# Technical Review of Telegram Transcriber Bot

## System Flow: Telegram Audio to WhatsApp Summary

1. **Receiving the Audio (Telegram)** – A user sends an audio message (voice note or audio file) to the Telegram bot. The bot (built with the Aiogram framework) detects the incoming voice or audio message via a handler. It first **downloads the audio** to a local downloads/ directory for processing (ensuring the file size is within the allowed limit)[[1]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/README.md#L6-L10)[[2]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/README.md#L16-L22).
2. **Transcription with Whisper** – The bot uses OpenAI’s Whisper model to **transcribe the audio into text**. The Whisper model (e.g. the "base" model by default) is loaded at startup, so transcription is done locally by the Python service. If the audio is long, the resulting transcript may be split into smaller chunks to comply with Telegram message length limits[[3]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/README.md#L5-L10)[[4]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/README.md#L46-L50). The bot can reply in Telegram with the full transcribed text (possibly split across multiple messages) and may also send the transcript as a text file (e.g. \*.txt in the chat) for convenience[[3]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/README.md#L5-L10).
3. **Summarization of Transcript** – Once the speech is transcribed to raw text, the application leverages an **AI summarizer** (OpenAI GPT model) to condense the transcription. The transcribed text is sent to the OpenAI API (using the provided OpenAI API key in the config), prompting it to generate a concise summary. This step produces a shorter, digestible summary of the audio content.
4. **Forwarding to WhatsApp** – Finally, the bot sends the summary to a designated WhatsApp recipient. The project integrates with the WhatsApp API (e.g. via Twilio’s WhatsApp messaging service) to deliver the summary. Using credentials (such as Twilio Account SID, Auth Token, and WhatsApp phone numbers) from the environment config, the code calls the API to **send a WhatsApp message containing the summary**. Once this message is successfully sent to WhatsApp, the end-to-end flow is complete – the user’s voice message has been transcribed and summarized on Telegram, and the summary is delivered on WhatsApp.

*(Throughout this flow, the system also logs events and errors to a logs/ directory, and enforces any admin or chat restrictions configured in the environment variables. For example, certain commands or functionalities might be limited to specific admin user IDs.)*

## Project Structure and File Breakdown

*(All source code resides in the repository mevaser/telegram\_transcriber. The compare\_transcripts/ directory is present in the repo but is* *excluded* *from this review as per request.)*

The project is organized into a Dockerized Python application with a clear separation of the bot logic. Below is a breakdown of each major file (and module) and its purpose, along with the functions defined in them:

* **Dockerfile** – Defines the container image for the bot. It sets up a Python 3 environment with system dependencies like ffmpeg (needed for audio processing)[[5]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Dockerfile#L4-L12). It installs the Python requirements and copies the application source into the image[[6]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Dockerfile#L16-L24). The Dockerfile creates the downloads/ and logs/ directories inside the container and specifies the default command to run the bot (e.g. CMD ["python", "-m", "telegram\_transcriber\_src.main"] to launch the app)[[7]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Dockerfile#L20-L28)[[8]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Dockerfile#L34-L37). *(No custom functions in this file; it’s a setup script for Docker.)*
* **docker-compose.yml** – Provides orchestration for running the bot container. It likely defines a service for the bot using the above Dockerfile, mounts the downloads/ and logs/ volumes for persistence, and loads the .env file for configuration. Using docker-compose up will build and start the bot in a container, automatically applying environment settings (like tokens and API keys). *(No functions; this is a configuration file.)*
* **Makefile** – Contains convenient make targets for common tasks in development and deployment. For example, it probably includes targets like build (to build the Docker image), up (to start the containers in the background), down (to stop containers), logs (to follow the logs), and maybe fmt (to run code formatters)[[9]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Makefile#L9-L17)[[10]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Makefile#L27-L32). These shortcuts help developers quickly build and run the bot or format the codebase. *(No functions in the Makefile; it defines shell commands for automation.)*
* **requirements.txt** – Lists the Python dependencies. This likely includes libraries such as aiogram (Telegram bot framework), whisper or faster-whisper (for transcription), openai (for the GPT API), twilio (if using Twilio for WhatsApp), and possibly others like python-dotenv (for loading env files) or logging utilities. Ensuring these are installed is necessary for the application to run. *(No functions; just a dependency list.)*
* **.env configuration** – *(Not a tracked code file, but essential for setup.)* This file (referenced in documentation) contains configurable parameters for the bot. Key variables include the Telegram bot token, Whisper model selection, file size limit, admin user IDs, and timezone[[2]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/README.md#L16-L22). In this project, additional variables would be present for the new features: e.g. OPENAI\_API\_KEY for the summarizer, and Twilio/WhatsApp credentials such as TWILIO\_ACCOUNT\_SID, TWILIO\_AUTH\_TOKEN, WHATSAPP\_FROM (the sending WhatsApp number), and WHATSAPP\_TO (the target number or group). The application reads these to know how to connect to Telegram, which Whisper model to use, which users are admins, and how to contact external APIs for summarization and messaging.
* **telegram\_transcriber\_src/** – This is the main Python package containing the bot’s source code[[11]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/prompt.txt#L26-L31). It is a directory (with an \_\_init\_\_.py) that encapsulates all bot logic.
* **telegram\_transcriber\_src/main.py** – The entry-point of the application. This file launches the Telegram bot and ties all components together.  
  **Purpose:** It initializes the bot and starts the event loop. On startup it loads configuration from the environment (e.g. gets BOT\_TOKEN, WHISPER\_MODEL, etc.), configures logging (so that runtime information and errors are logged to file in logs/), and initializes the Telegram bot client. It sets up the bot’s dispatcher and registers the message handlers from the bot module. Finally, it starts the bot’s polling loop to listen for incoming Telegram messages indefinitely.  
  **Functions:**  
  – *There may not be many standalone functions here*, since this script primarily runs setup code. It might define a helper like load\_config() to read env variables or a main() function that encapsulates the startup sequence. Generally, its responsibilities are executed at import/run time:
  + If logging is configured here, it might call logging.basicConfig (not a custom function, but an important action).
  + It likely calls Aiogram’s mechanism to register handlers (either by importing the handlers which use decorators, or by explicitly linking handler functions to dispatcher).
  + If using Aiogram v2, it would call executor.start\_polling(dp, ...) to start listening. If Aiogram v3, it might use dp.run\_polling(bot). In either case, this file **starts the bot** and does not return until the bot is stopped.  
    *(In summary, main.py contains the application bootstrap code rather than reusable functions. Its “function” is essentially to configure and run the bot.)*
* **telegram\_transcriber\_src/bot/** – This sub-package contains the core bot logic, separated into modules for clarity[[12]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/prompt.txt#L28-L31). An \_\_init\_\_.py likely exists here to make it a package. The notable files inside are:
  + **handlers.py** – Defines the Telegram message handlers (the functions that respond to specific incoming message types or commands).  
    **Purpose:** To handle incoming Telegram updates (messages) and route them to the appropriate processing functions. It uses Aiogram’s decorator or router system to bind certain message types to specific functions. This is where the bot decides what to do when a voice message is received, or when a user issues a command.  
    **Functions:**  
    – handle\_voice\_message(message: types.Message) – Triggered when a **voice note** is received. This async function downloads the voice message file (using Telegram’s file API) to the downloads/ folder, then calls the transcriber to process it. It awaits the transcription result (text). Once transcription is done, it likely replies to the Telegram chat with the text (or a message indicating the text is being summarized, if summary is separate). Then it calls the summarization function (see transcriber.py) to get a summary of the text, and finally calls the WhatsApp send function to forward that summary. All these steps are orchestrated here so that from the user’s perspective, they get a transcription back in Telegram and a summary on WhatsApp.  
