Audio-to-text-presciption

Siddharthakhandelwal/Audio-to-text-presciption

english

gemini-2.5-flash

676d0f6

Aug 3, 2025

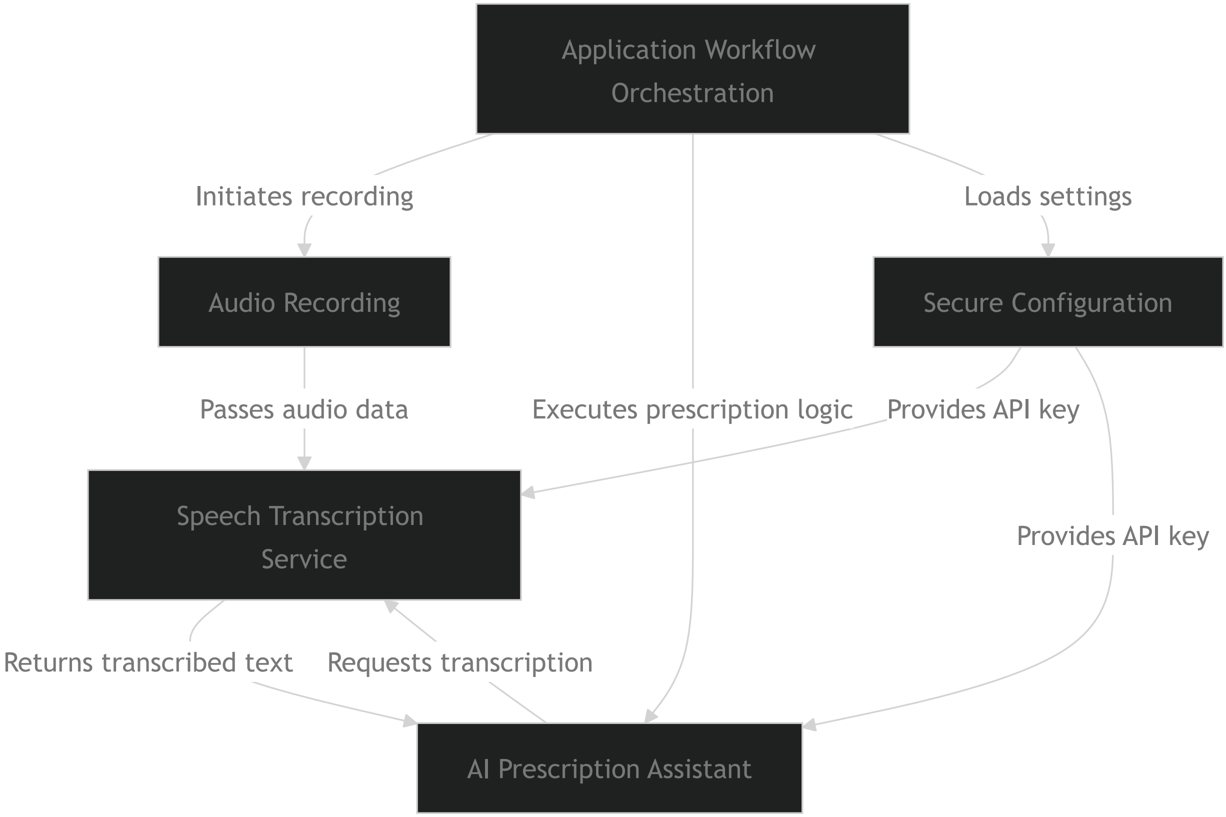
Chapters

Show Raw Markdown

Tutorial: Audio-to-text-presciption

The `Audio-to-text-presciption` project is a Python tool that acts like a **digital medical assistant**. It *records spoken medical notes*, then uses a speech service to *convert them into written text*. Finally, an advanced AI processes this text to *generate a structured and detailed prescription* document, streamlining record-keeping for healthcare professionals.

Visual Overview



Chapters

Generated by AI Codebase Knowledge Builder.

Codebase to Tutorial Open Source ★ 11.1K

Audio-to-text-presciption

Siddharthakhandelwal/Audio-to-text-presciption

english

gemini-2.5-flash

676d0f6

Aug 3, 2025

Chapters

Show Raw Markdown

Chapter 1: Audio Recording

Welcome to the exciting world of turning spoken words into digital text! In this project, we're building a "smart assistant" that can listen to a doctor's words and help create a prescription. But before we can understand what someone says, we first need to *hear* it.

Imagine you're a doctor, and you want to dictate a patient's prescription instead of writing it down. Your computer needs a way to "listen" to your voice. This is exactly what **Audio Recording** does for us.

What is Audio Recording?

Think of "Audio Recording" as a very smart voice recorder built right into our prog job is simple:

1. **Listen**: It uses your computer's microphone to listen for sounds, especially when you start speaking.
2. **Capture**: It captures those sounds.
3. **Save**: It saves the captured sounds as a digital audio file, much like how you might save a picture or a document. For our project, it saves it as a `.wav` file, which is a common format for audio.

This step is crucial because it takes the real-world sound of your voice and turns it into something the computer can understand and process in the next steps of our project.

How We Record Audio

Our project has a special function to handle this "smart voice recorder" task. It waits for you to tell it when to start and stop recording.

Here's a simplified look at how you would use it:

```python
import os
from dotenv import load_dotenv

# We need to load some settings first, but don't worry about this for now!
load_dotenv()

# This is our special function that records audio
def manual_record_audio(filename='user_input.wav', fs=22050):
    print("Press Enter to start recording...")
    input() # Waits for you to press Enter
    print("Recording... Press Enter to stop.")

    # ... more technical recording details are here ...

    input() # Waits for you to press Enter again to stop
    print("Recording saved.")
    return filename

# How we would use it in our main program:
def main():
    # If the audio file doesn't exist, we record it
    if not os.path.exists("user_input.wav"):
        manual_record_audio()
    # ... then we would do other things with the audio ...

```

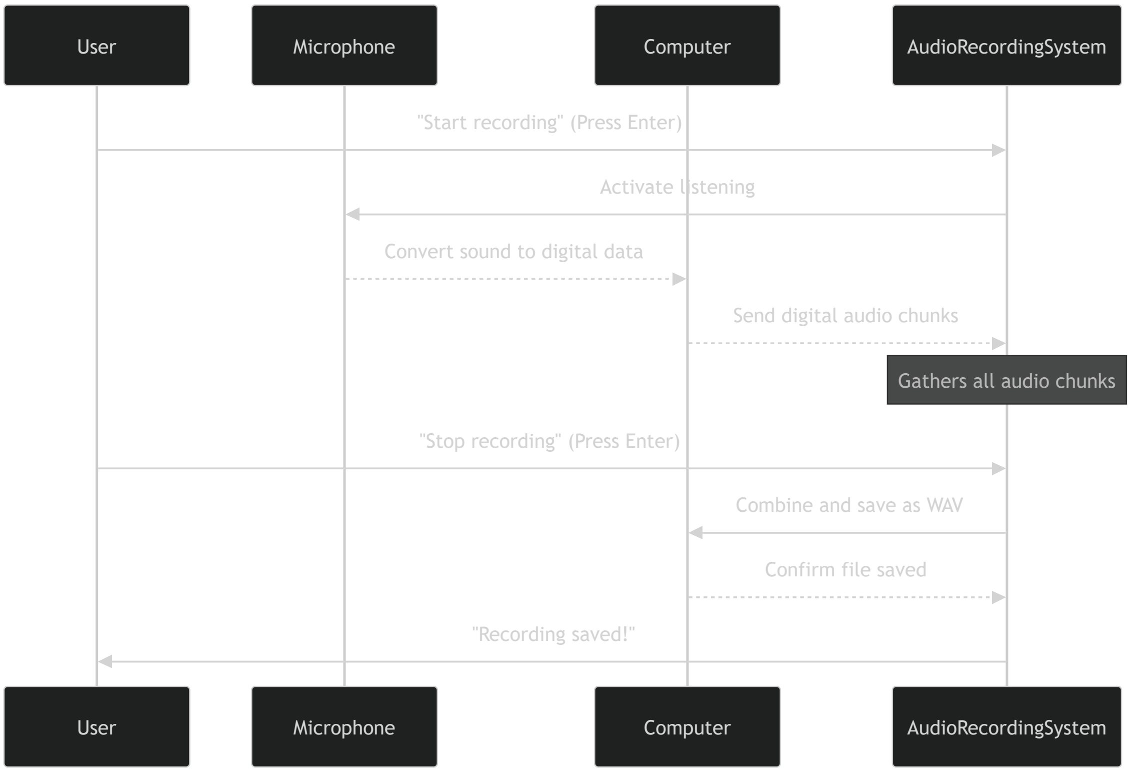
```
if __name__ == "__main__":
    main()

