ChatGPT

WhatsApp Message Processing Pipeline

The pipeline processes inbound WhatsApp messages in four main stages: (1) **data ingestion** from the message-funnel table, (2) **audio transcription and transliteration**, (3) **small-talk vs. query classification**, and (4) **theme clustering and reclassification**. The starting point is a database table (`glific_messages_funnel`) containing messages (text and audio) from community members. Each message has fields like `id`, `message_type`, `body_final`, and `body_final_phonetic`. We focus only on **inbound** messages (from members to the system). As the table is large, the scripts fetch rows in batches, process them, and update the database. This ensures efficiency and avoids reprocessing. The overall goal is to convert any audio into text, distinguish genuine user questions from casual “small talk,” and then assign each query message to a theme (e.g. *Delivery*, *Nutrition*, etc.). Each step is designed to enable downstream analysis and response generation.

1. Data Setup and Message Funnel

First, the script connects to the Postgres database (using SQLAlchemy/psycopg2) and queries the `glific_messages_funnel` table for relevant rows. For audio transcription, it selects rows where `message_type = 'audio'`, the text column (`body_final`) is empty, and `media_url` is not null. It fetches these in batches (e.g. 50 rows) ordered by `id`. For classification, it fetches rows where the `question_type` is null (unlabeled) and `message_type` is either text or audio. For theme clustering, it fetches rows where `question_type = 'query'` and text is present. This batching ensures incremental processing. Any errors (e.g. DB fetch failures or update issues) are logged for troubleshooting.

2. Audio Transcription and Transliteration

All inbound **audio** messages are first transcribed. The pipeline downloads each audio file from its `media_url`, saves it temporarily, and calls OpenAI’s Whisper model (via `openai.audio.transcriptions.create` with `model="whisper-1"`) to convert speech to text [1]. Whisper supports languages like Hindi/Marathi/Urdu (we pass `language="hi"` for Hindi). This yields a Hindi text transcript for each audio. Transcription is performed in parallel threads (`ThreadPoolExecutor`) to speed up processing, with a small sleep between batches to respect API rate limits. Any failures (e.g. download errors or API timeouts) are logged.

Once we have the Hindi text, we **transliterate** it into Latin script. This uses a GPT-4 chat completion with a system prompt instructing it to convert Hindi/Urdu/Marathi text to its phonetic English representation. The rationale is that further text processing (like classification or keyword matching) is easier on Latin characters. For example, the Hindi text “मेरी डिलेवरी कल है…” is transliterated to “Meri delivery kal hai…”. Each transcript and its phonetic form are then written back into the database (`body_final` and `body_final_phonetic` columns). This step ensures that all content, regardless of original media type, is available as text in a consistent script.

Key steps in audio transcription:
- Fetch a batch of audio-only messages.
- **Download audio** file via its URL; save to a temp file.
- **Transcribe** speech to Hindi text using Whisper.

- **Transliterate** Hindi text to phonetic Latin script using GPT-4.
- **Update DB** with the transcript and transliteration for each `id`.

By the end of this stage, every inbound message (audio or text) has text in `body_final_phonetic`. This unified text form enables subsequent NLP tasks.

3. Small-Talk vs. Query Classification

Next, each message's text is classified as either a **legitimate query** or **small talk**. This uses a fine-tuned GPT-based classifier. A fine-tuned model is essentially a pre-trained language model adapted on a labeled dataset for specific categories [2]. Here we have a fine-tuned GPT model that labels an input message as `query` (meaning the user is asking a substantive question), `small-talk` (e.g. greetings, chit-chat), or possibly another label. The script fetches text (from `body_final_phonetic`) where `question_type` is null. For each text row:
- If the message is audio **with no text** (should not occur after transcription but just in case), it auto-labels it "small_talk" (assuming accidental noise rather than a question).
- Otherwise it calls `openai.chat.completions.create` with the fine-tuned model, providing a prompt that asks it to **classify** the message as "query" or "small-talk." The model returns a single-word label.

Using a classifier trained (fine-tuned) on relevant examples ensures better accuracy in distinguishing actual questions from greetings or irrelevant chatter. This is crucial to filter out noise and focus only on messages that need expert response. Each resulting label is written back to the `question_type` column in the database for the message ID. The process logs each classification and repeats in batches until all messages are labeled.

Why fine-tuning? Instead of raw zero-shot GPT, a fine-tuned model leverages task-specific training data for accuracy [2]. It learns subtleties like distinguishing "Pregnancy mein private part garm Pani se dho sakte hain kya" (a real question) from "Hi" or "Kaise ho" (small talk). The script also records a log of classifications to CSV for auditing.

4. Theme Clustering with Embeddings

All messages labeled as **query** are then analyzed for thematic grouping. First, we obtain semantic **embeddings** for each query message using OpenAI's `text-embedding-3-large` model. Embeddings are high-dimensional numeric vectors that capture the meaning of a sentence. Before embedding, the text is cleaned: normalized (e.g. mapping "dard"→"pain"), stopwords removed, and character n-grams extracted to improve semantic capture.

Once we have embeddings for, say, thousands of query messages, we reduce dimensionality with UMAP (Uniform Manifold Approximation and Projection). UMAP projects high-dimensional vectors into a lower (e.g. 10) dimensional space while preserving structure. UMAP is often used as a preprocessing step to aid clustering [3]. In practice, UMAP can help algorithms like HDBSCAN find clusters more effectively because it condenses data manifolds. (Caution: UMAP can distort density, but it generally enhances cluster separation [3].)

