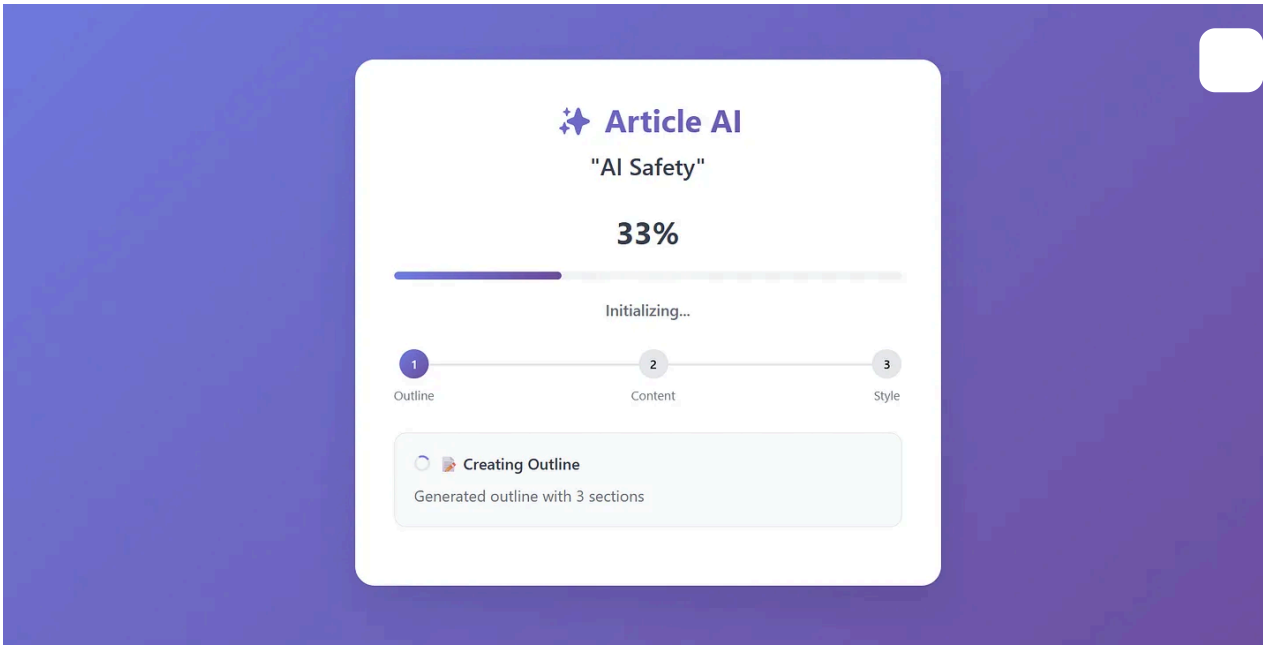
# Build an LLM Web App in Python from Scratch Part 4 (FastAPI, Background Tasks & SSE)

ZACHARY HUANG
JUN 12, 2025






> *Ever asked an AI to write a whole blog post, only to stare at a loading spinner for five minutes, wondering if your browser crashed? We've all been there. Today, we're fixing We'll build an AI web app that takes on heavy-duty tasks—like writing a full article- without freezing up. You'll see live progress updates in real-time, so you always know the AI is up to. Ready to make long-running AI tasks feel fast and interactive? Let's g building!*

## 1. The Spin Break Your

**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

Imagine your new A :ic about space exploration, nit enter, and... the dreaded loading spinner appears. T

whole page is frozen. You can't click anything. After a minute, your browser mig even give you a "This page is unresponsive" error. Yikes.

This is the classic problem with long-running tasks on the web. Standard web ap work like this:



1. **You ask for something.** (e.g., "Generate my article!")
2. **The server works on it.** (e.g., Calls the AI, which takes 2-3 minutes)
3. **You wait... and wait...** (The connection is held open, your browser is stuck)
4. **Finally, you get the result.** (Or a timeout error!)

This is a terrible user experience. Users need to know their request is being hanc and see that progress is being made.

