuff.

What goes in this memory bank? Anything your app needs to remember: user in calculation results, which screen you're on, and for our image generator – the us prompt, the generated image, and where we are in the process.

Connecting to Part 1: Remember the `shared_data` dictionary from our comma line app? `st.session_state` is exactly that, but for web apps. Instead of passi data between PocketFlow nodes in an in-mem dictionary, we'll store it in `st.session_state`.

Now that you get how Streamlit works, let's plan out our image generation workflo

---

3. Planning PocketFlov

Looks like an article worth saving!

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

Option Q

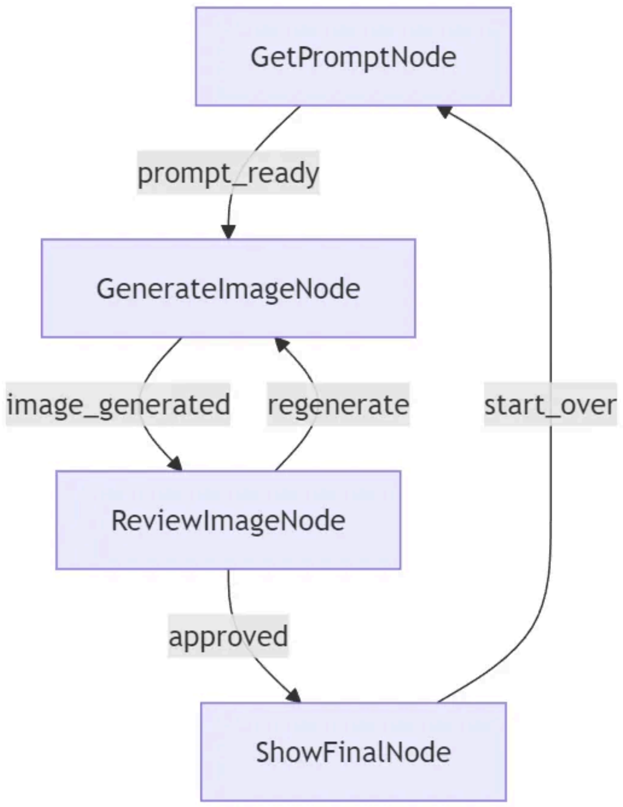
Remind me later Hide Forever

Time to plan our m mp let's map out what our AI Image Generator actually needs to do. Think of this lik sketching before you paint – we want to get the big picture right first.

We'll use PocketFlow to organize our workflow. Remember from Part 1? It's our tool for connecting different steps together in a logical sequence.

The Basic Journey: From Idea to Image

Our app has a simple but powerful flow: (1) User types what they want, (2) AI ger an image, (3) User reviews it and decides if it's good, and (4) Either keep it or try That's it! Here's how it looks:



PocketFlow Nodes: The Building Blocks

In PocketFlow, each step is a "Node" – think of them as LEGO blocks that do sp jobs. Every Node hε

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

- 🔧 `prep` – Grabs w
- ⚡ `exec` – Does the
- 📝 `post` – Saves res

Let's see these in action:

Node 1: Getting the User's Idea

```
class GetPromptNode(Node):
    def exec(self, prep_res):
        prompt = input("What do you want to see?")
        return prompt

    def post(self, shared, prep_res, exec_res_prompt):
        shared["task_input"] = exec_res_prompt  # Save to memory
        return "prompt_ready"  # Signal: we're ready for the next st
```

Node 2: AI Creates the Magic

```
class GenerateImageNode(Node):
    def prep(self, shared):
        return shared.get("task_input")  # Get the prompt from memor

    def exec(self, prep_res_prompt):
        image_data = generate_image(prep_res_prompt)
        return image_data

    def post(self, shared, prep_res_prompt, exec_res_image_data):
        shared["generated_image"] = exec_res_image_data  # Save imag
        return "image_generated"  # Signal: image is ready!
```