    – handle\_audio\_message(message: types.Message) – Similar to the above, but triggered when the bot receives an **audio file** (Telegram distinguishes voice notes vs general audio files). This function likely follows the same flow: save the file, transcribe it, respond with text, then send summary externally. It may even reuse the same logic as handle\_voice\_message internally (possibly by calling a common helper with the file reference).  
    – handle\_document\_audio(message: types.Message) – This could handle audio files sent as generic documents (e.g. an MP3 or WAV sent as a document). If implemented, it ensures that if the document’s MIME type is an audio format, the bot still processes it. (In some implementations, voice notes, audio, and audio-documents can all be handled by one unified handler by checking content type; but it’s likely separated for clarity.) The function would again delegate to the transcriber for processing.  
    – handle\_stats(message: types.Message) – Activated when an admin user sends the /stats command. It checks if the sender’s user ID is in the ADMIN\_USER\_IDS list (loaded from env) and, if authorized, returns some statistics about the bot usage. For example, it might count the number of files processed or the total audio duration transcribed, or simply the number of files in the downloads/ directory as a proxy for how many transcriptions were done. If the user is not an admin, this handler could ignore the command or reply with an unauthorized message. *(This function helps monitor the bot, but doesn’t affect the main transcribe->summary flow.)*  
    – **Helper subroutines**: The handlers file might also contain small helper functions or logic blocks used by the handlers. For instance, there might be a utility to **split a long transcript** into multiple Telegram messages (given Telegram’s message length limits)[[4]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/README.md#L46-L50). Another helper could be formatting the text or generating the filename for saving transcripts. These would typically be inner functions or just code within the handlers. Most heavy-lifting, however, is delegated to transcriber.py, so that handlers remain relatively thin coordinators.  
    *(The handlers module essentially maps incoming events to the appropriate processing and uses other modules to carry out the transcription and summarization.)*
  + **transcriber.py** – Implements the core processing: audio transcription via Whisper, text summarization via OpenAI, and message sending via WhatsApp API.  
    **Purpose:** To provide functions that take an audio file and return text (transcription), take text and return a summary, and send out messages to external services. This separation allows the handlers to simply call these functions without worrying about the details of Whisper or external APIs.  
    **Functions:**  
    – transcribe\_file(filepath: str) -> str – Loads the audio file from the given path and runs the Whisper model on it to produce a transcription. On initialization, this module likely already loaded the Whisper model into memory (e.g. model = whisper.load\_model(WHISPER\_MODEL) at the top, using the model name from env) so that this function can use it directly[[13]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/prompt.txt#L40-L46). The function handles Whisper’s output (which may include the full text and possibly segments or other info, depending on the library) and returns the transcribed text string. It may also handle exceptions (for instance, if the audio is not clear or no speech detected, it could return a message indicating that).  
    – summarize\_text(transcript: str) -> str – Sends the transcribed text to the OpenAI API to produce a summary. This function uses the OpenAI Python SDK (openai library) with the provided API key to create a completion or chat completion request. It likely constructs a prompt or message telling the model to summarize the given text. For example, it might use *ChatGPT* (gpt-3.5-turbo or similar) with a system or user prompt like: *“Summarize the following text in a few sentences: [transcript]”*. The API’s response is then parsed to extract the summary text. The function returns the summary string. (It might also enforce some max length on the prompt if needed to avoid exceeding token limits, perhaps splitting the transcript or truncating if very long – but for typical voice messages this is usually not an issue.)  
    – send\_summary\_to\_whatsapp(summary: str) -> None – Responsible for delivering the summary via WhatsApp. If Twilio is used, this function initializes a Twilio REST client with the account SID and auth token from env, then calls client.messages.create() with the from\_ number (the WhatsApp sandbox or business number), the to number (the target WhatsApp user or group), and the summary text as the message body. If the WhatsApp Cloud API is used instead, this function would issue an HTTPS POST request to the Meta Graph API endpoint with the appropriate authentication token and JSON payload. In either case, the function encapsulates the details of sending the message. It may log the result or any errors. This function doesn’t return a value – it triggers an external side-effect (the WhatsApp message) and lets the handler or main know if it succeeded (possibly via exceptions or log messages).  
    – **Other utilities**: The transcriber module might also have a helper to enforce the MAX\_FILE\_SIZE\_MB limit. For instance, a function within\_size\_limit(file: types.File) -> bool that checks the file size against the env config and returns True/False (the handlers could call this before downloading). Additionally, if audio format conversion is needed (whisper can typically handle many formats as long as ffmpeg is present), there might be logic to ensure the file is in a format Whisper accepts. This module might also define a global logger for logging transcribing progress or timing.  
    *(Overall, transcriber.py acts as the service layer: it knows how to transcribe audio, summarize text, and communicate with WhatsApp. The Telegram handlers invoke these functions to perform the heavy tasks.)*