**Our Solution: The Smart Restaurant Analogy**

Think of it like ordering food at a restaurant:

- **The Bad Way (Standard Web Request):** You order a steak. The waiter stands your table, staring at you without moving, for the entire 15 minutes it takes t cook. Awkward, right? You can't even ask for more water.
- **The Good Way (Our Approach):** You order a steak. The waiter says, "Excelle choice! I'll put that in with the chef," and walks away (**Background Task**). A minutes later, they bring you some bread ("Making progress!"). A bit later, yc drink arrives ("Almost ready!"). You're happy and informed while the main c is being prepar



That's exactly what

1. Instantly confi

2. Offload the heavy AI work to a **background task**.
3. Stream live progress updates back to the user with **Server-Sent Events** (SSE)

**Our toolkit for this mission:**

- 🔧 **FastAPI** `BackgroundTasks`: For running the AI job without freezing th
- 🔧 **Server-Sent Events** (SSE): A simple way to push live updates from the ser the browser.
- 🔧 [PocketFlow](): To organize our multi-step article writing process.

Let's dive into the tools that make this magic possible.

---

*You can find the complete code for the app we're building today in the PocketFlow coo [FastAPI Background Jobs with Real-time Progress]().*

---

# 2. Our Tools for the Job: Background Tasks & Server-Sent Events (SSE) 🛠️

To build our responsive AI article writer, we need two key pieces of technology: handle the work behind the scenes and another to report on its progress.

## FastAPI `BackgroundTasks`: The "Work in the Back" C

Think of `BackgroundTasks` as giving a job to a helper who works independent You tell FastAPI, "Hey, after you tell the user I got their request, please run this function in the back."



**The magic is that t** ait the background wo

It's like ordering fr ly. actual process of picking, packing, and shipping your item happens *later*, in

background.

Here's a simple example: sending a welcome email after a user signs up.

```
from fastapi import FastAPI, BackgroundTasks

app = FastAPI()

# This is our slow task (e.g., calling an email service)
def send_welcome_email(email: str):
    import time
    time.sleep(5) # Simulate a 5-second delay
    print(f"Email sent to {email}! (This prints in the server consol

@app.post("/signup")
async def user_signup(email: str, background_tasks: BackgroundTasks)
    # Add the slow email task to the background
    background_tasks.add_task(send_welcome_email, email)

    # Return a response to the user IMMEDIATELY
    return {"message": f"Thanks for signing up, {email}! Check your
inbox soon."}
```

**What's happening?**

1. When you send a request to `/signup`, FastAPI sees `background_tasks.add_task(...)`.
2. It immediately sends back the `{"message": "Thanks..."}` response. Yo browser is happy and responsive.
3. *After* sending the response, FastAPI runs `send_welcome_email()` in the background.



This is perfect for c iti process.

## Server-Sent Events (SSE): Your Live Progress Ticker

Okay, so the work is happening in the background. But how do we tell the user w going on? That's where **Server-Sent Events** (**SSE**) come in.

SSE is a super simple way for a server to push updates to a browser over a single, way connection. It's like a live news ticker: the server sends new headlines as the happen, and your browser just listens.

**Why not use WebSockets again?**
WebSockets (from Part 3) are awesome for two-way chat. But for just sending on progress updates, they're a bit like using a walkie-talkie when all you need is a pa SSE is simpler, lighter, and designed for exactly this "server-to-client" push scen

Here's how simple an SSE endpoint is in FastAPI:

```
import asyncio, json
from fastapi import FastAPI
from fastapi.responses import StreamingResponse

app = FastAPI()

async def progress_generator():
    for i in range(1, 6):
        # In a real app, this would be a progress update from our AI
        yield f"data: {json.dumps({'progress': i*20, 'step': f'Step
done'})}\n\n"
        await asyncio.sleep(1) # Wait 1 second

@app.get("/stream-progress")
async def stream_progress():
    return StreamingResponse(progress_generator(),
media_type="text/event-stream")
```

And on the browse



```
<div id="progre
<script>
    const progressStatus = document.getElementById('progressStat
```

```
    const eventSource = new EventSource("/stream-progress"); // Conn
to our stream!

    eventSource.onmessage = (event) => {
        const data = JSON.parse(event.data);
        progressStatus.textContent = `Progress: ${data.progress}% -
${data.step}`;
    };
</script>
```

When you open this page, `eventSource` connects to our endpoint, and the `progressStatus` div will update every second. Simple and effective!

# 3. The AI's To-Do List: A PocketFlow Workflow

So, how does our AI actually write an article? It doesn't happen in one go. We ne give it a step-by-step plan, like a recipe. This is where [PocketFlow](PocketFlow) helps us. It br the big job of "write an article" into small, manageable `Nodes` (or steps).

## The Central Hub: Our `shared` Dictionary

Before we dive into the nodes, let's look at the brain of our operation: a simple P dictionary. All our nodes will read from and write to this central `shared` data hu how they pass information to each other.