```

When you run this code, it will first tell you to "Press Enter to start recording...". Once you press Enter, it will start listening and capturing your voice. When you're done speaking, you press Enter again, and it saves your recording into a file named `user_input.wav` . This file now holds your spoken words!

Under the Hood: How Recording Works

Let's peek behind the curtain to see what happens when you use our `manual_record_audio` function. It's like a tiny factory working to turn your voice into a digital file.

The Recording Process



1. **You Start**: When you press Enter, you tell our `AudioRecordingSystem` to begin.
2. **Microphone Listens**: The system activates your computer's `Microphone` .
3. **Sound to Digital**: The `Microphone` (and your computer's sound card) convert which is an analog sound wave, into digital information (numbers!).

4. **Chunks of Data**: This digital audio comes in small "chunks" or pieces to the `AudioRecordingSystem`.
5. **Collecting Chunks**: Our system continuously collects all these little pieces of audio data.
6. **You Stop**: When you press Enter again, you tell the `AudioRecordingSystem` to stop collecting.
7. **Combine and Save**: All the collected pieces are then put together, like assembling a puzzle, and saved as a single `.wav` file on your computer.

The Code Behind It

Our `manual_record_audio` function uses a few special tools (libraries) in Python to make this happen:

- **`sounddevice`**: This library helps us talk to the microphone and capture the incoming sound.
- **`numpy`**: This library is great for handling large amounts of numerical data, which is exactly what digital audio is!
- **`scipy.io.wavfile`**: This part of the `scipy` library helps us save the processed audio data into a standard `.wav` file.

Let's look at the key parts of the `manual_record_audio` function (from `prescription.py`):

```
import sounddevice as sd # For talking to your microphone
import numpy as np       # For handling audio data
from scipy.io.wavfile import write # For saving the audio file

# ... (function start and user prompts) ...

def manual_record_audio(filename='user_input.wav', fs=22050):
    # ... user prompts ...

    recording = [] # This list will hold all the small pieces of audio
    def callback(indata, frames, time, status):
        # This function gets called repeatedly as audio comes in
        if status: # If there's any warning/error, print it
            print(status)
        recording.append(indata.copy()) # Add the new audio piece to our list

```

Here, `recording` is like a basket where we collect all the incoming audio pieces. T function is a helper that `sounddevice` uses to pass us new audio data whenever it something.

Next, we use `sounddevice` to actually start listening:

```python
# ... (previous code) ...

# Start the microphone stream and record
with sd.InputStream(samplerate=fs, channels=1, callback=callback):
    # We wait for you to press Enter again to stop recording
    input()
    # Once you press Enter, the 'with' block finishes, and recording stops

```

The `sd.InputStream` sets up the connection to your microphone. `samplerate` (like 22050) tells it how many "snapshots" of sound to take per second, and `channels=1` means it's recording in mono (single channel). The `callback` function we saw earlier gets called every time new audio data is ready. The `input()` line inside the `with` statement pauses the program, waiting for your second Enter press to signal that you're done recording.

Finally, we process and save the audio:

```python
# ... (previous code) ...

audio = np.concatenate(recording, axis=0) # Join all audio pieces together
audio_int16 = np.int16(audio * 32767) # Convert audio data to a suitable format
write(filename, fs, audio_int16) # Save the combined audio to a WAV file
print("Recording saved.")
return filename

```

After you stop recording, `np.concatenate` takes all those small audio pieces we collected in the `recording` list and stitches them into one continuous chunk. Then, `np.int16(audio * 32767)` converts the audio data into a format that's standard for WAV files. Finally, `write(filename, fs, audio_int16)` uses `scipy` to save this processed audio into your `user_input.wav` file.

And just like that, your spoken words are now a digital file, ready for the next step!

Conclusion

In this chapter, we learned about the crucial first step in our `Audio-to-text-pres` project: **Audio Recording**. We understood that it acts like a smart voice recorder, capt sound from your microphone and saving it as a digital WAV file. We also explored the main parts of the code that make this happen, using libraries like `sounddevice`, `numpy`, and `scipy`.

Now that we have our doctor's spoken words saved as an audio file, what's next? The computer can't yet *understand* the words from the audio. We need to convert that audio into actual text. This leads us to our next exciting concept: Speech Transcription Service.

Generated by AI Codebase Knowledge Builder. **References**: [1], [2], [3]

Terms of Service Privacy Policy

Codebase to Tutorial

Open Source ★ 11.1K

Audio-to-text-presciption

Siddharthakhandelwal/Audio-to-text-presciption

english

gemini-2.5-flash

676d0f6

Aug 3, 2025

Chapters

Show Raw Markdown

Chapter 2: Speech Transcription Service

Welcome back! In Chapter 1: Audio Recording, we learned how to capture a doctor's spoken words and save them as an audio file, like `user_input.wav`. Now we have a digital recording of the doctor saying things like "Patient needs Amoxicillin 500mg, three times a day for seven days." But here's the catch: the computer doesn't yet *understand* these words. It just has sound waves!

Imagine you have an audio recording of a meeting, and you need to share the key points in writing. You wouldn't re-listen and type it all out yourself, right? You'd probably use a service that can do it for you! This is exactly what the **Speech Transcription Service** does for our project.

What is Speech Transcription Service?

Think of our **Speech Transcription Service** as a highly skilled, super-fast secretary w specializes in converting spoken words into written text.

Its main job is to:

1. **Receive Audio**: Take the audio file we recorded (like `user_input.wav`).
2. **Send to Expert**: Send this audio file to a powerful, external service that specializes in understanding speech (in our case, `AssemblyAI`).
3. **Wait for Text**: Patiently wait for that expert service to listen to the audio and convert all the spoken words into neat, organized written text.
4. **Return Text**: Give us back the complete written transcript.

This step is absolutely vital because it transforms the doctor's verbal notes into a format (text) that our computer and the AI assistant can actually "read" and understand in the next stages of our project.

Why an "External Service" like AssemblyAI?

You might wonder, why don't we just transcribe the audio ourselves on our computer?