We then apply **HDBSCAN**, a density-based clustering algorithm, to the UMAP-reduced vectors. HDBSCAN groups similar vectors into clusters without specifying the number of clusters ahead of time. It also labels outliers as noise (`cluster_id = -1`). In code, we set `min_cluster_size=10` and `min_samples=5` (tunable) and fit HDBSCAN to the data. This yields a `cluster_id` for each message. For example, messages asking about delivery timing might form one cluster, while those about nutrition form another. This approach of using OpenAI embeddings with UMAP and HDBSCAN has been shown effective for document clustering [4] [3].

After clustering, we need human-readable **themes** for each cluster. For each identified cluster (except the `-1` noise cluster), the code samples a few messages from that cluster and sends them to GPT-4 with a prompt listing possible themes (Delivery, Newborn Health, Nutrition, etc.). GPT-4 then picks the most fitting theme label for that cluster (e.g. "Delivery" or "Diet and Nutrition"). This uses GPT-4's natural language understanding to summarize clusters. The script collects one theme per cluster and maps all cluster members to that theme. For outliers (`-1`), we assign a default theme "General." The theme labels are stored in the `theme` column in the database. This means every query now has an assigned topic.

Step-by-step:
- **Fetch** all messages where `question_type = 'query'` and text is non-empty.
- **Embed** each text using OpenAI embeddings (transform to numeric vector).
- **Reduce** vectors with UMAP to ~10 dimensions [3].
- **Cluster** the reduced vectors with HDBSCAN (groups similar messages) [4].
- **Label clusters** by GPT: sample messages from each cluster, prompt GPT with a fixed theme list, and record the returned theme string.
- **Update DB**: write the theme string for every message in that cluster.

This unsupervised clustering followed by GPT-based labeling helps organize open-ended user queries into manageable categories. Visualization (e.g. a 2D UMAP+PCA scatterplot) can be generated for exploration, though the script focuses on data processing.

5. Theme Reclassification with Rules and GPT

Finally, the initial cluster-generated themes are refined by a **rule-based + GPT fallback** process. There is a fixed list of allowed themes (`General, Breastfeeding, Delivery, Fetal Movement, Diet and Nutrition, Newborn Baby Health, Maternal Pain or Sickness, Government Schemes`). For each query message, the script first tries simple keyword rules: e.g. if the text contains "delivery" or "birth," label it *Delivery*; if it contains "dard" or "pain," label it *Maternal Pain or Sickness*; etc. This catches clear cases with minimal cost.

If no rule matches, we then use a GPT-4 classifier. We send the message text to GPT-4 with a prompt asking it to pick one of the allowed themes. GPT's response is parsed; if it is one of the allowed themes, we adopt it. If GPT fails or returns something else, we default to *General*. This ensures every message ends up with exactly one of the approved themes. The script processes messages in batches (50 at a time) where the theme is empty or contains a comma (indicating multiple or unset). Each batch: apply keyword rules, then GPT for the rest (using a ThreadPoolExecutor to parallelize up to 5 concurrent GPT calls). All changes are logged, and the database is updated with the new theme.

Why this two-step approach? It combines the efficiency and precision of rules for obvious cases with the flexibility of GPT for the others. This helps enforce a consistent theme taxonomy. As a result, any message originally tagged as *query* will end this pipeline with a single curated theme label.

Summary

In summary, the four scripts form a pipeline that transforms raw inbound WhatsApp messages into structured, analyzable data. First, any audio is transcribed (via Whisper) and transliterated into Latin script [1]. Next, each message is classified as a *query* or *small-talk* using a fine-tuned GPT model (leveraging pre-trained knowledge and task-specific training [2]). Then, all query messages are embedded and clustered (using OpenAI embeddings + UMAP + HDBSCAN [4] [3]) to discover thematic groups, each labeled by GPT-4. Finally, every query's theme is refined through keyword matching and a final GPT classification to ensure it fits a predefined list. Each stage updates the `glific_messages_funnel` table so that, in the end, every inbound query message has clean text, a query label, and an assigned theme. This comprehensive process enables efficient handling of community questions and supports downstream analytics or automated response systems.

Sources: OpenAI Whisper API documentation [1]; principles of fine-tuning GPT models [2]; tutorial on text embeddings, UMAP, and HDBSCAN for clustering [4] [3].

---

[1] OpenAI's Whisper API tutorial | Teemu Maatta | Medium
https://tmmtt.medium.com/whisper-api-speech-to-text-2e7e366f9ec6

[2] Real-World Example: Building a Fine-Tuned Model with OpenAI | by Sriram Parthasarathy | GPTalk | Medium
https://medium.com/gptalk/real-world-example-building-a-fine-tuned-model-with-openai-58a0557143af

[3] Using UMAP for Clustering — umap 0.5.8 documentation
https://umap-learn.readthedocs.io/en/latest/clustering.html

[4] Clustering Documents with OpenAI embeddings, HDBSCAN and UMAP – Dylan Castillo
https://dylancastillo.co/posts/clustering-documents-with-openai-langchain-hdbscan.html