Streaming LLM Responses — Tutorial For Dummies (Using PocketFlow!)

ZACHARY HUANG
MAY 06, 2025

> *Tired of staring at a loading spinner while the AI thinks? Wish you could just yell "Sto when it goes off track? This guide shows you how to get AI answers *instantly* by* **streaming LLM responses** word-by-word, and cut them off anytime, using a sir [PocketFlow LLM Streaming Example](#).*

See AI Answ Brakes!)

We've all been there. ıay it's... weird. What if you could see the answer appear *as it's typed* and hit an "e" "

button if it's not what you want?

That's **LLM Streaming** (seeing it live) and **User Interruption** (hitting stop). Instea getting a whole essay dropped on you, text pops up piece by piece. It feels *way* fast Plus, the stop button puts *you* in charge, saving time and maybe even cash on API calls.

What you'll learn in this easy guide:

- Why watching AI type is awesome.
- The simple idea behind showing text live while listening for your "STOP!" command.
- How to build this feature with just one simple PocketFlow `Node`.

We'll use the [PocketFlow LLM Streaming example](). PocketFlow helps organize the steps s see exactly how the magic happens without getting lost in code spaghetti. Let's make AI less like waiting for dial-up and more like a chat!

Why Stream? Why Interrupt? The Benefits

Think about ChatGPT typing out answers. That's **streaming**. Being able to stop it **Interruption**. Here's why they rock:

- **Feels Faster (Be** Ugh. Streaming and keeps you er You see the func

- **Get the Point Faster:** Sometimes you just need one quick thing. **Example:** "W the weather in London?" Stream starts: *"In London, it's currently 15°C..."* Perfe Maybe you don't care about the humidity forecast for next Tuesday. Streamin gets you the core info ASAP.
- **Stop the Nonsense (Control & Savings):** AI can ramble or get stuck. Interrupt is your "Okay, buddy, that's enough" button. **Example:** Ask for 10 marketing slogans. The third one is GOLD! Hit stop. Why wait (and pay) for 7 more medi ones? Grab the winner and go!
- **Change Your Mind Mid-Stream:** Ideas change! **Example:** You ask for "things do in Paris." It starts listing museums. You see "Louvre" and think, "Actually, want *outdoor* stuff!" Interrupt, ask about parks instead, and get relevant info f

Streaming + Interruption = AI that feels like a conversation, not a lecture you can' escape.

How It Works: Live Feeds & Listening Skills

Getting this live text + stop button involves two main tricks: how the AI sends dat differently, and how our code juggles showing text while listening for your "STOP command.

1. From Snail Mail to Live TV: `stream=False` vs. `stream=True`

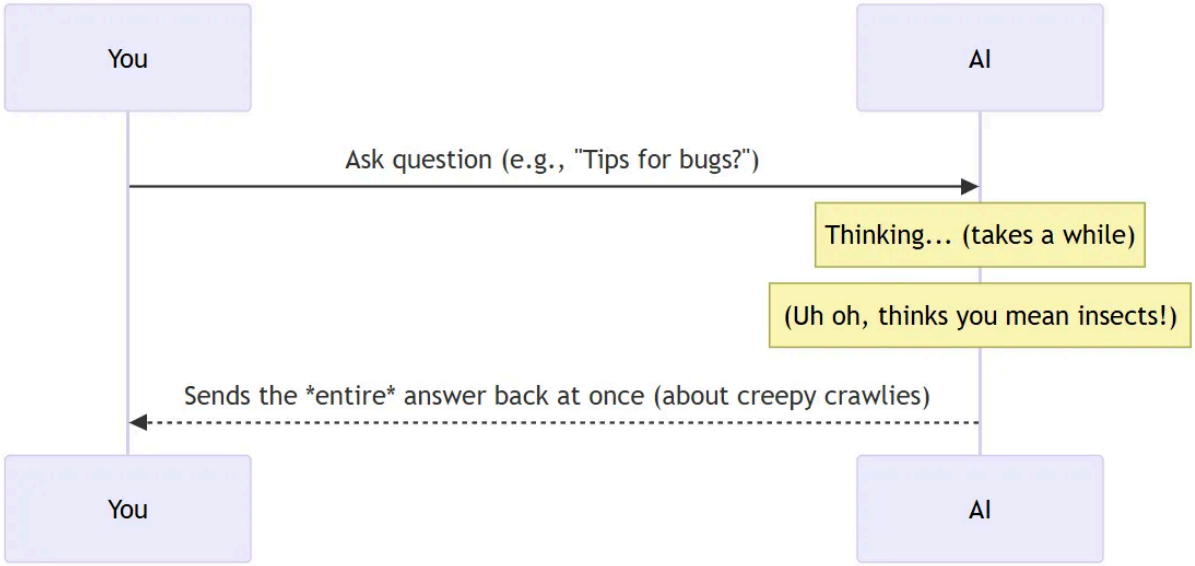
Think about asking an AI for help. The key is one little switch in the code: `stream=True`.

```
# Simplified concept of calling the AI
from openai impo

# The magic swit
response = clien
    model="gpt-4
    messages=[{"
    stream=True # <--- THIS IS THE KEY!
```

```
)

# Now 'response' isn't the final answer... it's a live feed!
```