Feature	Doing it Yourself (Local)	Using an External Service (like AssemblyAI)
Complexity	Requires deep knowledge of speech recognition, powerful hardware.	Simple to use; just send audio and get text back.
Accuracy	Hard to achieve high accuracy, especially with different accents or noisy environments.	Highly accurate, trained on vast amounts of data, handles many languages.
Speed	Can be slow without specialized hardware.	Very fast, uses powerful servers to process quickly.
Maintenance	Needs constant updates, model training.	Handled by the service provider.

So, using a service like AssemblyAI is like hiring a top-tier professional: it's efficient, accurate, and saves us a lot of complex work!

How We Use the Speech Transcription Service

In our project, we have a special function called `transcribe_audio`. This function is responsible for talking to the AssemblyAI service and getting the text back.

Here's how you would use it:

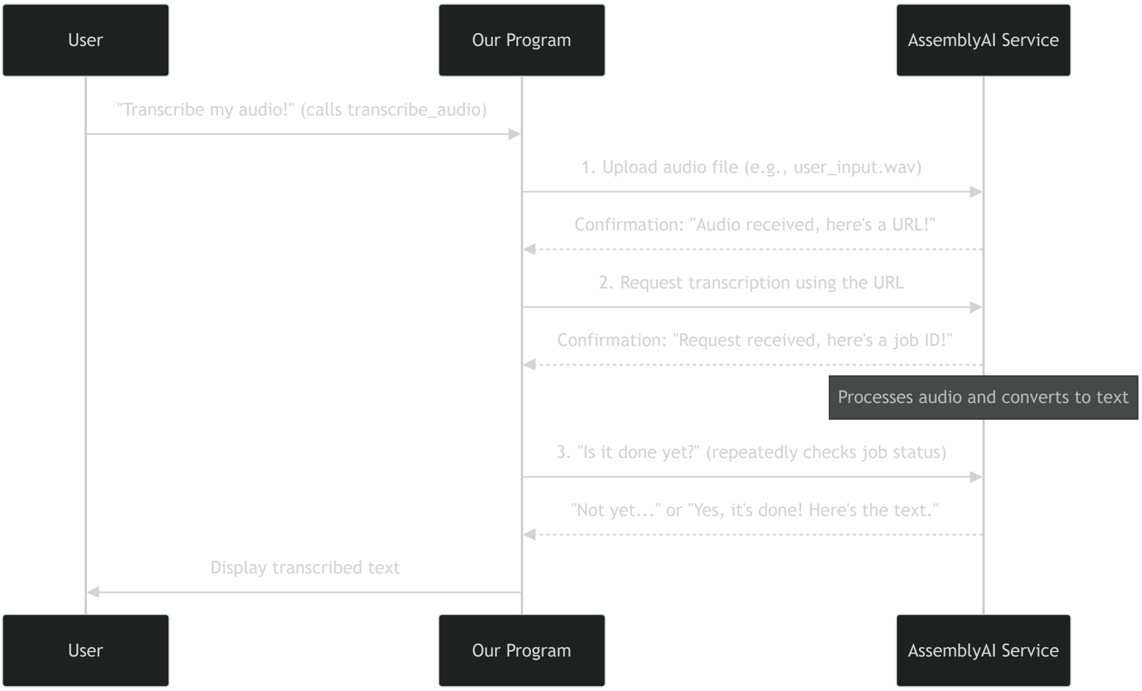
```
import os
# ... other imports ...

# This is our special function that talks to AssemblyAI
def transcribe_audio(filename):
    print("Sending audio for transcription...")
    # ... more technical details inside this function ...

    transcribed_text = "Patient has a cough. Prescribe cough syrup." # Example re
    # In reality, this function waits for AssemblyAI to return the real text

    print("Transcription completed!")
    return transcribed_text

# How we would use it in our main program:
def main():
    audio_file_path = "user_input.wav"
    # Ensure the audio file exists (from Chapter 1)
    if not os.path.exists(audio_file_path):
        # We'd call manual_record_audio() here if it didn't exist
        print("Audio file not found. Please record first.")
        return

    # Call our transcription service
    doctors_notes_text = transcribe_audio(audio_file_path)
    print("Doctor's spoken notes (as text):")
    print(doctors_notes_text)
    # ... then we would send this text to the AI assistant ...

if __name__ == "__main__":
    main()

```

When this code runs, it takes the `user_input.wav` file (which contains the doctor's spoken words). It then calls `transcribe_audio`, which handles all the communication with AssemblyAI. After a short wait, `transcribe_audio` returns the exact words spoken by the doctor, now in text format! For example, if the doctor said "Patient has a cough. Prescribe cough syrup `doctors_notes_text` variable would contain exactly that sentence.

Under the Hood: How Transcription Works

Let's peek behind the curtain to see the steps our `transcribe_audio` function takes to get the job done with AssemblyAI. It's like a three-step dance:

The Transcription Process


User
Our Program
AssemblyAI Service
"Transcribe my audio!" (calls transcribe_audio)
1. Upload audio file (e.g., user_input.wav)
Confirmation: "Audio received, here's a URL!"
2. Request transcription using the URL
Confirmation: "Request received, here's a job ID!"
Processes audio and converts to text
3. "Is it done yet?" (repeatedly checks job status)
"Not yet..." or "Yes, it's done! Here's the text."
Display transcribed text
User
Our Program
AssemblyAI Service

1. **Upload the Audio**: First, `Our Program` sends the `user_input.wav` file to AssemblyAI's cloud servers. AssemblyAI gives us back a special web address (URL) for where our audio is stored.
2. **Request Transcription**: Next, `Our Program` tells AssemblyAI, "Hey, please transcribe the audio file located at this URL." AssemblyAI then gives us a "job ID" — like a ticket number for our transcription request.
3. **Poll for Results**: Since transcription takes a bit of time (especially for longer audio), `Our Program` doesn't just sit and wait. Instead, it periodically asks AssemblyAI, "Is my transcription job (with this ID) finished yet?" This is called "polling."
   - AssemblyAI might say, "Not yet, still working!"
   - `Our Program` waits a few seconds and asks again.
   - Eventually, AssemblyAI replies, "Yes, it's completed! Here is the written text."
4. **Receive and Use**: Once `Our Program` gets the completed text, it can then use it for the next steps in our project. It also cleans up by deleting the `user_input.wav` file.

The Code Behind It

Let's look at the key parts of the `transcribe_audio` function from `prescription.`[illegible]ake this happen.

First, we set up some important details:

```
import requests # This library helps us make web requests
# ... other imports ...

base_url = "https://api.assemblyai.com" # The main address for AssemblyAI's service
headers = {
    "authorization": os.environ.get("ASSEMBLYAI_API_KEY") # Your secret key to use
}

def transcribe_audio(filename):
    # ... function details ...

```

- `requests` is a powerful Python library for communicating with websites and online services (APIs).
- `base_url` is the starting point for all our requests to AssemblyAI.
- `headers` contain important information for AssemblyAI, including your `authorization` key (like a password) that tells AssemblyAI you're allowed to use their service. We get this key from our secure configuration (more on this in Chapter 5: Secure Configuration).

Now, let's see the three steps in action:

Step 1: Upload the Audio File

```
def transcribe_audio(filename):
    with open(filename, "rb") as f: # Open the audio file in "read binary" mode
        upload_response = requests.post( # Send a POST request to upload
            base_url + "/v2/upload", # The specific AssemblyAI address for uploadin
            headers=headers,         # Include our authorization key
            data=f                   # The actual audio file data
        )
    audio_url = upload_response.json()["upload_url"] # Get the URL AssemblyAI gives
    print(f"Uploaded audio URL: {audio_url}")
    # ... more code for transcription request ...

