Build Your Own Voice Chatbot From Scratch — PocketFlow Tutorial!

ZACHARY HUANG
MAY 15, 2025

16 1 Sl

> *Ever chatted with your smart speaker and wondered, "How'd it DO that?" Or maybe y dreamed of building your own voice assistant, like the cool one in our [PocketFlow Voic Chat Cookbook](#)? This guide is your ticket! We'll build an AI Voice Chatbot from zero, the super-duper simple [PocketFlow](#) framework.*

1. Hello, Voi

Ever talked to your p the weather? Yep, that's ng our voice. And guess what? It's popping up everywhere: in our phones (think Siri

Google Assistant), smart speakers (like Alexa), cars, and even when you call custon service.

Why's everyone gabbing with their tech?

- **It Just Feels Right:** Talking is human! It's often easier than typing or clicking.
- **Hands-Free, Eyes-Free!** Awesome for when you're busy, like cooking or drivir Plus, it helps more people use tech easily.
- **Super Speedy:** Saying what you want is often quicker than navigating menus.

Ready to build one? Sweet! In this tutorial, **you're going to create your very own v chatbot.** We're talking an app that listens, gets what you're saying, thinks, and cha right back. We'll snag some cool tricks and code snippets from our [PocketFlow Vo Chat Cookbook] to make it easier.

Our main tool for this quest? [PocketFlow]! Think of PocketFlow as your easy-peas toolkit for building tricky apps (like voice ones!) without the usual headaches. No confusing code jungles here! PocketFlow chops the big job into small, clear steps, like a recipe. It's your simple map from "you talking" to "app talking back smartly

---

Let's give your apps a voice!

---

2. The Magic of Voice: How's It Actually Work?

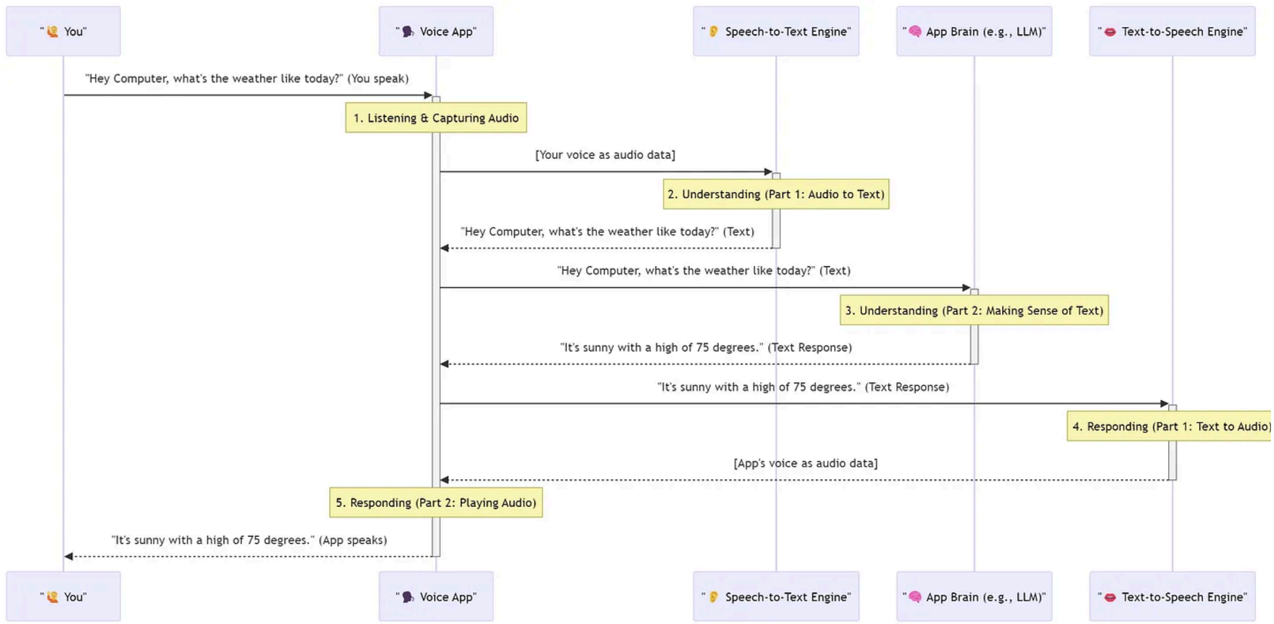
Ever wonder what's l t q sorcery, but more lik p needs to:

1. **Listen** to you.

2. **Understand** what you said.
3. **Talk back** helpfully.

Let's imagine you ask your app:
You: "Hey Computer, what's the weather like today?"

Here's the behind-the-scenes journey:



Let's walk through that journey step-by-step:

1. **You Speak & The App Listens (The "Listening" Part):**
   - You kick things off: "Hey Computer, what's the weather like today?"
   - The Voice App is all ears! It grabs your voice and turns it into digital soun data. Think of it like hitting "record" on a tape recorder.
2. **Sound to Words (Speech-to-Text – First part of "Understanding"):**
   - Our app is sm... but it doesn't understand raw sound. So, it sends your recorded voi...
   - This STT en... dic... types out the ... e `today?"`.