*Aside from these, the repository may contain additional files such as unit tests or utility scripts (and the compare\_transcripts/ directory which we’ve excluded). However, the core operation revolves around the files listed above.* The separation of concerns is clear: **handlers.py deals with Telegram I/O, transcriber.py deals with data processing and external API calls, and main.py ties it all together and runs the bot**.

## Strengths of the Project

* **Clear Modular Design:** The codebase is organized logically into separate components for handling Telegram interactions and performing the transcription/summarization tasks. This modular structure (distinct files for handlers and processing) makes the code more readable and maintainable. New developers can quickly locate where certain functionality lives (Telegram bot logic vs. audio processing, etc.).
* **Use of Reliable Libraries:** Leveraging **OpenAI Whisper** for transcription ensures high-quality speech-to-text results. Similarly, using the **Aiogram framework** for Telegram provides a robust, well-tested foundation for bot interactions, and using the **OpenAI API** for summarization yields powerful summaries. These choices mean the heavy lifting is done by proven AI models and libraries, rather than reinventing the wheel.
* **End-to-End Automation:** The project automates the entire flow from message intake to output delivery. This is a strong point – a user can drop a voice note into Telegram and seamlessly receive a useful summary on WhatsApp without manual steps. It bridges platforms in a way that could save users time (for example, summarizing meeting memos or lengthy voice messages and forwarding the key points to a broader group on WhatsApp).
* **Dockerized Deployment:** Providing a Dockerfile and docker-compose configuration is a strength for developer experience and deployment consistency. Anyone can spin up the whole stack in a container with minimal setup, and the environment will be consistent (correct Python version, all dependencies, ffmpeg installed, etc.). This reduces the “it works on my machine” issues and is especially useful for deploying on servers or devices like a Raspberry Pi.
* **Configurable and Secure:** The use of a .env file for all tokens and config makes it easy to configure the bot for different environments without code changes. Sensitive info (API keys, tokens) are not hard-coded but injected via environment variables, which is a security best practice. Additionally, the inclusion of an admin ID list for privileged commands (like /stats) is a good security measure to prevent unauthorized use of certain features.
* **Logging and Persistence:** Storing incoming audio files under downloads/ and logging events to logs/ are good practices for persistence and debugging. If something goes wrong (e.g., a file fails to transcribe), a developer can inspect the saved audio or log output after the fact. The logs help in monitoring the bot’s activity and diagnosing issues over time.

## Weaknesses and Potential Issues

* **Blocking Calls in Handlers:** The bot’s design might perform heavy tasks (Whisper transcription, OpenAI API call, Twilio API call) within the async handlers directly. Whisper transcription can be CPU/GPU-intensive and OpenAI calls involve network latency. If these calls are not offloaded to background threads or tasks, the **Telegram bot may become unresponsive** to other incoming messages while it’s processing a request. In a high-traffic scenario or with long audios, this could be a bottleneck.
* **Lack of Error Handling & Resilience:** The current code may not comprehensively handle failure cases. For example, if the OpenAI summarization API fails (network error or rate limit) or Twilio’s API call fails, the bot might not catch exceptions to inform the user or retry gracefully. Also, if Whisper fails to transcribe (e.g., due to an unsupported format or no speech detected), the bot’s response might not inform the user appropriately. Robust error handling and user feedback are areas for improvement.
* **Scalability Constraints:** The design is single-bot instance oriented. Because it likely loads a Whisper model in memory, scaling to multiple instances or threads is not straightforward. Whisper models can be large and consume significant RAM/VRAM, so running multiple instances (to handle more simultaneous users) on the same host could be problematic. There’s also no load balancing or queue system for handling many transcription jobs at once.
* **Limited Scope of Summary Customization:** The summarization step currently produces a single generic summary. There’s no mechanism for the user to request different summary lengths or formats (e.g., bullet points vs. paragraph, or translations). Every voice note triggers an automatic summary. This one-size approach may not fit all use cases – for instance, sometimes a full transcript might be needed, but the system always pushes a summary to WhatsApp. In its current form, the user cannot easily choose to skip the summary or adjust it without code changes.
* **Dependency on External Services:** While using external APIs (OpenAI, Twilio) adds powerful functionality, it also introduces **external dependencies**. If the OpenAI API is slow or unreachable, the summarization will hang or fail. If Twilio/WhatsApp has issues, the final delivery fails. The system doesn’t appear to have fallbacks (like retrying later or caching results to send when possible). This reliance means the bot’s functionality is partially at the mercy of these services’ uptime and the user’s API quota.
* **Testing and Monitoring:** There is no mention of automated tests or continuous integration. The absence of unit tests or integration tests is a weakness, as future changes might introduce regressions. Also, while logs exist, there’s no built-in monitoring or alerting. If the bot crashes or an exception is thrown in a handler, it might simply stop processing without notification. A more robust system might include self-monitoring (like a watchdog or an external uptime monitor pinging the bot).