```

Here, we open our `user_input.wav` file and send it to AssemblyAI's `/v2/upload` ad[illegible] successful, AssemblyAI tells us the `audio_url` where our file is now stored on their side.

Step 2: Submit Transcription Request

```
# ... (previous code) ...

data = {
    "audio_url": audio_url,      # Tell AssemblyAI where our audio is
    "language_detection": True   # Ask AssemblyAI to figure out the language au
}

transcript_response = requests.post( # Send a POST request to start transcripti
    base_url + "/v2/transcript",     # The specific AssemblyAI address for tran
    json=data,                       # Our request details in JSON format
    headers=headers                  # Our authorization key again
)
transcript_id = transcript_response.json()["id"] # Get the 'job ID' for this re
polling_endpoint = f"{base_url}/v2/transcript/{transcript_id}" # The address to
# ... more code for polling ...

```

With the `audio_url`, we then send another request to `/v2/transcript`. We tell AssemblyAI where the audio is and also ask it to automatically detect the language (like English, Spanish, etc.). AssemblyAI gives us back a `transcript_id`, which is like our tracking number for this specific transcription job. We also create a `polling_endpoint` — this is the specific web address we'll keep checking for our job's status.

Step 3: Poll Until Transcription is Complete

```
# ... (previous code) ...

while True: # Keep repeating until the job is done
    result = requests.get(polling_endpoint, headers=headers).json() # Check s

    if result['status'] == 'completed': # Is it finished?
        print("\nTranscription completed:")
        print(result['text'])          # Yes! Print and return the text
        os.remove(filename)            # Clean up: delete the loca      o fil
        return result['text']

    elif result['status'] == 'error': # Oh no, did something go
        raise RuntimeError(f"Transcription failed: {result['error']}")

    else: # Not finished yet, wait and try again
        print("Transcription in progress... Waiting 3 seconds.")
        # time.sleep(3) # In the real code, there's a small pause here

```

This `while True` loop is the "polling" part. We keep asking `AssemblyAI` (using `requests.get` on our `polling_endpoint`) for the status of our transcription job.

- If `result['status']` is `'completed'` , we celebrate! We print the `result['text']` (the actual transcribed words) and then remove the `user_input.wav` file from our computer, because we don't need it anymore. Finally, we return the text.
- If `result['status']` is `'error'` , something went wrong, and we stop with an error message.
- Otherwise, the transcription is still `in progress` , so we print a message and typically wait a few seconds before checking again (the original code includes a `time.sleep(3)` here, which pauses the program).

And that's how our `transcribe_audio` function acts as the "middleman" between our recorded audio and the powerful `AssemblyAI` service, bringing back the written words ready for our AI assistant!

Conclusion

In this chapter, we explored the **Speech Transcription Service**, a crucial component that transforms raw audio into usable text. We learned that it leverages an external expert service, AssemblyAI, to accurately convert spoken words into written notes, much like a skilled secretary. We also looked at the `transcribe_audio` function and how it performs the three key steps: uploading audio, requesting transcription, and patiently polling for the final text.

Now that we have the doctor's verbal notes accurately transcribed into text, the next exciting step is to use this text to actually generate a medical prescription! This is where our smart AI Prescription Assistant comes into play.

Generated by AI Codebase Knowledge Builder. **References**: [1], [2] Codebase to Tutorial

Open Source ★ 11.1K

Audio-to-text-presciption

Siddharthakhandelwal/Audio-to-text-presciption

english

gemini-2.5-flash

676d0f6

Aug 3, 2025

Chapters

Show Raw Markdown

Chapter 3: AI Prescription Assistant

Welcome back, future digital health expert! In Chapter 1: Audio Recording, we learned how to capture a doctor's voice. Then, in Chapter 2: Speech Transcription Service, we transformed that spoken audio into written text. Now, we have a digital version of the doctor's spoken notes, like "Patient has a cough. Prescribe cough syrup."

But having raw text isn't enough. We need to turn that text into a properly formatted, clear, and complete medical prescription. This is where the magic happens! We need a "brain" that can read this text, understand it, and then write a professional prescription.

What is the AI Prescription Assistant?

Imagine a super-smart medical secretary who can instantly read a doctor's notes an[illegible]y write out a prescription. That's exactly what our **AI Prescription Assistant** is!

This component is the "brain" or the "intelligence" of our application. Its main job is to:

1. **Read and Understand**: Take the transcribed text (from Chapter 2) and truly understand its meaning. It's not just matching words; it's grasping medical context.
2. **Extract Key Information**: Identify important details like patient symptoms, diagnoses, specific medications, dosages, and follow-up instructions.
3. **Generate Prescription**: Use all this information to create a complete, structured prescription document, just like a doctor would write it.

This step is incredibly powerful because it automates a critical and complex task, turning simple spoken words into a useful medical document.

How We Use the AI Prescription Assistant

Our project uses a powerful AI model called **Gemini** (developed by Google) to act as our AI Prescription Assistant. We interact with Gemini through a special function called `prescription()` in our code.

Here's how it generally works:

```python
# First, we get the transcribed text (from Chapter 2)
# text = "Patient has a cough. Prescribe cough syrup." # This comes from transcribe

def prescription():
    # This line calls the transcription service from Chapter 2
    # It gets the spoken words as text, e.g., "Patient has a cough..."
    text = transcribe_audio("user_input.wav")
    print("Transcribed:", text)

    # ... more AI magic happens here ...

    # Finally, the AI generates and prints the prescription
    # print("\nPrescription:\n", response.text)

# How we would use it in our main program:
def main():
    # ... (code to ensure audio file exists from Chapter 1) ...

    # Call our AI Prescription Assistant to generate the prescription
    prescription()

if __name__ == "__main__":
    main()

```

When you run this, the `prescription()` function first gets the transcribed text. Then, it sends this text to the Gemini AI model along with very specific instructions. Gemini processes everything and then sends back the generated prescription, which our program displays for you.

For example, if the doctor's transcribed text was:

```
"Patient complains of severe headache for two days, no fever. Advise Paracetamol
```