3. **Figuring Out Your Request (The "App Brain" – Second part of "Understanding")**:
    - Now that your question is in text form, the Voice App passes it to its **App Brain**. This brain could be simple logic for basic commands, or for more complex chats (like ours!), it's often a Large Language Model (LLM).
    - The App Brain deciphers your request (you want the weather forecast!) an prepares a text answer, something like: `"It's sunny with a high o 75 degrees."`.
4. **Words Back to Sound (Text-to-Speech – First part of "Responding")**:
    - You asked with your voice, so you probably want to hear the answer, not r it. The app sends the text response to a **Text-to-Speech (TTS) Engine**.
    - The TTS engine does the reverse of STT. It takes the written words and generates spoken audio, complete with a voice.
5. **The App Talks Back! (Playing Audio – Second part of "Responding")**:
    - Finally, the Voice App plays this newly created audio. You hear: "It's sunn with a high of 75 degrees."
    - And voilà! The conversation cycle is complete.

So, it's not quite magic, but a well-orchestrated flow: your app listens to your audio, chan to text, figures out what you mean, drafts a text reply, turns that reply into audio, and t plays it for you. Each step is a crucial piece of the puzzle for a smooth, human-like voi interaction. In the next sections, we'll explore the specific components that handle each these important jobs.

---

3. Core Con

Alright, let's pop the k together to make our members, each with a critical job to do in our voice interaction assembly line

1. Voice Activity Detection (VAD) – The Smart Listener

Ever notice how your smart speaker only perks up when you *actually* say somethin and not when the TV is blaring or you're just quietly sipping your coffee? That's **V Activity Detection** (**VAD**) in action! It's like a smart bouncer for your app's microphone, deciding when to hit "record" because it hears real speech, and wher chill because it's just background noise or silence. It does this by checking the ene of the sound it picks up.

Here's how we can build a basic VAD to record speech. We'll break the code dowr small, easy-to-digest pieces.

Piece 1: Setting up the Recorder

```
import sounddevice as sd
import numpy as np

# Our VAD recorder function
def record_with_vad(
    sample_rate=44100,      # How many sound snapshots per second
    chunk_size_ms=50,       # How big is each audio piece we check (ir
milliseconds)
    silence_threshold_rms=0.01, # How quiet is "silence"?
    min_silence_duration_ms=1000 # How long silence means "user stoppe
talking"
):
    recorded_frames = []    # This will store our recorded audio
    is_recording = False    # Are we currently recording?
    silence_counter = 0     # Counts how long it's been silent

    # Convert ms to frames/chunks for sounddevice
    chunk_size_frames = int(sample_rate * chunk_size_ms / 1000)
    min_silence_chunks = int(min_silence_duration_ms / chunk_size_ms)

    print("🎙️  Sl
    # Open the m:
    # 'with' ensu
    with sd.Input
```

```
blocksize=chunk_size_frames) as stream:
        # ... more code inside the loop coming next!
```

First, we set up our `record_with_vad` function. It has a few important settings, knobs on an old radio, that tell it *how* to listen:

- `sample_rate` (e.g., `44100` Hz): This is like the number of "snapshots" your r takes of the sound *every second*. More snapshots mean clearer, higher-quality a – think a high-res photo vs. a blurry one! 44,100 is a common standard.
- `chunk_size_ms` (e.g., `50` milliseconds): Our app doesn't listen to everything once; it listens in tiny "chunks." This setting decides how big each chunk is.
- `silence_threshold_rms` (e.g., `0.01`): This is our "shhh!" level. If a sound chunk's energy is below this, we consider it silence.
- `min_silence_duration_ms` (e.g., `1000` milliseconds): This is the "are you yet?" timer. If it stays quiet for this long, the app assumes you've stopped talki

Inside the function, we prepare to store `recorded_frames` (that's your voice!) ar use `is_recording` and `silence_counter` to keep track. We also do a quick calculation to turn our millisecond settings into something `sounddevice` understands (frames/chunks). Finally, `sd.InputStream` opens up the microphon The `with` keyword is neat because it makes sure the mic is properly closed when done.

Piece 2: The Listening Loop and Energy Check

(Continuing from inside the `with sd.InputStream...` block from the previou snippet)

```
        # ... (p
sd.InputStream..
        # This l
        while Tru
            # Rea
            # audio_chunk_np is an array of numbers representing the
sound
```

```
            audio_chunk_np = stream.read(frames=chunk_size_frames)[0]

            # Calculate the 'energy' (RMS) of this audio chunk
            # Big RMS = loud sound (maybe speech!), Small RMS = quiet
(maybe silence!)
            rms = np.sqrt(np.mean(audio_chunk_np**2))

            # ... logic for starting/stopping recording based on RMS
comes next!
```

Now for the main action! The `while True:` loop means our app is constantly listening. In each round of the loop:

1. `stream.read(...)` grabs a tiny piece of sound from your microphone. This sound data comes in as a list of numbers (a NumPy array, which we call `audio_chunk_np`).
2. Then, `rms = np.sqrt(np.mean(audio_chunk_np**2))` is like our VAI ear. It calculates the "Root Mean Square" (RMS) of the audio chunk. Fancy na simple idea: it tells us the average energy or loudness of that tiny sound piece. higher RMS means more energy (likely speech!), and a lower RMS means less energy (likely silence!).