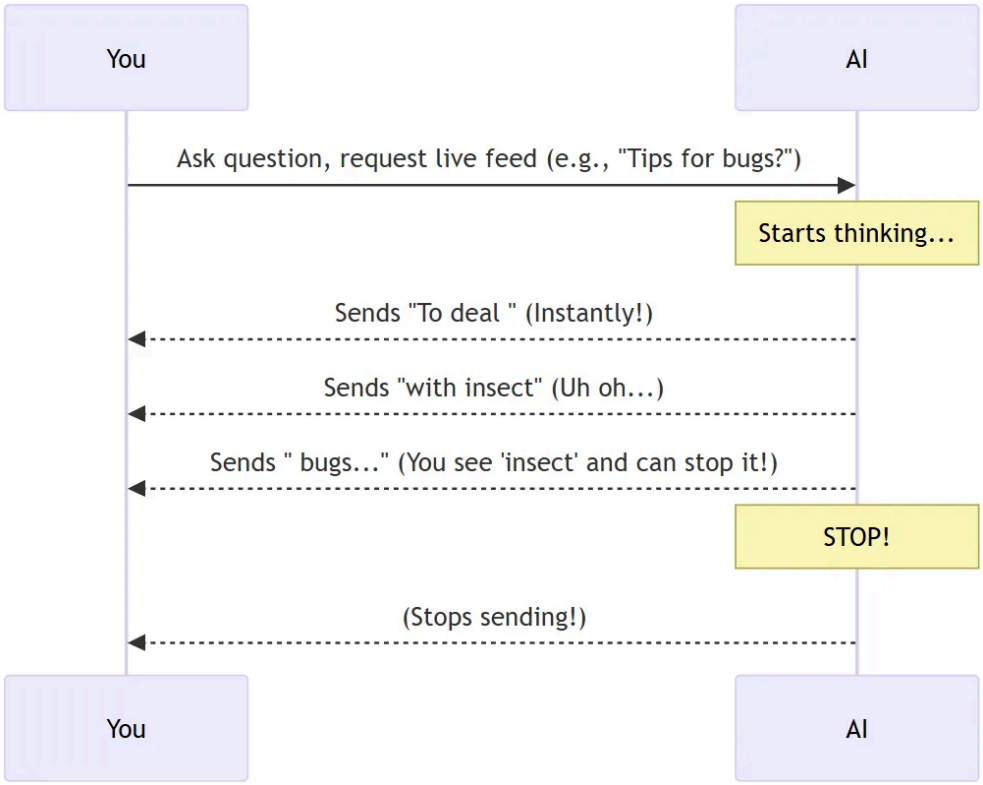
Let's imagine you ask: "Give me tips for dealing with bugs." You mean *software* bu

(A) The Old Way: `stream=False` (Like Sending a Letter)



- **What Happens:** You ask, you wait. The AI misunderstands, writes a whole an about insects, *then* sends it all back. You waited 10 seconds for useless advice. Annoying!
- **Data:** You get one big chunk of text after the delay.
- **Experience:** Wait... wait... BAM! "Seal cracks in your home..." Ugh, wrong bug Total waste of time.

(B) The Live Way: `stream=True` (Like a Phone Call)