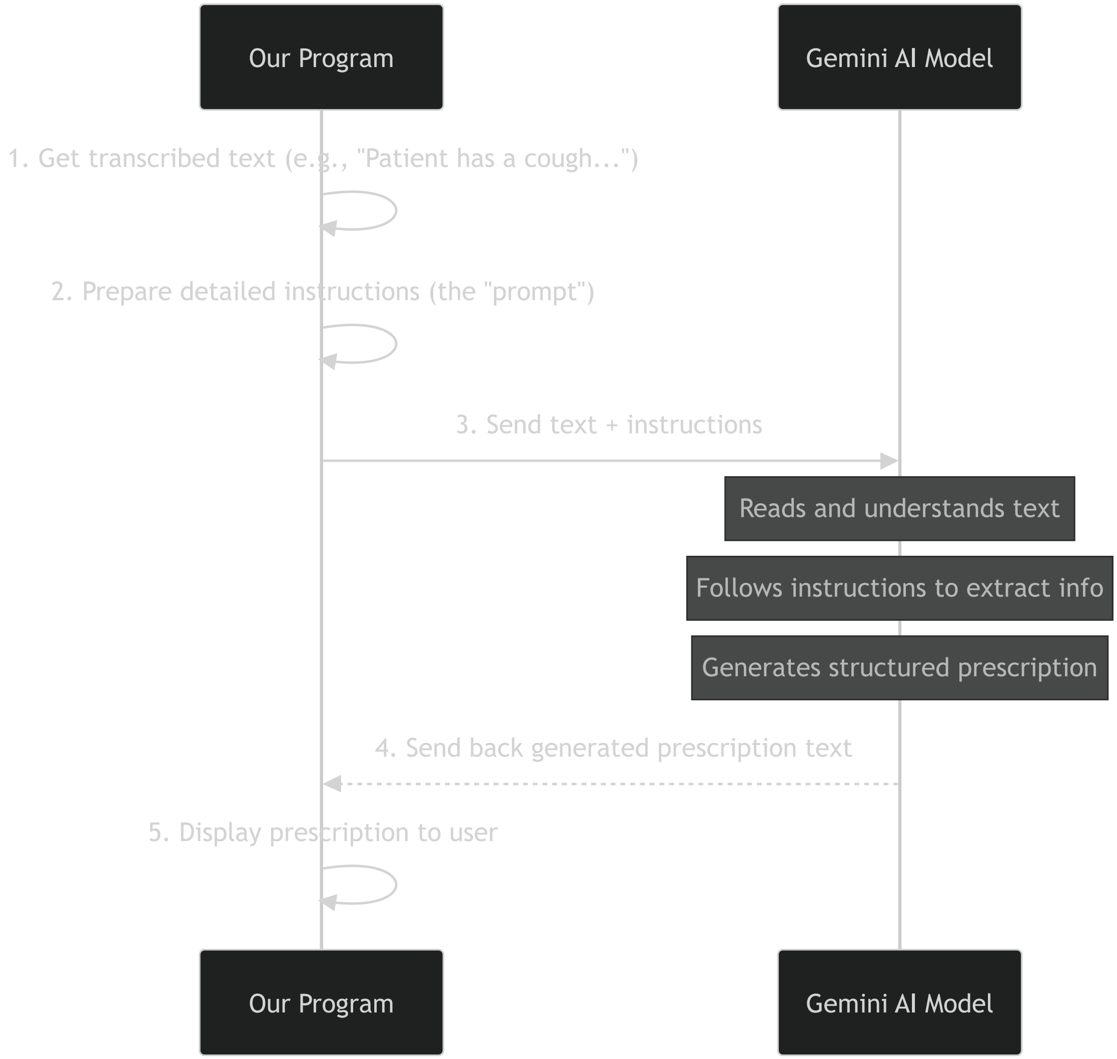
The AI Prescription Assistant would process this and might generate something like:

```
 Patient Complaints: Severe headache for two days.
Clinical Observations: No fever.
Diagnosis: Headache (symptomatic)
Medications:
- Paracetamol 500mg: 1 tablet, twice daily for 3 days.
Follow-up Instructions: Review in 3 days if symptoms do not improve.
```

Under the Hood: How the AI Prescription Assistant Works

Let's peek behind the curtain to see how our `prescription()` function uses the Gemini AI model to generate a prescription. It's like having a very detailed conversation with a super-intelligent assistant.

The AI Prescription Process


Our Program
Gemini AI Model
1. Get transcribed text (e.g., "Patient has a cough...")
2. Prepare detailed instructions (the "prompt")
3. Send text + instructions
Reads and understands text
Follows instructions to extract info
Generates structured prescription
4. Send back generated prescription text
5. Display prescription to user
Our Program
Gemini AI Model

1. **Get Transcribed Text**: Our `prescription()` function first obtains the transcribed text from the Speech Transcription Service.

2. **Prepare Instructions (The "Prompt")**: This is a crucial step! We don't just send the text to the AI; we send it along with a very detailed set of instructions. This is called a "prompt." The prompt tells the AI exactly what role to play (a doctor), what information to look for, and how to format the output (a complete prescription).

3. **Send to Gemini**: Our program then sends both the transcribed text and the detailed prompt to the Gemini AI Model.

4. **Gemini Processes**: Gemini's powerful brain reads the prompt, understands its role, then analyzes the patient's conversation, extracts the required medical information, and finally generates the prescription following all the formatting rules we gave it.

5. **Get Prescription Back**: Gemini sends the generated prescription text back to o

6. **Display to User**: Our program then prints this final prescription for you to see.

The Code Behind It

Let's look at the key parts of the `prescription` function in `prescription.py` that bring this to life.

First, we need to set up our connection to the Gemini AI model:

```python
import os
from google import generativeai as genai # Our AI model library

# ... (other imports and functions) ...

def prescription():
    # ... (code to get transcribed text) ...

    # Configure Gemini: tells our program how to talk to Gemini
    # It uses a secret key from your settings (more in Chapter 5!)
    genai.configure(api_key=os.environ.get("GEMINI_API_KEY"))

    # Choose the specific Gemini model we want to use
    # "gemini-1.5-flash" is a good, fast model for this task
    model = genai.GenerativeModel(model_name="gemini-1.5-flash")

    # ... (more code for the prompt and generating response) ...

```

- `genai.configure()` is like plugging in our AI tool. It needs an `api_key`, which is a special secret code that identifies us to Google's AI service (we'll talk about this in Chapter 5: Secure Configuration).
- `genai.GenerativeModel()` selects which specific Gemini model we want to use. `gemini-1.5-flash` is a good choice because it's fast and smart enough for this kind of task.

Next, we create the "prompt" — our detailed instructions for the AI:

```python
def prescription():
    # ... (previous code) ...

    # This is the "prompt" - detailed instructions for the AI
    prompt = f'''
    You are a doctor. The following is a recorded conversation...
    Based on this conversation, write a clear, detailed, and medically accurate pre

    Ensure the prescription includes:
    - Patient complaints/symptoms
    - Diagnosis (if implied or stated)
    - Medications (with dosage and duration)
    # ... and more instructions ...

    Conversation:
    {text}
    '''
    # The `{text}` part inserts the actual transcribed conversation here!

    # ... (code to generate response) ...

```

This `prompt` is a multi-line string (text). Notice how we tell the AI to:

- `You are a doctor.` This sets the "role" for the AI.
- `write a clear, detailed, and medically accurate prescription` . This is the main task.
- `Ensure the prescription includes: ...` . These are the specific items the AI must extract and include (symptoms, diagnosis, medications, etc.).
- `Conversation: {text}` . This is where the actual transcribed notes from the doctor are inserted into the prompt, so the AI knows what to analyze.

Finally, we send the prompt to the AI and get its answer:

```
def prescription():
    # ... (previous code including prompt) ...

    # Send the prompt to the Gemini model and get its response
    response = model.generate_content(prompt)

    # Print the AI's generated prescription
    print("\nPrescription:\n", response.text)

    # Return the generated text for further use if needed
    return response.text

```

- `model.generate_content(prompt)` is the command that sends our instructions (the `prompt`) to the Gemini AI model. Gemini then processes it and sends back its "answer" (the generated prescription) which is stored in the `response` variable.
- `response.text` gives us the actual written prescription that Gemini has generated. We then `print` it to your screen.

And just like that, our intelligent AI Prescription Assistant turns spoken words, via transcription, into a professional medical document!

Conclusion

In this chapter, we discovered the **AI Prescription Assistant**, the "brain" of our application. We learned how it uses a powerful AI model like Gemini to read transcribed text, understand medical details, and generate a complete, structured prescription document. We explored how to connect to the AI model, how to give it very specific instructions using a "prompt," and how to get its generated output.

Now that we understand each individual piece of our project – from recording audio to transcribing it, and then using AI to generate a prescription – the next exciting step is to see how all these parts fit together and work in harmony. This leads us to our final core concept: Application Workflow Orchestration.