Piece 3: Deciding When to Record (The VAD Logic)
(Continuing from inside the `while True:` loop from the previous snippet)

```
            # ... (RMS calculation from Piece 2) ...

            if is_recording:
                # If we are already recording...
                recorded_frames.append(audio_chunk_np) # Add the curre
sound to our col
                :
                                                                      if
been quiet long e
                      print("🤫 Silence detected, stopping
```

```
recording.")
                            break # Stop the loop (and thus recording)
                    else:
                        silence_counter = 0 # Oh, you're talking again!
Reset silence count.

                elif rms > silence_threshold_rms: # If NOT recording, and
it's loud enough...
                    print("🎉 Speech detected, starting recording!")
                    is_recording = True       # Start recording!
                    recorded_frames = [audio_chunk_np] # Begin with this
first speech chunk
                    silence_counter = 0       # Reset silence counter

    # After the loop finishes (because of 'break')...
    return recorded_frames # Send back all the recorded voice!
```

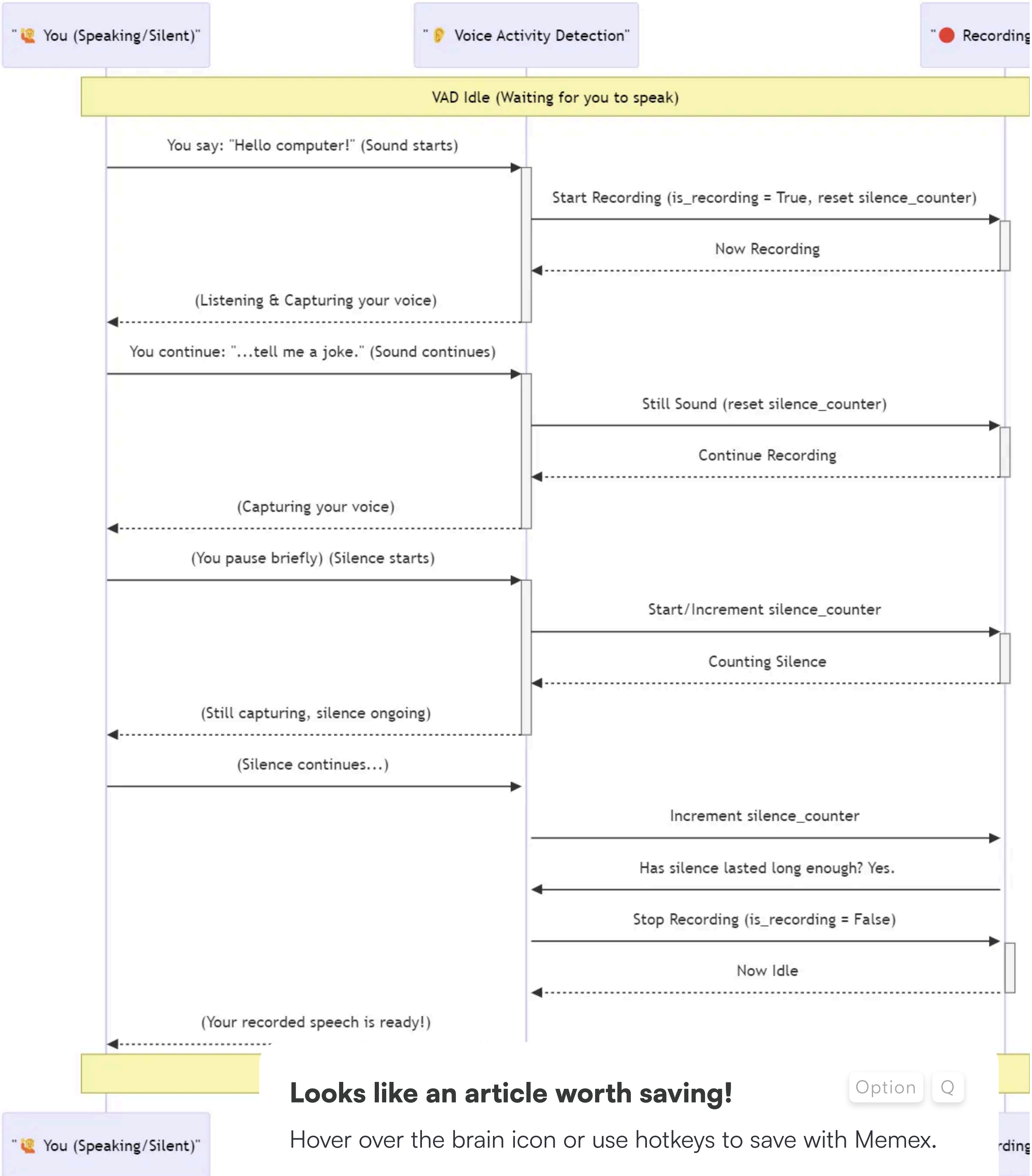
This is where the VAD magic happens, based on that RMS energy we just calculat

- **If `is_recording` is `True` (we're already capturing your voice):**
  - We keep adding the incoming `audio_chunk_np` to our `recorded_fra`
  - If the `rms` drops below our `silence_threshold_rms` (it got quiet), we s counting (`silence_counter`).
  - If it stays quiet for long enough (`silence_counter >= min_silence_chunks`), we print a message and `break` out of the `whil True` loop, meaning recording is done for this segment.
  - If it gets loud again while we were counting silence, we reset `silence_counter` because you're still talking!
- **Else if `is_recording` is `False` (we're waiting for you to speak) AND `rms` i high enough:**
  - Woohoo! Spe irs chunk of spe

Once the loop breaks ns the `recorded_frames` it collected. That's your voice, ready for the next ster

Visualizing the VAD Logic (Walking through the code):

That's how our VAD code figures out when you're talking! To see the overall flow these decisions, let's look at a quick diagram:



In summary, the `record_with_vad` function continuously listens to the micropl in small chunks. It calculates the RMS energy of each chunk. If speech is detected (energy goes above `silence_threshold_rms`), it starts recording. If it's already recording and detects a period of silence (energy stays below the threshold for `min_silence_duration_ms`), it stops that recording segment and returns the captured audio frames.