Here's what it looks like at the start of a job:

```
shared_data = {
    "topic": "The user's article topic",
    "sse_queue":
    "sections":
    "draft": ""
    "final_arti
}
```

The `sse_queue` is our "mailbox." Each node will drop progress updates into it, our FastAPI server will read from it to update the user.

## Step 1: `GenerateOutline` Node

This node's only job is to create the article's structure. (We'll assume a `call_ll` function exists that talks to the AI).

```
class GenerateOutline(Node):
    def prep(self, shared):
        return shared["topic"]

    def exec(self, topic):
        prompt = f"Create 3 section titles for an article on '{topic
        return call_llm(prompt) # e.g., "Intro,Main Points,Conclusio

    def post(self, shared, outline_str):
        sections = outline_str.split(',')
        shared["sections"] = sections

        progress = {"step": "outline", "progress": 33, "data": secti
        shared["sse_queue"].put_nowait(progress)
```

**What's happening here:**

- `prep` grabs the user's `topic` from the shared dictionary.
- `exec` takes that `topic` and asks the AI for an outline.
- `post` saves the outline for the next step and—most importantly—**drops a pr message into the mailbox**. This tells the frontend, "Hey, I'm 33% done!"

## Step 2: `Writ`

This node takes the

```
class WriteContent(Node):
    def prep(self, shared):
        return shared["sections"], shared["sse_queue"]

    def exec(self, prep_result):
        sections, queue = prep_result
        full_draft = ""
        for i, section in enumerate(sections):
            prompt = f"Write a paragraph for the section: '{section}
            paragraph = call_llm(prompt)
            full_draft += f"<h2>{section}</h2>\n<p>{paragraph}</p>\n

            progress = {"step": "writing", "progress": 33 + ((i+1)*2
            queue.put_nowait(progress)

        return full_draft

    def post(self, shared, full_draft):
        shared["draft"] = full_draft
```

**Why this is cool:**

- Inside the `exec` loop, after each paragraph is written, we immediately send another progress update.
- This means the user will see the progress bar jump forward multiple times d this step, making the app feel very responsive.

## Step 3: `ApplyStyle` Node

This is the final touch. It takes the combined draft and asks the AI to polish it.

```
class ApplyStyl
    def prep(se
        return

    def exec(se
        prompt =
{draft[:500]}"
```

```
        return call_llm(prompt)

    def post(self, shared, final_article):
        shared["final_article"] = final_article

        progress = {"step": "complete", "progress": 100, "data":
final_article}
        shared["sse_queue"].put_nowait(progress)
```

**The grand finale:**

- This node does the final rewrite.
- Crucially, it sends the `complete` message to the mailbox with `progress: 100`. This tells our frontend that the job is finished and the final article is ready!

## Tying It All Together with a `Flow`

Finally, we just need to tell PocketFlow the order of our to-do list.

```
from pocketflow import Flow

def create_article_flow():
    outline_node = GenerateOutline()
    content_node = WriteContent()
    style_node = ApplyStyle()

    # Define the sequence: outline -> write -> style
    outline_node >> content_node >> style_node

    return Flow(start_node=outline_node)
```

This is super readal `style_node`. This run.



Here is a visual summary of the entire process:


With our AI's "to-do list" ready, let's connect it to our FastAPI backend.

# 4. Connecting the Dots: The FastAPI Backend

Okay, our AI has its "to-do list" from PocketFlow. Now, let's build the web serve acts as the project manager. It will take requests from users, give the work to our and report on the progress.

We'll walk through the main `main.py` file piece by piece.

## Part 1: The Job Center

First, we need a place to keep track of all the article-writing jobs that are current running. A simple Python dictionary is perfect for this.

```
# A dictionary to hold our active jobs
# Key: A unique job_id (string)
# Value: The "mailbox" (asyncio.Queue) for that job's messages
active_jobs = {}
```

Think of `active_jobs` as the front desk of an office. When a new job comes in, give it a ticket number (`job_id`) and a dedicated mailbox (`asyncio.Queue`) for internal memos.