Generated by AI Codebase Knowledge Builder. **References**: [1], [2], [3], [4]

Audio-to-text-presciption

Siddharthakhandelwal/Audio-to-text-presciption

english

gemini-2.5-flash

676d0f6

Aug 3, 2025

Chapters

Show Raw Markdown

Chapter 4: Application Workflow Orchestration

Welcome back, future digital health expert! In our previous chapters, we've built some amazing individual tools:

- In Chapter 1: Audio Recording, we learned how to capture a doctor's spoken words as an audio file.
- In Chapter 2: Speech Transcription Service, we transformed that audio into written text using a smart external service.
- In Chapter 3: AI Prescription Assistant, we used powerful AI to turn that text in d medical prescription.

Now, imagine you have a fantastic team, but they all work in separate rooms without talking to each other. One person records, another transcribes, another writes the prescription. How do you make sure the audio goes to the transcriber, and the text goes to the prescription writer, and it all happens in the right order?

This is where **Application Workflow Orchestration** comes in!

What is Application Workflow Orchestration?

Think of **Application Workflow Orchestration** as the **project manager** or the **conductor of an orchestra** for our application.

Its main job is to:

1. **Define the Sequence**: Decide which step needs to happen first, second, and so on. It lays out the entire "plan."
2. **Manage the Flow**: Make sure that when one step finishes, its output is correctly passed as input to the next step. It's like ensuring the audio file from recording goes directly to the transcription service.
3. **Oversee the Entire Process**: From the moment you start the program until the final prescription is displayed, the orchestrator is in charge, ensuring everything runs smoothly and logically.

Without orchestration, our smart tools would just sit there, not knowing what to do or when to do it. The orchestrator brings them all together into a useful, working application!

How We Orchestrate Our Project

In our `Audio-to-text-presciption` project, the `main()` function acts as our workflow orchestrator. It’s the central point that decides the logical flow of operations.

Here's the problem it solves: We want to generate a prescription from a doctor's spoken words.

The orchestrator (`main()` function) follows a simple, logical plan:

1. **Check for Audio**: Does an `user_input.wav` audio file already exist?
2. **Record if Needed**: If not, it first asks the user to record new audio using the Audio Recording function.
3. **Generate Prescription**: Once we *have* the audio (either pre-existing or newly recorded), it then calls the AI Prescription Assistant function to process it, which internally uses t Transcription Service.

Let's look at the core of the `main()` function that manages this:

```python
import os
# ... (other imports) ...

# This is our conductor, the main orchestrator!
def main():
    # Step 1 & 2: Check and record audio if needed
    if not os.path.exists("user_input.wav"):
        manual_record_audio() # Function from Chapter 1

    # Step 3: Now, generate the prescription!
    prescription() # Function from Chapter 3 (which uses Chapter 2)

if __name__ == "__main__":
    main()

```

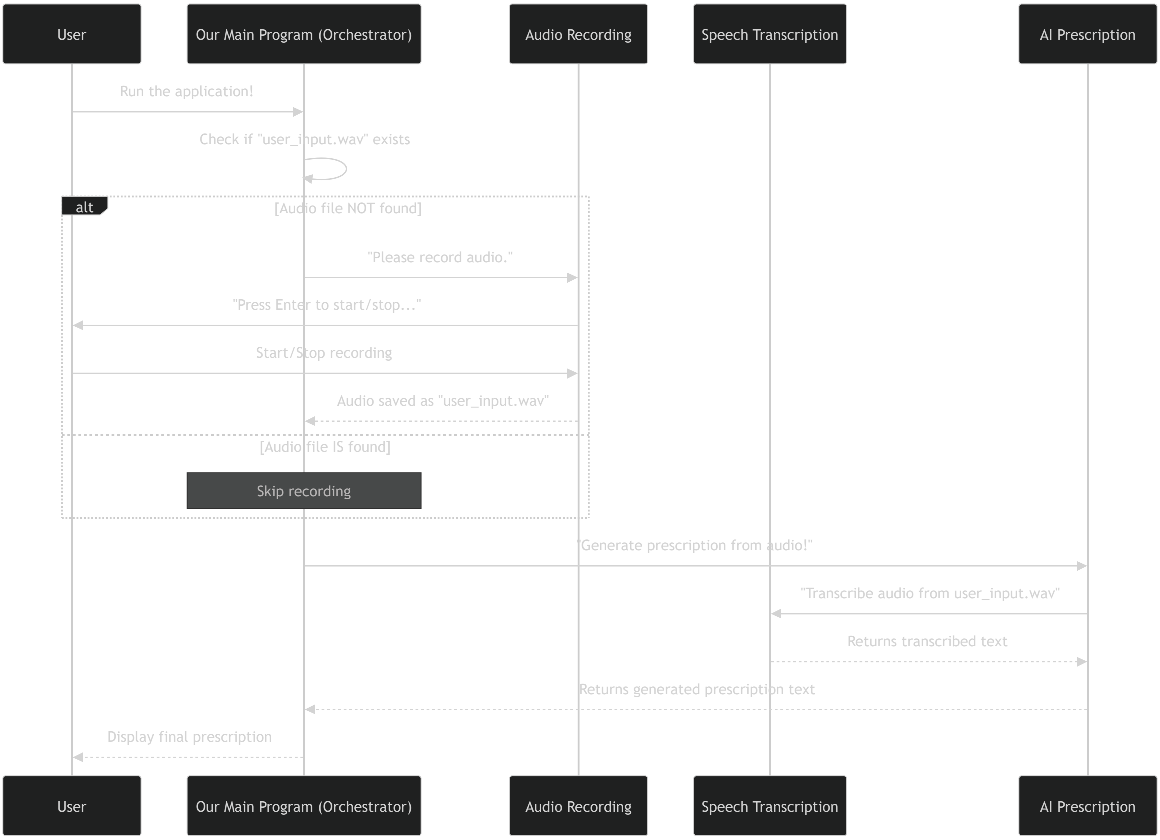
When you run `prescription.py` , the `main()` function is the first thing that gets called (because of `if __name__ == "__main__": main()`). It checks for the `user_input.wav` file. If it's missing, it tells `manual_record_audio()` to capture your voice. Once that's done (or if the file was already there), it confidently calls `prescription()` to take over and handle the rest (transcription and AI generation).

This simple `main()` function ensures that all the individual pieces of our project work together seamlessly, step by step!

Under the Hood: How the Orchestration Works

Let's trace what happens when you run our `prescription.py` program, focusing on the `main` function as the orchestrator.

The Orchestration Process


User
Our Main Program (Orchestrator)
Audio Recording
Speech Transcription
AI Prescription
Run the application!
Check if "user_input.wav" exists
alt
[Audio file NOT found]
"Please record audio."
"Press Enter to start/stop..."
Start/Stop recording
Audio saved as "user_input.wav"
[Audio file IS found]
Skip recording
"Generate prescription from audio!"
"Transcribe audio from user_input.wav"
Returns transcribed text
Returns generated prescription text
Display final prescription
User
Our Main Program (Orchestrator)
Audio Recording
Speech Transcription
AI Prescription

1. **User Starts**: You (the `User`) initiate `Our Main Program` .
2. **Orchestrator Checks**: `Our Main Program` (the orchestrator) first checks its files to see if the `user_input.wav` audio file is already there.
3. **Record if Necessary**:
   - **If the file is missing**: The orchestrator tells the `Audio Recording` part to start recording. `Audio Recording` interacts with you to capture your voice. Once done, it saves the file and tells the orchestrator it's ready.
   - **If the file exists**: The orchestrator skips the recording step entirely, knowing it already has the audio.
4. **Initiate Prescription Generation**: Once `Our Main Program` confirms it has an audio file, it then tells the `AI Prescription` part, "Okay, now generate the prescription from this audio!"
5. **Transcription (Internal Step)**: The `AI Prescription` doesn't work directly with audio. So, it internally calls the `Speech Transcription` service to first convert the audio into text.
6. **AI Processing (Internal Step)**: `Speech Transcription` sends the text back to `AI Prescription` . `AI Prescription` then uses its intelligence to create the final pre document.
7. **Final Output**: `AI Prescription` sends the completed prescription text back to `Program` .