2. Speech-to-Text (STT) – The Transcriber

Alright, our app has "heard" you, thanks to VAD. But computers are way better at understanding written words than raw sound waves. That's where the **Speech-to-T** **(STT)** engine swoops in – it's like a super-fast, accurate typist for audio!

Here's how we can get text from our recorded audio using OpenAI:

```
import io # We'll use this small helper

# And 'audio_bytes' contains the sound data from VAD (e.g., in WAV
format)
def speech_to_text_api(audio_bytes):
    if not audio_bytes:
        print("🤷 No audio to transcribe!")
        return None

    print("🗣️➡️📄 Sending audio to OpenAI for transcription...")
    # We need to send the audio as if it's a file
    # io.BytesIO helps us treat our 'audio_bytes' like a file in memo
    audio_file_like = io.BytesIO(audio_bytes)
    audio_file_like.name = "input.wav" # Giving it a filename helps tl
API

    response = client.audio.transcriptions.create(
        model="gı                                                    ech
to text
        file=aud:
    )
    print(f"💬 O|
    return respoı
```

Let's break that down:

1. Our function `speech_to_text_api` needs the `client` (our connection to OpenAI) and the `audio_bytes` (your recorded voice).
2. We use `io.BytesIO(audio_bytes)` to wrap our audio data. Think of it lik putting your audio into a digital envelope. Giving it a `.name` like `"input.wa` like labeling the envelope – it helps the OpenAI service understand what kind audio it is.
3. Then, `client.audio.transcriptions.create(...)` is the magic call. tell it which AI model to use (`"gpt-4o-transcribe"` is a powerful one) anc give it our "audio envelope."
4. OpenAI listens to the audio, types out what it hears, and sends back the `response`. We just grab the `response.text`.

And just like that, your spoken words are now text, ready for the app's brain!

3. Command Processing / LLM Interaction – The Brain

Now that we have text, it's time for our app to *think* and figure out what to say bac This is where the "brain" of our chatbot comes in – usually a **Large Language Mo** (**LLM**). Imagine it as a super-smart, quick-witted assistant who has read tons of bo and conversations, ready to chat about almost anything.

First, we need to prepare the full conversation history, including the user's latest message, in a format the LLM understands. This is typically a list of messages, eac marked with who said it ("user" or "assistant"):

```
# 'user_query' is
# 'chat_history'
# [{"role": "use
#  {"role": "ass
playground? ..."]
```

```
messages_for_llm = chat_history + [{"role": "user", "content":
user_query}]
```

With our `messages_for_llm` ready, here's how we get a response using OpenAI

```
def call_llm(messages_for_llm):
    if not messages_for_llm:
        print("🤔 No messages for the LLM.")
        return None

    print(f"🧠 Sending to LLM: latest query '{messages_for_llm[-1]
['content']}'")
    response = client.chat.completions.create(
        model="gpt-4o",      # A powerful chat model from OpenAI
        messages=messages_for_llm
    )

    llm_reply = response.choices[0].message.content
    print(f"💡 LLM replied: {llm_reply}")
    return llm_reply
```

What's happening here?

1. Our `call_llm` function now needs the `client` and the complete `messages_for_llm` list.
2. It sends this entire conversation to a chat model like `"gpt-4o"` using `client.chat.completions.create(...)`.
3. The LLM thinks and generates a reply based on the whole conversation. We g this reply from `response.choices[0].message.content`.

Cool! The app now h

4. Text-to-Spe Speaker

The app has figured out *what* to say (thanks, LLM!), but that reply is still just text. want our chatbot to *talk* back! This is the job of the **Text-to-Speech** (**TTS**) engine. Think of it as a talented voice actor who can read any script you give them, in vari voices.

Let's make OpenAI say our LLM's response out loud:

```
def text_to_speech_api(text_to_speak, voice_choice="alloy"):
    if not text_to_speak:
        print("🔇 Nothing to say for TTS.")
        return None

    print(f"📄➡️🗣️ Sending to OpenAI TTS: '{text_to_speak[:30]}...'")
    response = client.audio.speech.create(
        model="gpt-4o-mini-tts", # A good TTS model
        voice=voice_choice,   # You can pick different voices!
        input=text_to_speak,
        response_format="mp3" # We want the audio back as an MP3
    )

    # The audio comes back as raw bytes (a sequence of 0s and 1s)
    audio_bytes = response.content
    print(f"🎤 TTS created {len(audio_bytes)} bytes of MP3 audio.")
    return audio_bytes
```

Here's the play-by-play:

1. Our `text_to_speech_api` function needs the `client`, the `text_to_spea` and optionally a `voice_choice` (OpenAI offers several voices like "alloy", "echo", "nova", etc.).
2. We call `client.audio.speech.create(...)`, giving it:
   - The `model` f
   - The `voice` v
   - The `input` t

   - The `response_format` – we're asking for an `"mp3"` file, which is a com audio format.
3. OpenAI works its magic and sends back the spoken audio as `response.content`. This `audio_bytes` is the digital sound of the voice.