## Part 2: Kickin Off the Job

This is the first thir
`/start-job` endp

```
@app.post("/start-job")
async def start_job(background_tasks: BackgroundTasks, topic: str =
Form(...)):
    job_id = str(uuid.uuid4())

    # Create a new, empty mailbox for this specific job
    sse_queue = asyncio.Queue()
    active_jobs[job_id] = sse_queue

    # Tell FastAPI: "Run this function in the background"
    background_tasks.add_task(run_article_workflow, job_id, topic)

    # IMMEDIATELY send a response back to the user
    return {"job_id": job_id}
```

**Let's break that down:**

1. **Get a Ticket Number:** `job_id = str(uuid.uuid4())` creates a unique t this request.
2. **Create the Mailbox:** `sse_queue = asyncio.Queue()` creates the messa queue. We then store it in our `active_jobs` dictionary using the `job_id`.
3. **Hand off the Work:** `background_tasks.add_task(...)` is the magic. I FastAPI, "Don't wait! After you send the response, start running the `run_article_workflow` function."
4. **Instant Reply:** The `return {"job_id": job_id}` is sent back to the use browser right away. The user's page doesn't freeze!

## Part 3: The Background Worker

This is the function ac runs our PocketFlo



```
def run_article
    sse_queue = active_jobs[job_id]
    shared = {
```

```
        "topic": topic,
        "sse_queue": sse_queue, # Here's where we pass the mailbox i
        "sections": [],
        "draft": "",
        "final_article": ""
    }

    flow = create_article_flow()
    flow.run(shared) # Start the PocketFlow!
```

**Here's the crucial connection:**

- It gets the correct `sse_queue` (mailbox) for this job from our `active_jobs` dictionary.
- It creates the `shared` data dictionary and **puts the `sse_queue` inside it.**
- When `flow.run(shared)` is called, our PocketFlow nodes now have acces this queue and can drop their progress messages into it!

## Part 4: Streaming the Progress Updates

While the background task is running, the user's browser connects to our `/progress/{job_id}` endpoint to listen for updates.

This function looks a bit complex, but it's just a loop that checks the mailbox.

```
@app.get("/progress/{job_id}")
async def get_progress(job_id: str):

    async def event_stream():
        # First, find the right mailbox for this job
        if job_
            yie
            ret

        sse_que

        while True:
```

```
            # Wait for a new message to arrive in the mailbox
            progress_msg = await sse_queue.get()
            yield f"data: {json.dumps(progress_msg)}\n\n"

            # If the message says "complete", we're done!
            if progress_msg.get("step") == "complete":
                del active_jobs[job_id] # Clean up the job
                break

    return StreamingResponse(event_stream(), media_type="text/event-stream")
```

**The logic is simple:**

1. `event_stream` is a special `async` generator that can send (`yield`) data ov time.
2. It finds the correct mailbox (`sse_queue`) for the `job_id`.
3. The `while True:` loop starts.
4. `await sse_queue.get()` **pauses and waits** until a message appears in th mailbox.
5. As soon as a PocketFlow node drops a message in, this line wakes up, grabs message, and sends it to the browser with `yield`.
6. It keeps doing this until it sees the `"step": "complete"` message, at whi point it cleans up and closes the connection.

And that's the whole system! It's a clean loop: the user starts a job, a background worker runs it while dropping messages into a mailbox, and a streamer reads fro mailbox to keep the user updated.

## 5. Mission ( s ' Heavy Lifti



You did it! You've s AI without breaking a sweat or frustrating your users. They get an instant resp

live progress updates, making the whole experience feel smooth, interactive, and professional.

No more dreaded loading spinners or "page unresponsive" errors. Your app now like a modern, intelligent assistant: it acknowledges your request, works on it diligently in the background, and keeps you informed every step of the way.

**What you conquered today:**

- ✅ **No More Freezing UIs:** Used FastAPI's `BackgroundTasks` to offload h AI work so your app stays responsive.
- ✅ **Live Progress Updates:** Mastered Server-Sent Events (SSE) to stream stat updates from the server to the browser in real-time.
- ✅ **Clean, Organized Logic:** Structured a complex, multi-step AI job with PocketFlow, keeping your AI logic separate from your web code.
- ✅ **Tied It All Together:** Used an `asyncio.Queue` as a simple "mailbox" to your background task communicate with your web server.

This architecture is a game-changer for building serious AI applications. You no have the skills to create tools that can generate reports, analyze data, or perform other time-intensive task, all while keeping your users happy and engaged.

**Our "Build an LLM Web App" Journey is Complete!**

- Part 1: Command-line AI tools ✅
- Part 2: Interactive web apps with Streamlit ✅
- Part 3: Real-time streaming chat with WebSockets ✅
- **Part 4 (You jus**



***Ready to see it all i*** ***ke***
*nodes, from our coo* ***e l***

*up your AI dev skills in a big way. Now go build something amazing!* 🚀

---

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

---

8 Likes · 1 Restack

← Previous Nex

## Discussion about this post

Comments Restacks

Write a comment...

---