8. **Display to User**: Finally, `Our Main Program` displays the structured prescription on your screen.

This entire flow ensures that each operation depends correctly on the one before it, leading to a smooth and successful generation of your prescription.

The Code Behind It (`prescription.py`)

Let's look at the exact lines in our `prescription.py` file that implement this orchestration logic:

```
# prescription.py

# ... (imports and other functions like transcribe_audio, manual_record_audio, pres

def main():
    # Load environment variables (like API keys), always a good first step!
    load_dotenv()

    # This is the orchestration logic!
    # Check if the audio file exists
    if not os.path.exists("user_input.wav"):
        # If not, call the manual_record_audio function (from Chapter 1)
        manual_record_audio()

    # Once we are sure user_input.wav exists (either pre-existing or newly recorded
    # call the prescription function (from Chapter 3)
    prescription()

# This makes sure main() runs when the script is executed directly
if __name__ == "__main__":
    main()

```

- `load_dotenv()` : This line, typically at the very start of `main()` , loads any secret keys or settings your program needs from a `.env` file. This is important for configuration, which we'll discuss in Chapter 5: Secure Configuration.
- `if not os.path.exists("user_input.wav"):` : This is the decision point. `os.path.exists()` checks if a file or folder exists at the given path. If `user_inpu` *t* found (`not`), then the code inside the `if` block runs.
- `manual_record_audio()` : This function, which we explored in detail in Chapte Recording, is called *only if* new audio needs to be recorded.

- `prescription()` : This function, which is the heart of our AI Prescription Assistant and internally uses the Speech Transcription Service, is called *after* we are sure an audio file is ready. The orchestrator doesn't care *how* the audio file got there (whether it was pre-existing or newly recorded), just that it *is* there before moving to the next step.
- `if __name__ == "__main__": main()` : This is a standard Python line that simply means: "If this script is being run directly (not imported as a module into another script), then please execute the `main()` function."

The `main()` function, therefore, acts as the central control tower, directing the flow of data and operations through our entire `Audio-to-text-presciption` application.

Conclusion

In this chapter, we understood the critical role of **Application Workflow Orchestration**. We learned that it acts as the project manager, defining the sequence of operations and ensuring that each part of our `Audio-to-text-presciption` project works together smoothly, from recording audio to generating the final prescription. We saw how the `main()` function in our `prescription.py` file orchestrates this entire process.

Now that we have a clear understanding of how our application works end-to-end, there's one more crucial aspect to cover: how we manage important, sensitive information like API keys. This brings us to our next chapter: Secure Configuration.

Generated by AI Codebase Knowledge Builder. **References**: [1], [2], [3]

Audio-to-text-presciption

Siddharthakhandelwal/Audio-to-text-presciption

english

gemini-2.5-flash

676d0f6

Aug 3, 2025

Chapters

Show Raw Markdown

Chapter 5: Secure Configuration

Welcome to the final core concept of our `Audio-to-text-presciption` project! In our previous chapters, we built a fantastic system: from Chapter 1: Audio Recording capturing your voice, to Chapter 2: Speech Transcription Service converting it to text, and finally to Chapter 3: AI Prescription Assistant turning that text into a prescription, all orchestrated by Chapter 4: Application Workflow Orchestration.

You might have noticed something special in the code for transcribing audio and using the AI assistant: we used special "keys" or "passwords" to access services like AssemblyAI and Gemini. For example, in the `transcribe_audio` function, there was a line like:

```
headers = {
    "authorization": "eb4d1dd9aa884e91b70bc8bf9f583f1c"  # This was a placeholder!
}

```

And for Gemini in the `prescription` function:

```
genai.configure(api_key=os.environ.get("GEMINI_API_KEY"))

```

That "eb4d1dd9aa884e91b70bc8bf9f583f1c" was just a placeholder. In a real application, these keys are **secret**! They are like your house keys or your bank PIN. If someone gets hold of them, they could potentially use your account, or worse, use up your service credits!

So, how do we use these secret keys without writing them directly into our code where everyone can see them? This is where **Secure Configuration** comes in!

What is Secure Configuration?

Imagine you have a very important, secret diary. You wouldn't write your secrets on the first page of a book you share with everyone, right? Instead, you'd keep it hidden in a special, locked vault.

Secure Configuration is exactly like that secure vault for our program's sensitive information. Instead of embedding secret keys, passwords, or special settings directly into the main code (which is called "hardcoding"), this approach loads them from a separate, protected file.

In our project, this "protected file" is a `.env` file. The `.env` file is a plain text file that holds these secret settings.

Its main jobs are:

1. **Keep Secrets Safe**: It prevents sensitive information like API keys from being directly visible in your main program files. This is important if you share your code online (e.g., on GitHub), as you wouldn't want your private keys exposed.
2. **Make it Flexible**: It allows you to easily change settings for different environments (e.g., a "testing" environment versus a "live" environment) without touching the main program's code.
3. **Improve Security**: By not hardcoding secrets, you reduce the risk of accidental exposure.

This step is absolutely vital for building professional and secure applications.

How We Use Secure Configuration

In our project, we use a Python library called `python-dotenv` to help us with secure configuration. This library makes it super easy to load information from our `.env` file.

Here's how we set it up and use it:

Step 1: Create a `.env` file

In the very same folder as your `prescription.py` file, you create a new file named `.env`. It's just a simple text file. Inside it, you'd put your actual secret keys like this:

```
 # .env file content
GEMINI_API_KEY="YOUR_ACTUAL_GEMINI_API_KEY_HERE"
ASSEMBLYAI_API_KEY="YOUR_ACTUAL_ASSEMBLYAI_API_KEY_HERE"
```

- `GEMINI_API_KEY` and `ASSEMBLYAI_API_KEY` are names we choose for our secret keys.
- `"YOUR_ACTUAL_..."` is where you would paste the real keys you get from Google's AI Studio (for Gemini) and AssemblyAI's website. These keys are unique to you!

Step 2: Load the variables in your Python code

At the very beginning of your `main()` function in `prescription.py` (or sometimes at the top of the file), you use `load_dotenv()`:

```python
# In prescription.py

from dotenv import load_dotenv # Import the function

def main():
    load_dotenv() # This magical line loads settings from .env!
    # ... rest of your main function ...

# ... other functions ...

```

The `load_dotenv()` function tells Python to look for the `.env` file in your project folder. When it finds it, it reads all the `NAME="VALUE"` pairs and adds them to something called "environment variables." Think of environment variables as a special storage area that your computer programs can easily access.

Step 3: Access the variables in your code

Once `load_dotenv()` has done its job, you can access these values anywhere in your Python script using `os.environ.get("VARIABLE_NAME")` .

For example, to get your AssemblyAI key:

```
import os # Need to import 'os' to access environment variables

# ... other code ...

def transcribe_audio(filename):
    # ... code ...
    headers = {
        # Get the API key from environment variables
        "authorization": os.environ.get("ASSEMBLYAI_API_KEY")
    }
    # ... rest of transcribe_audio ...