Awesome! Now we have actual audio bytes – the app is ready to make some noise.

5. Audio Playback – The Speaker

We've got the voice, recorded as a bunch of digital `audio_bytes` (likely in MP3 format from TTS). The very last step? Decoding those bytes and playing the sound through your computer's speakers so you can hear it! This is like hitting the "play' button on your favorite music app.

Here's how we can decode and play those TTS audio bytes using `sounddevice` a: `soundfile`:

```
import sounddevice as sd
import soundfile as sf
import io # For BytesIO

def decode_and_play_tts_bytes(audio_bytes_from_tts):
    if not audio_bytes_from_tts:
        print("🔇 No audio bytes to play.")
        return

    print("🔊 Playing audio...")
    # soundfile needs to read the bytes as if it's a file
    # so we wrap audio_bytes_from_tts in io.BytesIO
    # It decodes the MP3 data and gives us raw sound (sound_data) and
its speed (samplerate)
    sound_data, s                                                     s),
dtype='float32')

    sd.play(sounc
    sd.wait() # V
    print("🎶 Pl
```

Let's tune into what this code does:

1. The `decode_and_play_tts_bytes` function gets the `audio_bytes_from_tts` (the MP3 data we got from OpenAI).
2. To play an MP3, we first need to decode it into raw sound waves that the computer understands. The `soundfile` library (we call it `sf`) is great for this `sf.read(io.BytesIO(audio_bytes_from_tts), ...)` does the trick. use `io.BytesIO` again to make our bytes look like a file to `soundfile`. It gi us back the `sound_data` and the `samplerate` (how fast the sound should be played).
3. Then, our old friend `sounddevice` (called `sd`) steps up. `sd.play(sound_da samplerate)` sends the sound to your speakers.
4. `sd.wait()` is pretty important: it tells our program to pause and patiently w until the sound has completely finished playing before doing anything else.

And *boom*! Your chatbot speaks! The whole cycle – listening, understanding, think and responding with voice – is complete.

Phew! That was a quick tour of all the main parts. Each one is a specialist, and wh they work together, you get a smooth, chatty app!

4. Orchestrating Voice Interactions with PocketFlow

Okay, we've got all these cool parts for our voice chatbot: a listener (VAD), a transcriber (STT), a brain (LLM), a voice (TTS), and a speaker. But how do we get t to work together like a well-rehearsed orchestra? That's where **PocketFlow** steps

Think of PocketFlow ... hat what our component ... "Node" in PocketFlo ... ow smoothly from one to ... a couple of easy ideas: LEGOs® and a Shared Notepad!

1. Nodes (The LEGO® Bricks)

Each main job in our voice chat (like "listen to audio" or "get LLM reply") become `Node`. A `Node` is just a Python class representing one step. Here's a simplified pee what PocketFlow's basic `Node` looks like internally:

```
class Node:
    def __init__(self): self.successors = {}
    def prep(self, shared): pass
    def exec(self, prep_res): pass
    def post(self, shared, prep_res, exec_res): return "default"
    def run(self, shared): p=self.prep(shared); e=self.exec(p); returr
self.post(shared, p, e)
```

So, each Node generally knows how to:

- `prep(shared)`: Get ready! It grabs any info it needs from the common `shar` notepad.
- `exec(prep_res)`: Do the main work! This is where it would call our functio like `record_with_vad` or `call_llm`.
- `post(shared, ...)`: Clean up! It puts its results back on the `shared` note and returns an `action_signal` (like a string, e.g., `"default"` or `"error"`) the Flow, telling it what to do next.

When a Node finishes and returns the `"default"` signal from its `post` method, PocketFlow automatically sends the `shared_notepad` to the next Node in this c

2. Shared Store (The Central Notepad)

This is just a regular des
read from and write t
assembly line – like t
conversation history.

3. Flow & Connecting Nodes (The Conductor & The Pat

A `Flow` object is in charge of the overall process. You tell it which Node to start w Here's a simplified look at PocketFlow's `Flow`:

```
class Flow(Node): # A Flow manages a sequence of Nodes
    def __init__(self, start):
        self.start = start # The first Node in this Flow
    def orch(self, shared, params=None): # orch for internal
orchestration
        curr = self.start
        while curr: action=curr.run(shared);
curr=curr.successors.get(action)
```

The `Flow`'s `orch` (orchestration) method is the core loop. It takes the `shared` not and:

1. Starts with the `self.start` node.
2. Runs the current node (`curr.run(shared)`), which executes that node's `pr > exec -> post` cycle and returns an `action` signal.
3. Looks up that `action` in the current node's `successors` dictionary to find t next node.
4. Continues until no next node is found for a given signal (which ends the flow)

How does the `Flow` know which Node is next for a given `action` signal? You defi these connections! PocketFlow makes it easy to define the default order of your N using the >> operator:

```
# This sets up a
node1 >> node2 >:
```


When a Node finishe

whatever signal >> is configured for), the `Flow` (using the `successors` popu

`>>`) sends the `shared_notepad` to the next Node in this chain.