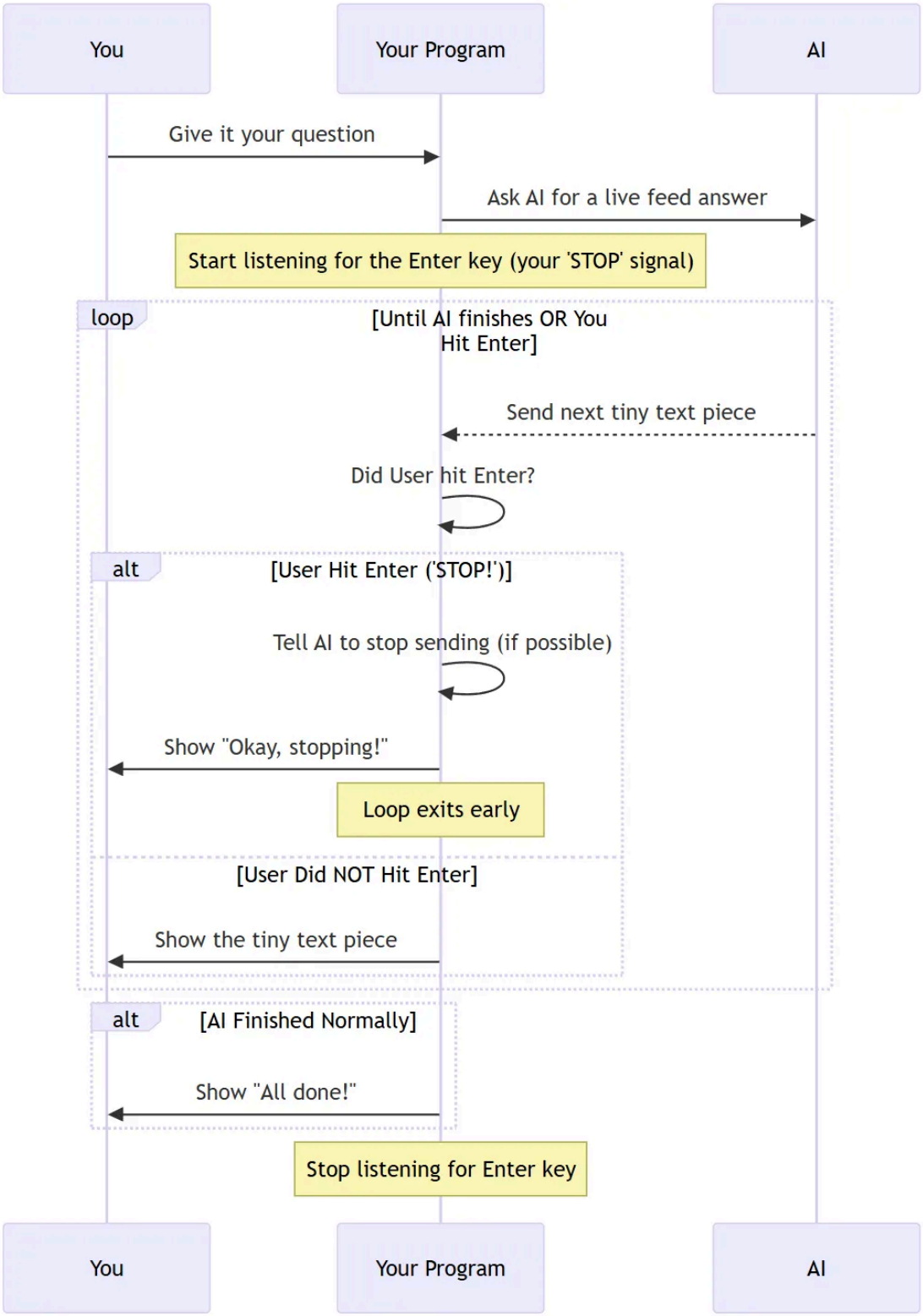
- **What Happens:** You ask. The AI sends back a *live feed* (an "iterator" or "generator" in code terms). Your code starts getting tiny pieces *immediately*.
- **Data:** It's like a conveyor belt delivering small packages (chunks) one after anc

```
# Conceptual idea: Looping through the live feed
for chunk in response:
    content = chunk.choices[0].delta.content or ""
    print(content, end="", flush=True)
    # (Add a check here to see if user wants to stop)
    if user_interrupts: break
```

- **Experience:** Text appears right away: "To deal with insect..." You see "insect" think, "Nope!" Because it's arriving piece by piece, you have the *chance* to interrupt it.

2. The Jugglin

Okay, `stream=True` r y hitting Enter (because you saw "insect"). Our program needs to multitask.



The Simple Steps:

1. **You Ask:** Give th
2. **Program Starts '**
   - Asks the AI i

- *At the same time*, starts listening for you to press ENTER.

3. **The Loop (Show & Check):**
   - AI sends a tiny text piece.
   - Program *instantly* checks: "Did they hit ENTER?"
   - **YES?** --> Tell AI to stop, show "Stopped!", end of story.
   - **NO?** --> Show the text piece, wait for the next one. Repeat fast!
4. **Finish Line:** If the AI sends everything *without* you hitting ENTER, show "Don Either way, stop listening for ENTER.

---

*This sounds tricky to code, right? Juggling two things at once? This is where a simple tool **PocketFlow** makes life much easier by giving us a structure.*

Making It Happen with PocketFlow

We want the live text + stop button. PocketFlow helps organize this.