```

And for your Gemini key:

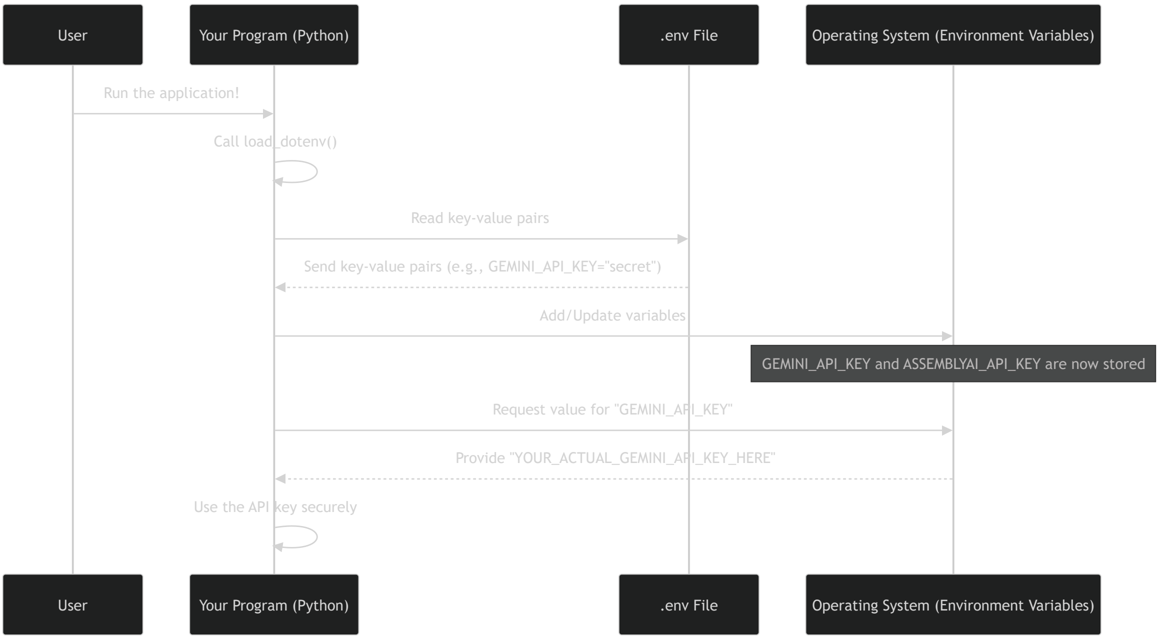
```
import os
from google import generativeai as genai

# ... other code ...

def prescription():
    # ... code ...
    genai.configure(
        api_key=os.environ.get("GEMINI_API_KEY") # Get the API key from environment
    )
    # ... rest of prescription ...

```

Notice how we use `os.environ.get("VARIABLE_NAME")` instead of writing the key directly. This keeps the actual secret value out of our visible code!

Under the Hood: How Secure Configuration Works

Let's peek behind the curtain to see how `load_dotenv()` and `os.environ.get()` together like a secret agent team.

The Secure Configuration Process


User
Your Program (Python)
.env File
Operating System (Environment Variables)
Run the application!
Call load_dotenv()
Read key-value pairs
Send key-value pairs (e.g., GEMINI_API_KEY="secret")
Add/Update variables
GEMINI_API_KEY and ASSEMBLYAI_API_KEY are now stored
Request value for "GEMINI_API_KEY"
Provide "YOUR_ACTUAL_GEMINI_API_KEY_HERE"
Use the API key securely
User
Your Program (Python)
.env File
Operating System (Environment Variables)

1. **You Run the Program**: When you start your Python script, the `main()` function begins.
2. **`load_dotenv()` is Called**: The very first important line in `main()` is `load_dotenv()`.
3. **Read `.env`**: The `python-dotenv` library (behind `load_dotenv()`) looks for and reads your `.env` file.
4. **Update Environment Variables**: It then takes all the `NAME=VALUE` pairs it finds in `.env` and adds them to your computer's "environment variables." This is a special, hidden place where programs can store small bits of information.
5. **Program Asks for Secrets**: Later, when your `transcribe_audio` or `prescription` functions need an API key, they don't look in the `.env` file directly. Instead, they ask the `Operating System` for the value of `ASSEMBLYAI_API_KEY` or `GEMINI_API_KEY`.
6. **Operating System Provides Secret**: The `Operating System` looks up the requested variable in its environment variables (where `load_dotenv()` put them) and provides the value back to your Python program.
7. **Program Uses Secret**: Your program then uses this key to talk to the external services, without ever having the secret key directly written in the code.

The Code Behind It

Let's look at the actual lines in `prescription.py` and `.env` that implement this.

The `.env` file:

```
 # File: .env
GEMINI_API_KEY="AIzaSyBkI50xIaE0zhL-S-61RexARSkhIHxPBRU" # This is an example, u
ASSEMBLYAI_API_KEY="YOUR_ASSEMBLYAI_API_KEY_HERE" # You need to replace this too
```

The content of your `.env` file lists your secret keys. Notice that these values are enclosed in double quotes. This is generally a good practice, especially if your keys might contain special characters.

In `prescription.py`:

First, in `main()`, we load the variables:

```python
# File: prescription.py

import os
from dotenv import load_dotenv # New import!

# ... other functions ...

def main():
    # This line loads everything from your .env file
    load_dotenv()

    if not os.path.exists("user_input.wav"):
        manual_record_audio()
    prescription()

if __name__ == "__main__":
    main()

```

The `load_dotenv()` call here is crucial. It's the first thing `main()` does, ensuring all your environment variables are available before any other function tries to use them.

Next, how `transcribe_audio` gets its key:

```python
# File: prescription.py

import requests
import os # Don't forget to import 'os' if you haven't already!
```

```python

# ... other code ...

base_url = "https://api.assemblyai.com"
headers = {
    # Instead of a hardcoded string, we get it from environment variables
    "authorization": os.environ.get("ASSEMBLYAI_API_KEY")
}

def transcribe_audio(filename):
    # ... (rest of the function using the headers dictionary) ...
    pass # Placeholder for actual code

```

Here, `os.environ.get("ASSEMBLYAI_API_KEY")` safely retrieves the value of the `ASSEMBLYAI_API_KEY` that `load_dotenv()` placed into the environment. If the key isn't found, `get()` returns `None`, preventing errors.

Finally, how `prescription` gets its key:

```python
# File: prescription.py

import os
from google import generativeai as genai

# ... other code ...

def prescription():
    # ... (code to get transcribed text) ...

    # Configure Gemini using the API key from environment variables
    genai.configure(api_key=os.environ.get("GEMINI_API_KEY"))

    # ... (rest of the function) ...
    pass # Placeholder for actual code

```

Similar to AssemblyAI, `os.environ.get("GEMINI_API_KEY")` fetches the Gemini securely.

By using this pattern, your `prescription.py` file remains clean and doesn't expose any sensitive credentials, making your project much safer and more adaptable!

Conclusion

In this chapter, we learned about the critical concept of **Secure Configuration**. We understood that it's like a secure vault for sensitive information, preventing secrets like API keys from being hardcoded directly into our main program files. We saw how to use a `.env` file and the `python-dotenv` library to load these secrets, and how to access them securely in our code using `os.environ.get()`.

This secure approach is a fundamental practice in professional software development, ensuring your application is safe and flexible.

Generated by AI Codebase Knowledge Builder. **References**: [1], [2], [3], [4]