Tiny Example: Number Game!

Let's make this super clear with a tiny game, now with code: take a number, add 1, then multiply the result by 2.

First, our LEGO® bricks (Nodes):

```
# Assuming our minimal 'Node' class is defined as shown before

class AddOneNode(Node):
    def prep(self, shared_notepad):
        return shared_notepad.get("number", 0) # Get number, default t
0

    def exec(self, number_from_prep):
        result = number_from_prep + 1
        print(f"AddOneNode: {number_from_prep} + 1 = {result}")
        return result

    def post(self, shared_notepad, _, exec_result):
        shared_notepad["number"] = exec_result # Update number in
notepad
        return "default" # Signal to go to next default node

class MultiplyByTwoNode(Node):
    def prep(self, shared_notepad):
        return shared_notepad.get("number", 0)

    def exec(self, number_from_prep):
        result = number_from_prep * 2
        print(f"MultiplyByTwoNode: {number_from_prep} * 2 = {result}")
        return result

    def post(sel
        shared_n
result
        print(f"M
        return "
```

Now, let's set up and run this game with PocketFlow:

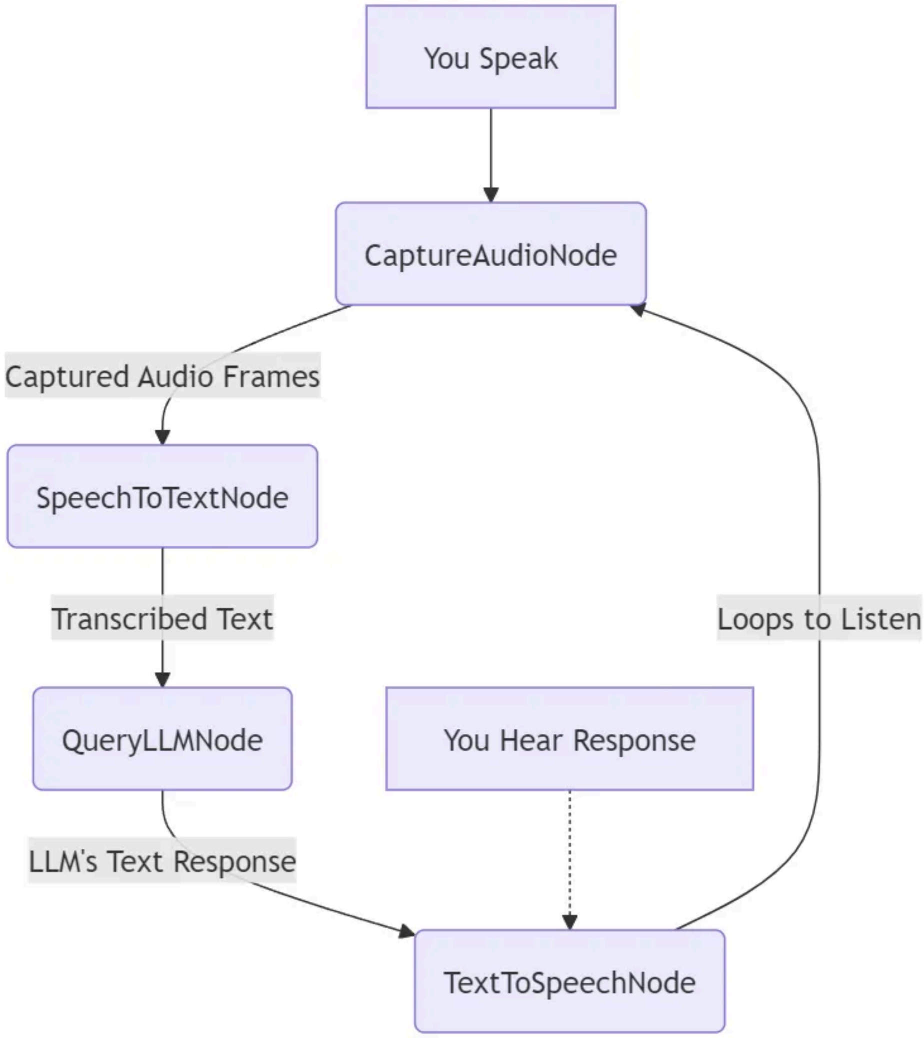
```
# 1. Create instances of our node bricks
add_node = AddOneNode()
multiply_node = MultiplyByTwoNode()

# 2. Connect them! This says "add_node is followed by multiply_node" f
the "default" signal.
add_node >> multiply_node

# 3. Create the Flow, telling it to start with add_node
# (Assuming our minimal 'Flow' class is defined as shown before)
number_game_flow = Flow(start_node=add_node)

# 4. Let's prepare our shared notepad and run the game!
shared_notepad = {"number": 5} # Start with 5
print(f"Starting Number Game with: {shared_notepad}")
number_game_flow.run(shared_notepad)
print(f"Number Game finished. Notepad: {shared_notepad}")

# Expected console output:
# Starting Number Game with: {'number': 5}
# AddOneNode: 5 + 1 = 6
# MultiplyByTwoNode: 6 * 2 = 12
# MultiplyByTwoNode: Final result is 12
# Number Game finished. Notepad: {'number': 6, 'final_result': 12}
```

See? PocketFlow just helps us snap our LEGO bricks together and run the process
The `shared_notepad` carries the data between them.