Node 3: User Reviews the Result

```
class ReviewIma
    def prep(se
        display

    def exec(se
        choice 
        return choice
    def post(self, shared, prep_res, exec_res_choice):
        if exec_res_choice == 'a':
            shared["final_result"] = shared["generated_image"]
            return "approved"
        else:
            return "regenerate"  # Try again!
```

Node 4: Celebrating the Final Result

```
class ShowFinalNode(Node):
    def prep(self, shared):
        display(shared["generated_image"])

    def exec(self, prep_res):
        choice = input("Start Over (s)?")
        return choice

    def post(self, shared, prep_res, exec_res_choice):
        if exec_res_choice == 's':
            # Clear memory for fresh start
            for key in ["task_input", "generated_image",
"final_result"]:
                shared.pop(key, None)
            return "start_over"
```

Connecting the Dots

PocketFlow makes connecting these steps super clean:

```
def create_image generator flow():
    # Create ou
    get_prompt
    generate_im
    review_imag
    show_final

    # Connect them with signals
    get_prompt     - "prompt_ready"    >> generate_image
    generate_image - "image_generated" >> review_image
    review_image   - "approved"        >> show_final
    review_image   - "regenerate"      >> generate_image  # Loop bac
    show_final     - "start_over"      >> get_prompt      # Start fr

    return Flow(start_node=get_prompt)
```

The Web App Challenge: No More `input()`

Here's the thing: command-line apps can pause and wait with `input("What's next?")`. Web apps can't do that! A web server needs to stay responsive for all u not freeze while waiting for one person to click a button.

That's where we need a different approach – and that's exactly what we'll solve with F State Machines in the next section!

---

4. FSM to the Rescue: Managing Interactive W States

Remember the challenge we just hit? Command-line apps can pause and wait wi `input()`, but web apps need to stay responsive. That's where **Finite State Mach (FSMs)** come to the rescue!

Think of an FSM like a roadmap for your app. Instead of getting lost wondering "where am I in the process?", your app always knows exactly which "room" it's i what should happer

You can see the comp

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

What's a Fini

An FSM is just a fancy way to organize your app into different "modes" or "

🏠 **States** – Different rooms your app can be in:

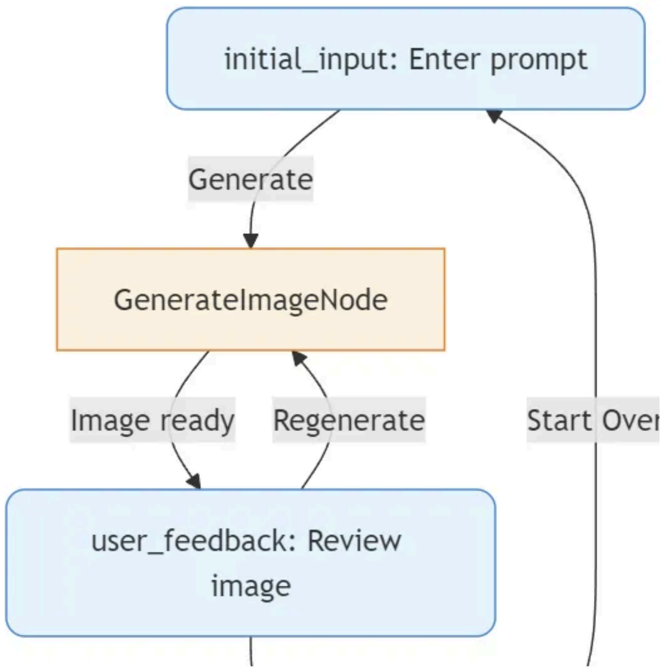
- `initial_input`: "Hey, tell me what you want to see!"
- `user_feedback`: "Here's your image. Like it or want me to try again?"
- `final`: "Awesome! Here's your approved masterpiece!"