PocketFlow: Simple Recipe Cards (Nodes)

Imagine PocketFlow gives you recipe cards, called **Nodes**. Each Node has 3 simple steps:

1. `prep`: Get your ingredients ready.
2. `exec`: Do the main cooking steps.
3. `post`: Clean up the kitchen.

This keeps complex t


```
# Super simplifie
class MyTaskNode:
    # 1. Get ready
```

```
    def prep(self, shared):
        # ... setup code goes here ...
        return ingredients_for_exec

    # 2. Do the main work
    def exec(self, ingredients_from_prep):
        # ... main logic goes here ...
        return results_for_post

    # 3. Tidy up
    def post(self, shared, ingredients, results):
        # ... cleanup code goes here ...
        pass

    def run(self, shared):
        p = self.prep(shared)
        e = self.exec(p)
        return self.post(shared, p, e)
```

- **Shared Store:** Think of `shared` as a shared pantry to grab things from (`prep`) put things back into (`post`).

The Secret Signal: `threading.Event` (Like Raising You Hand)

How does the part showing text know the *other* part listening for Enter wants to st They need a signal. `threading.Event` is perfect.

Think of it like **raising your hand in class**:

- The listener (when you press Enter) **raises the hand**: `signal.set()`
- The text-shower `signal.is_se`


```
# --- Conceptual
import threading
```

```
import time

# The shared signal (hand is initially down)
stop_signal = threading.Event()

# Background job: Student waits, then raises hand
def student_action():
    print("Student: Waiting 3 secs...")
    time.sleep(3)
    print("Student: Raising hand! (Signaling stop)")
    stop_signal.set() # <--- Hand goes UP!

# Start the student job
student_thread = threading.Thread(target=student_action)
student_thread.start()

print("Teacher: Starting class (checking for hand)...")
# Teacher keeps checking the signal while teaching
for i in range(10): # Let's pretend class has 10 parts
    # ---> Check the signal BEFORE teaching next part <---
    if stop_signal.is_set(): # <--- Is hand raised?
        print("Teacher: Hand raised! Stopping class.")
        break # Stop teaching
    # ---> If no hand raised, teach the next part <---
    print(f"Teacher: Teaching part {i+1}...")
    time.sleep(0.8) # Simulate teaching

print("Teacher: Class ended.")
student_thread.join() # Wait for student thread to fully stop
print("Everyone dismissed.")
# --- End Example ---
```

Run this! You'll see the "Teacher" teaching parts, but after the "Student" signals (raises hand), the class will end early. The `Event` is our simple stop signal.

Expected Output (ti


```
Teacher: Startin
Student: Waiting
Teacher: Teaching part 1...
Teacher: Teaching part 2...
```

```
Teacher: Teaching part 3...
Student: Raising hand! (Signaling stop)
Teacher: Teaching part 4...
Teacher: Hand raised! Stopping class.
Teacher: Class ended.
Everyone dismissed.
```

Building the `StreamNode`: Our Recipe Card

Let's write the PocketFlow `Node` for our streaming task.

Step 1: `prep` - Get Ingredients

We need:

1. The AI's live feed (the `stream` from `stream_llm`).
2. The hand-raise signal (`threading.Event`).
3. Someone listening for Enter in the background.

```
# --- StreamNode: prep ---
import threading
from pocketflow import Node
# from utils import stream_llm # Assume this gets the AI stream

class StreamNode(Node):
    def prep(self, shared):
        prompt = shared["prompt"] # Get the question
        print("Requesting stream...")
        # Start the AI stream (this returns the live feed/iterator)
        chunks_iterator = stream_llm(prompt)

        # Create
        interrupt

        # Define                                                    ha
        def liste
            inpu
            print("--- Enter pressed! Sending stop signal ---")
```

```
            interrupt_event.set() # Raise the hand!

        # Start the listener job in the background
        print("Listener started...")
        listener_thread = threading.Thread(target=listen_for_enter,
daemon=True)
        listener_thread.start()

        # Pass the live feed, signal, and listener thread to the 'exec
step
        return chunks_iterator, interrupt_event, listener_thread
```

`prep` is like getting set up. It grabs the user's `prompt`, calls the AI with `stream=True` to get the `chunks_iterator` (our live feed), creates the `interrupt_event` signal (like an empty flag pole), and starts a background helpe (`listener_thread`) whose only job is to wait for you to press Enter, then raise tl flag (`interrupt_event.set()`). It hands off the feed, the flag, and the helper tc next step (`exec`).