5. The Voice Conversation Loop in Action: A PocketFlow Walkthrough

Our Voice Cha

Now, let's see how ou esi
to loop for a continu



Each box in the diagram is a custom `Node` we'll build (inheriting from PocketFlow `Node` from Section 4). `CaptureAudioNode`, `SpeechToTextNode`, and `QueryLLMNode` handle their specific tasks. The `TextToSpeechNode` will both convert the LLM's text to audio and play it, then decide whether to loop back. The `shared` dictionary (our shared notepad) will carry information like API keys, capt audio, transcribed text, and conversation history between them.

The Voice Chat's `shared` Dictionary Structure

We learned that Pock ... ıge information. For our ... typically holding the ... s:


```
shared = {
    "user_audio_data": None,       # NumPy array of user's voice
```

```
    "user_audio_sample_rate": None, # int: Sample rate of user's voice
    "chat_history": [],           # list: Full conversation history
    "continue_conversation": True # boolean: Controls conversation loo
}
```

Each Node can access and update the relevant keys in this dictionary. For example `CaptureAudioNode` would populate `"user_audio_data"` and `"user_audio_sample_rate"`, while `QueryLLMNode` would use and update `"chat_history"`. This keeps our Nodes focused and the data flow clear.

Meet the Node Workers: The prep -> exec -> post Cycl

Remember the `Node` structure from Section 4, with its `prep -> exec -> post` cycle? The Nodes we're about to explore are built just like that. They all inherit fro PocketFlow's base `Node` class and use the `shared` dictionary we just discussed to necessary data and to store their results or pass information along.

1. `CaptureAudioNode` – The Attentive Listener

This Node is our chatbot's ears. Its main job is to listen, use Voice Activity Detect (VAD) to know when you're talking, and capture your speech.

```
class CaptureAudioNode(Node):
    def prep(self, shared):
        print("🎙️ CaptureAudioNode: Ready to listen...")
        return None

    def exec(self, _):
        audio_np_array, sample_rate = record_with_vad()
        if audio_np_array is None:
            prir
            retu
        return au

    def post(sel
        audio_np_                                          se
(None, None)
        if audio_np_array is not None:
```

```
            shared["user_audio_data"] = audio_np_array
            shared["user_audio_sample_rate"] = sample_rate
            return "default"
        else:
            print("🎤 CaptureAudioNode: No speech detected. Trying again...")
            return "next_turn"
```

How it works:

- `prep`: Gets ready to listen.
- `exec`: Calls our `record_with_vad()` function (from Section 3), which retur NumPy array of the audio and its sample rate.
- `post`: If voice was captured, it puts the NumPy audio data onto `shared["user_audio_data"]`, its sample rate onto `shared["user_audio_sample_rate"]`, and signals `"default"`. If not, i signals `"next_turn"` to loop back and listen again.

2. `SpeechToTextNode` – The Accurate Transcriber

Got audio? This Node sends it to a Speech-to-Text (STT) service.

```
class SpeechToTextNode(Node):
    def prep(self, shared):
        audio_np_array = shared.get("user_audio_data")
        sample_rate = shared.get("user_audio_sample_rate")
        if audio_np_array is None or sample_rate is None:
            print("🎤 SpeechToTextNode: No audio data from shared.")
            return None
        return audio_np_array, sample_rate

    def exec(sel
        if not au
        audio_np_

        # Conver
        byte_io = io.BytesIO()
        scipy.io.wavfile.write(byte_io, sample_rate, audio_np_arr
```

```
        wav_bytes = byte_io.getvalue()

        transcribed_text = speech_to_text_api(audio_data=wav_bytes)
        return transcribed_text

    def post(self, shared, _, transcribed_text_from_exec):
        if transcribed_text_from_exec:
            history = shared.get("chat_history", [])
            history.append({"role": "user", "content":
transcribed_text_from_exec})
            shared["chat_history"] = history
            print(f"💬 STT adds to history: User said
'{transcribed_text_from_exec}'")
        else:
            print("🎤 SpeechToTextNode: No text transcribed.")

        shared["user_audio_data"] = None
        shared["user_audio_sample_rate"] = None
        return "default"
```

How it works:

- `prep`: Grabs `shared["user_audio_data"]` (the NumPy array) and `shared["user_audio_sample_rate"]`.
- `exec`: Converts the NumPy audio array and its sample rate into WAV bytes (u `io.BytesIO` and `scipy.io.wavfile.write`), then calls `speech_to_text_api()` which returns the transcribed text.
- `post`: If text was transcribed, it updates `shared["chat_history"]` by adc a new entry for the user's query. It then clears the audio data from `shared`.