## Recommended Improvements (Prioritized)

1. **Implement Asynchronous Processing** (High Priority) – To prevent the bot from stalling on large tasks, run heavy operations off the main event loop. For example, use asyncio.to\_thread or background tasks for Whisper transcription and OpenAI calls. This way, the Telegram bot can continue responding to new messages while a transcription is in progress. It will greatly improve responsiveness under load.
2. **Robust Error Handling & User Feedback** (High Priority) – Add try/except blocks around external API calls (Whisper, OpenAI, Twilio). If transcription fails or no text is detected, catch the error and send a friendly message back to the user (instead of silently failing). If summarization fails, perhaps send the full transcription to WhatsApp as a fallback, or notify that summarization is currently unavailable. Similarly, handle WhatsApp send failures (maybe queue and retry, or alert the admin). Logging these exceptions to the logs is important for debugging.
3. **Configurability of Features** (Medium Priority) – Introduce options to customize behavior. For instance, allow an environment flag to turn **off** automatic WhatsApp forwarding (in case one wants to use the bot for transcription only). Or provide a Telegram command like /summary or /nosummary to let the user choose per-message whether a summary should be generated and sent. This flexibility would make the bot useful in more scenarios.
4. **Enhance Summarization Control** (Medium Priority) – Improve how summaries are generated. This could include: using prompt engineering to ensure the summary is of desired length, or splitting very long transcripts into sections and summarizing section-wise for better coherence. If users might want full transcripts on WhatsApp for longer audios, consider sending both a summary and perhaps a link to the full text file. Also, verify the summary process can handle the maximum possible transcript size (Telegram voice messages are limited by duration/file size, but in group chats users could send longer audio files as documents). Ensuring the summarizer doesn’t break on very large input (or implementing chunk-and-summarize iteratively) would make the feature more robust.
5. **Parallelize or Queue Transcription Jobs** (Medium Priority) – In anticipation of multiple users or group chats using the bot, implement a simple queue for transcription tasks. For example, if two audio messages come in at nearly the same time, the second could be queued and processed once the first is done. This could be as simple as using an asyncio.Lock or semaphore to ensure one Whisper runs at a time (to avoid memory strain), combined with user notifications like “Your message is in queue…” if waiting is needed. This improvement overlaps with asynchronous processing and would enhance stability under multiple concurrent uses.
6. **Monitoring and Persistence** (Low Priority) – Add health monitoring or at least an auto-restart policy. Since the bot is likely run via Docker, using restart: unless-stopped in docker-compose or a supervisor process can ensure it restarts on crash. Additionally, consider persisting conversation state or transcripts in a more structured way (database or cloud storage) if historical data is important. For now, transcripts are just stored as text files which is okay, but a future improvement could be to index them or provide a way to search past transcriptions.
7. **Documentation and Tests** (Low Priority) – Write unit tests for critical functions in transcriber.py (e.g., feeding a sample audio to transcribe function if possible, testing that summarization returns something for a sample text, testing that the WhatsApp send function is called with correct parameters by mocking Twilio). This will catch issues early when modifying the code. Also, expanding documentation (in the README or a wiki) on how the system works, how to extend it, and known limitations would aid future maintainers.

*(The above improvements are ordered by impact: making the bot non-blocking and error-tolerant will yield the biggest reliability gains, whereas documentation and nice-to-have features come later.)*

## Setup and Deployment Guide

Setting up this project for the first time requires obtaining a few API credentials and preparing the environment. Below are step-by-step instructions for developers new to the codebase:

1. **Prerequisites:** Ensure you have **Docker and Docker Compose** installed (for containerized deployment). If you prefer running directly on the host, have **Python 3.10+**, and install **FFmpeg** on the system (since Whisper relies on it for audio decoding)[[14]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Dockerfile#L4-L7). You’ll also need a **Telegram Bot Token** (create a bot via @BotFather on Telegram) and accounts/credentials for **OpenAI** (for summarization) and **WhatsApp API** (e.g. a Twilio account for WhatsApp sandbox, or WhatsApp Cloud API setup).
2. **Clone the Repository:** Download the code from the GitHub repository mevaser/telegram\_transcriber to your local machine. (If you have Git installed, you can use git clone <repo\_url>.)
3. **Create the Environment File:** In the project root, create a file named .env (or duplicate the provided example if one exists). Open it in a text editor and fill in all required configuration values. For example:

* BOT\_TOKEN=<your\_telegram\_bot\_token>  
  WHISPER\_MODEL=base # model name for transcription (tiny, base, small, etc.)  
  MAX\_FILE\_SIZE\_MB=25 # max file size to process  
  ADMIN\_USER\_IDS=1234567890 # your Telegram user ID (or comma-separated list of admin IDs)  
  TIMEZONE=UTC   
  OPENAI\_API\_KEY=<your\_openai\_api\_key>   
  TWILIO\_ACCOUNT\_SID=<your\_twilio\_sid>   
  TWILIO\_AUTH\_TOKEN=<your\_twilio\_auth\_token>   
  WHATSAPP\_FROM=<whatsapp\_sender\_number>   
  WHATSAPP\_TO=<whatsapp\_recipient\_number>
* Ensure no quotes around the values and no spaces around =. The Telegram user IDs in ADMIN\_USER\_IDS should be the numeric IDs of users allowed to use admin commands (you can put your own ID so you can use /stats). The WhatsApp numbers should include the country code and the whatsapp: prefix if required by the API (e.g., whatsapp:+14151234567). [[2]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/README.md#L16-L22)

1. **Install Dependencies:** There are two options:
2. **Docker Compose:** The easiest way – simply run docker compose up -d in the project directory. This will build the Docker image (downloading all dependencies) and start the bot container in the background. The bot will automatically use the .env file for configuration.
3. **Manual (local Python):** Make sure you have Python 3.10 and ffmpeg available. Create a virtual environment and activate it. Then run pip install -r requirements.txt to install Python dependencies. Once done, you can start the bot with python -m telegram\_transcriber\_src.main. (It will read the .env by itself if the code uses python-dotenv; if not, ensure environment variables are exported in your shell.)
4. **Running the Bot:** If using Docker, the bot should be running as a service after the compose up. Monitor the logs with docker compose logs -f bot (assuming the service is named “bot”). You should see output indicating it connected to Telegram (and possibly that Whisper model was loaded). If running locally, simply executing the module will start polling Telegram for messages. In the console, you’ll see log prints when the bot starts.  
   *Tip:* To run the bot in the background on a remote server (without it stopping when you disconnect), you can use Docker (which naturally runs in background as a daemon). If running via Python directly on a server, consider using something like nohup python -m telegram\_transcriber\_src.main & to keep it alive after logout.
5. **Testing the Setup:** Open Telegram and send a voice note or audio file to your bot (you should have added the bot to your contacts or a group and have started a conversation so it can send you messages). The bot should respond with the transcribed text within the Telegram chat. Shortly after, you (or the configured WhatsApp recipient) should receive a WhatsApp message from the bot (via the configured number) containing the summary of that transcription. This will confirm that Telegram, Whisper, OpenAI, and WhatsApp integrations are all functioning.
6. **Deployment Considerations:** For a production or long-term deployment, ensure the machine running the bot has sufficient resources (CPU for Whisper, possibly GPU if using larger models and you configured a CUDA image). If using a GPU, modify the Dockerfile to use a CUDA base image and install CUDA-enabled PyTorch[[15]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/README.md#L32-L40). Keep your .env secrets secure (do not commit it to any repo). It’s also wise to set up some monitoring – at minimum, check the logs periodically or use Docker’s restart policy to auto-reboot the bot if it crashes. Regularly update the OpenAI and Twilio credentials if they expire.

By following these steps, a developer or DevOps engineer unfamiliar with the project should be able to get the Telegram Transcriber bot up and running, and understand how to configure and deploy it in their own environment.

[[1]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/README.md#L6-L10) [[2]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/README.md#L16-L22) [[3]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/README.md#L5-L10) [[4]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/README.md#L46-L50) [[15]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/README.md#L32-L40) README.md

<https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/README.md>

[[5]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Dockerfile#L4-L12) [[6]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Dockerfile#L16-L24) [[7]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Dockerfile#L20-L28) [[8]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Dockerfile#L34-L37) [[14]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Dockerfile#L4-L7) Dockerfile

<https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Dockerfile>

[[9]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Makefile#L9-L17) [[10]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Makefile#L27-L32) Makefile

<https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/Makefile>

[[11]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/prompt.txt#L26-L31) [[12]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/prompt.txt#L28-L31) [[13]](https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/prompt.txt#L40-L46) prompt.txt

<https://github.com/mikprin/audio_transcriber_bot_gpt/blob/21b739829ebb2a3e0073d1546375c5a3d80eb875/prompt.txt>