Step 2: exec - The Main Cooking Loop

Here's where we show the text, piece by piece, but *always* check the signal flag firs

```
# --- StreamNode: exec ---
# (Continuing the StreamNode class)
    def exec(self, prep_res):
        # Get the ingredients from prep
        chunks, interrupt_event, listener_thread = prep_res

        print("Streaming response:")
        stream_finished_normally = True # Assume it will finish ok

        # Loop th
        for chunk
            # --
            if in
                p
                s
                break # STOP THE LOOP!
```

```
            # ----> If flag not raised, show the text <----

            # Get the text bit from the chunk and print it
            # (Real APIs might need slightly different code here)
            content = chunk.choices[0].delta.content or ""
            print(content, end="", flush=True) # Show text immediately

        if stream_finished_normally:
            print("--- Stream finished ---")

        # Pass the signal flag and listener thread to cleanup
        return interrupt_event, listener_thread
```

`exec` takes the live feed (`chunks`), signal (`interrupt_event`), and listener helpe (`listener_thread`) from `prep`. It loops through each `chunk` arriving from the **Crucially**, the very first thing inside the loop is checking `if interrupt_event.is_set()`. If the flag is up (you hit Enter), it prints an inter message, notes that it didn't finish normally, and `break`s out of the loop immedia If the flag *isn't* up, it pulls the text out of the `chunk` and `print`s it (using `end="" flush=True` makes it appear right away on the same line). If the loop finishes without being interrupted, it prints a "finished" message. Finally, it passes the sig and listener to `post` for cleanup.

Step 3: `post` - Clean Up the Kitchen

Whether the stream finished or was interrupted, make sure the background listen stops cleanly.

```
# --- StreamNode: post ---
# (Continuing the StreamNode class)
    def post(self
        # Get the
        interrupt

        # Ensure
        # Signal
        interrupt_event.set()
```

```
        # Wait briefly for the listener thread to finish
        listener_thread.join(timeout=1.0)
        print("Listener stopped.")
```

`post` is cleanup time. It gets the signal flag (`interrupt_event`) and the listener helper (`listener_thread`). It makes sure the signal flag is raised (`interrupt_event.set()`), just to be absolutely sure the listener (which is wait for `input()`) gets the message to stop. Then, `listener_thread.join()` tells t main program to wait politely (up to 1 second) for the listener thread to pack up ar home. This prevents leftover processes hanging around.

And that's our `StreamNode`! PocketFlow's `prep`/`exec`/`post` structure helps keep t potentially tricky logic neat and tidy.

Running the Example Yourself

Okay, theory's done, let's see it run!

1. **Get the Code:** Grab the complete example from GitHub: [PocketFlow LLM Streaming Example](#).
2. **Install Stuff:** Open your terminal, go into the example folder, and run:

```
pip install -r requirements.txt
```

3. **Run It!**

```
python main.py
```


What You'll See:

First, the setup messages:

```
Listener started...
Press ENTER anytime to stop...

Requesting stream...
Streaming response:
```

Then, text will start appearing on one line:

```
This is a streaming response from LLM. Today is a sunny day. The sun i
shining...
```

- **Try Interrupting:** Hit ENTER while it's typing. Boom! The text stops instantly!

```
--- Enter pressed! Sending stop signal ---

--- Interrupted by user ---
Waiting for listener to stop...
Listener stopped.
```

- **Let it Finish:** Run it again, but don't press anything. It will type out the full message, then:

```
--- Stream finished ---
Waiting for listener to stop...
Listener stopped.
```

Conclusion

So there you have it!

1. **LLM Streaming** (`stream=True`): Get text piece by piece.
2. **Background Listening** (`threading`): Watch for the Enter key.
3. **Simple Signal** (`threading.Event`): A basic flag to say "STOP!".
4. **Organized Code** (**PocketFlow Node**): Keep the setup, work, and cleanup tidy.

...you can make AI interactions feel *way* faster and put yourself back in the driver's seat. Stop the AI when *you* want to!
Using PocketFlow just makes managing the tricky bits (like background tasks) mu cleaner with its simple `prep`/`exec`/`post` recipe.
Go grab the code, play around, and build AI tools that feel snappy and responsive!

---

*Want more? Check out **The Code** at the [GitHub: PocketFlow LLM Streaming Cookbo](#) explore the **PocketFlow Framework** on the [PocketFlow GitHub Repo](#), or **Chat with Us** or [PocketFlow Discord](#). Happy streaming (and interrupting)!*

Write a comment...

TRAN QUANG THIEN May 10

Liked by Zachary Huang

Thanks for the wonderful post. Just one question. What will happen if we don't raise interrupt_event.set() in the post processing part?

LIKE (1) REPLY

1 reply by Zachary Huang

1 more comment...