3. `QueryLLMNode` – The Intelligent Brain

This Node chats witl


```
class QueryLLMNo
    def prep(sel
        chat_history_for_llm = shared.get("chat_history", [])
```

```
        if not chat_history_for_llm or
chat_history_for_llm[-1].get("role") != "user":
            print("🧠 QueryLLMNode: Chat history empty or doesn't end
with user query.")
            return None
        return chat_history_for_llm

    def exec(self, messages_for_llm):
        if not messages_for_llm: return None
        llm_response_text = call_llm(messages=messages_for_llm)
        return llm_response_text

    def post(self, shared, _, llm_response_from_exec):
        if llm_response_from_exec:
            history = shared.get("chat_history", [])
            history.append({"role": "assistant", "content":
llm_response_from_exec})
            shared["chat_history"] = history
            print(f"💡 LLM adds to history: Assistant said
'{llm_response_from_exec}'")
        else:
            print("🧠 QueryLLMNode: No response from LLM.")
        return "default"
```

How it works:

- `prep`: Gets the full `shared["chat_history"]` (which should include the l
user query from `SpeechToTextNode`).
- `exec`: Sends this history to `call_llm()` which returns the LLM's textual response.
- `post`: If the LLM responded, it updates `shared["chat_history"]` by add
new entry for the assistant's reply.

4. TextToSpe

Looks like an article worth saving!

Option Q

ak

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

Turns the LLM's text

Remind me later Hide Forever

```
class TextToSpeechNode(Node):
    def prep(self, shared):
        chat_history = shared.get("chat_history", [])
        if not chat_history or chat_history[-1].get("role") !=
"assistant":
            print("🤖 TextToSpeechNode: No assistant reply in history
for TTS.")
            return None
        text_to_speak = chat_history[-1].get("content")
        return text_to_speak

    def exec(self, text_to_speak):
        if not text_to_speak: return None
        audio_bytes = text_to_speech_api(text_to_speak) # from Sec 3.4
        return audio_bytes

    def post(self, shared, _, audio_bytes_from_exec):
        if audio_bytes_from_exec:
            print(f"🎤 TTS generated audio data (approx.
{len(audio_bytes_from_exec)} bytes). Playing...")
            decode_and_play_tts_bytes(audio_bytes_from_exec)
            print("🎶 Playback finished.")
        else:
            print("🤖 TextToSpeechNode: TTS failed or no input")

        if shared.get("continue_conversation", True):
            return "next_turn"
        else:
            print("Conversation ended by user flag.")
            return "end_conversation"
```

How it works:

- `prep`: Grabs the latest message from `shared["chat_history"]`, expectin to be the LLM as
- `exec`: Calls `tex` e synthesized audi
- `post`: If audio by te (from Section 3.5) to play the audio. It then checks

`shared["continue_conversation"]` and signals `"next_turn"` to loop to listening or `"end_conversation"`.

5. Connecting the Nodes: The Voice Chat `Flow`

Now we snap our LEGO® Node bricks together using PocketFlow to create the ful conversation loop.

```
# 1. Create instances of all Node workers
capture_node    = CaptureAudioNode()
stt_node        = SpeechToTextNode()
llm_node        = QueryLLMNode()
tts_node        = TextToSpeechNode() # This node now also handles
playback

# 2. Connect them to define flow paths based on signals
capture_node.add_successor(stt_node, action="default")
capture_node.add_successor(capture_node, action="next_turn") # If no
audio, loop back

stt_node >> llm_node >> tts_node # Default path for successful
processing

tts_node.add_successor(capture_node, action="next_turn") # After TTS &
playback, loop to listen

# 3. Create the main flow, starting with capture_node
voice_chat_flow = Flow(start_node=capture_node)

# 4. Running the Conversation Loop
shared = {
    "user_audio_data": None,
    "user_audio_sample_rate": None,
    "chat_history": []
    "continue_co
}
voice_chat_flow.
```

And that's how PocketFlow helps us build a sophisticated voice chatbot by connec simple, focused Nodes in a clear, manageable loop! This modular design makes it to understand, test, and even swap out parts later.

6. Conclusion: Your Voice Adventure Awaits!

And there you have it! You've journeyed from a simple "hello" with voice comman seeing how all the pieces click together to make a voice chatbot that really chats. peeked under the hood to see how your voice gets heard (VAD & Audio Capture), turned into words (STT), understood by a smart brain (LLM), and then spoken bac you (TTS & Playback), all in a smooth, looping conversation.

Hopefully, this tutorial has shown you that building cool voice-controlled apps do have to feel like rocket science. With a friendly toolkit like **PocketFlow**, complex t become way more manageable. PocketFlow is like your helpful conductor, letting focus on the fun part – what each piece of your voice app should *do* – instead of ge tangled in the tricky wiring. It's all about breaking big ideas into small, understandable `Nodes` and letting the `Flow` make sure they all play nicely togethe

Now, it's your turn to grab the mic and get creative! We really encourage you to explore the [PocketFlow Voice Chat Cookbook]() we've mentioned. Play around witl maybe even try to break it (that's often the best way to learn!), and then build it ba up. What if you tried a different voice for the TTS, or made the VAD more or less sensitive? What other awesome voice-powered tools can you dream up?

The world of voice interaction is getting bigger and more exciting every day, and tools like PocketFlow, you're all set to jump in and be a part of it. Go on, give your project a voice – we can't wait to see what you build!

Ready to explore m ... *ma*
PocketFlow reposito ... *all*
latest updates. The co ... *ket*
voice-chat directory within the PocketFlow cookbook.

Happy building, and happy chatting with your new AI creations!

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

16 Likes · 1 Restack

← Previous Next

Discussion about this post

Comments Restacks

Write a comment...