🚪 **Transitions** – How you move between rooms:

- User clicks "Generate" → move from `initial_input` to `user_feedback`
- User clicks "Approve" → move from `user_feedback` to `final`
- User clicks "Start Over" → move from `final` back to `initial_input`

Visualizing Our App's States

Here's how our image generator flows between states:



How FSM + Streamlit + PocketFlow Work Together

Here's the beautiful part: `st.session_state` becomes our master control cen
tracks both **which state we're in** AND **all our PocketFlow data**.

```
# Initialize our app's memory
if 'state' not in st.session_state:
    st.session_state.state = "initial_input"  # Start here
    st.session_state.task_input = ""
    st.session_state.generated_image = ""
    st.session_state.final_result = ""
```

Then we use simple `if/elif` blocks to show different screens based on the cur
state:

The Magic: State-Based UI

```
# State 1: Getting user input
if st.session_state.state == "initial_input":
    st.header("🎨 What do you want to see?")
    prompt = st.text_area("Describe your image:")

    if st.button("Generate Image"):
        st.session_state.task_input = prompt

        # Run PocketFlow node
        image_node = GenerateImageNode()
        image_node.run(st.session_state)

        # Move to next state
        st.session_state.state = "user_feedback"
        st.rerun()

# State 2: User
elif st.session_
    st.header("
    st.image(st

    col1, col2 =
    with col1:
        if st.button("👍 Approve"):
            st.session_state.final_result =
st.session_state.generated_image
            st.session_state.state = "final"
            st.rerun()

    with col2:
        if st.button("🔄 Try Again"):
            # Run PocketFlow node again
            image_node = GenerateImageNode()
            image_node.run(st.session_state)
            # Stay in same state, just refresh
            st.rerun()

# State 3: Show final approved image
elif st.session_state.state == "final":
    st.header("🎉 Your Masterpiece!")
    st.success("Image approved!")
    st.image(st.session_state.final_result)

    if st.button("🆕 Start Over"):
        # Reset everything
        st.session_state.task_input = ""
        st.session_state.generated_image = ""
        st.session_state.final_result = ""
        st.session_state.state = "initial_input"
        st.rerun()
```

The Power of This Approach

- 🔄 **No More Confusion:** Your app always knows exactly where it is and what should happen next.
- 🎯 **Clean Code:** Each state handles its own UI and logic – no messy spaghett code!
- 🚀 **Streamlit F**
- 🔧 **PocketFlow**
- 👥 **User-Frienc** d d

This FSM approach transforms our PocketFlow workflow from a linear comman script into an interactive web experience. Users can bounce between states natu and your code stays organized and predictable.

5. Mission Accomplished! What's Next? 🚀

Boom! We just built a fully interactive AI Image Generator web app. Look what achieved: (1) Streamlit handles the UI magic, (2) FSM keeps our app states organi (3) PocketFlow manages our AI workflow, and (4) Users get a smooth, intuitive experience.

What we learned:

- FSMs make interactive web apps way easier to manage
- `st.session_state` is perfect for both FSM states and PocketFlow data
- Streamlit + FSM + PocketFlow = a powerful combo for AI apps

Our journey so far:

- **Part 1:** Built HITL logic with PocketFlow (command line) ✅
- **Part 2 (This part!):** Created an interactive web app with Streamlit + FSM ✅
- **Part 3 (Coming up!):** Real-time features with FastAPI & WebSockets
- **Part 4:** Background processing and progress updates

Ready to add real-time superpowers to your AI apps? Part 3 will show you how FastAPI and WebSockets can create instant, live interactions – think real-time c with AI or live image generation updates!

Want to try this ou *ket*

5 Likes · 1 Restack

← Previous Nex

Discussion about this post

Comments Restacks

Write a comment...

Sanchit Jun 8

Liked by Zachary Huang

Really great series happy to have found this substack thank you!

LIKE (1) REPLY

ramon betancourt Jun 6

Liked by Zachary Huang

Ok nice and will this work for interactive video generation too?

LIKE (1) REPLY

1 reply by Zachary Huang

2 more comments...