

# ACE-Step Data-Tool

Modular tool for the automated creation of datasets for the ACE-Step music model.

It automatically generates accompanying text data (song lyrics and descriptive tags) from audio files for training data generation.

## Contents

- Overview
- Technical Architecture & Main Functions
- Installation & Setup
- Dependencies
- Usage Examples
- Model Compatibility & Hardware Requirements
- Project Structure

## Overview

ACE-Step Data-Tool is a data preparation tool for audio data, specifically developed to automatically create training data for the ACE-Step music AI model. It scans a folder full of audio files (e.g., songs) and generates additional text files for each track – such as the lyrics and a list of keywords/tags describing the song's mood, genre, instrumentation, etc. These files serve as accompanying data, for example, to train an AI model like ACE-Step with suitable input prompts (tags + lyrics) or to analyze music metadata.

Main goals of the tool are:

- **Automation** (minimizes manual steps in data preparation)
- **Modularity** (clear, separate components for lyrics, BPM, tagging, etc.)
- **User-friendliness** (easy to use via a web interface or scripts, adaptable to different music genres and models)

The tool is aimed at developers and researchers who work with Python/Conda and audio datasets, and enables the quick generation of descriptive metadata for a music collection – without outdated, manual workflows.

## Technical Architecture & Main Functions

The ACE-Step Data-Tool consists of several components that work together in a processing pipeline, supported by an optional Gradio web interface for convenient operation:

1. **Directory Scan & File Processing:** By default, the tool expects a folder (`data/` in the project directory) containing audio files (supported: `.mp3`, `.wav`, `.flac`, `.m4a`). All files (recursively in subfolders) are read and processed one after another. Intermediate results and logs are displayed for each track.
2. **Metadata Extraction:** Initially, the artist and title are extracted from each audio file (via ID3 tags, using the `TinyTag` library). This information is used later for lyrics search. If a file has no tags (e.g., untagged WAVs), the filename is used as the title.
  - **Note:** Ensure that the audio files have correct artist and title tags for reliable lyrics detection.
3. **Lyrics Acquisition (Lyrics Scraping):** The tool attempts to automatically retrieve the lyrics for the respective piece from the internet. It uses Genius.com as the source, first forming a suitable lyrics URL from the artist/title data. If the direct URL is unsuccessful, a search query is executed on Genius and

the first matching result is used. The lyrics are extracted as plain text (using `Requests + BeautifulSoup4`) and saved in a file `<Name>_lyrics.txt`.

- **File Format:** Each `<Name>_lyrics.txt` file initially records the artist and title, followed by the complete lyrics (UTF-8 encoded).
- **Note:** A short pause is inserted between requests to avoid overloading the target website. If no text is found, the tool aborts processing of that song (i.e., no tag generation occurs without lyrics – purely instrumental tracks or unknown songs are skipped with a corresponding note).


4. **Tempo Analysis (BPM Detection):** In parallel, the approximate tempo in BPM (Beats Per Minute) is determined from the audio signal. For this, `Librosa` (audio processing library) is used: onset detection and rhythm functions are utilized to estimate the dominant tempo in the first seconds of the song. Extreme values are normalized into a sensible range (e.g., BPM over 140 is halved; under 60 is potentially doubled) to compensate for half/double counting. The result (integer) is used as the BPM value for tag generation. If automatic detection fails, the tool tries to read a BPM number from the filename (e.g., `songname_bpm120.mp3`). Otherwise, BPM remains unknown.

5. **Tag Generation (Keyword List):** As a central feature, the tool generates a list of descriptive tags per track, mapping the song's genre, mood, instrumentation, vocal type, etc. These tags are generated using an LLM-based AI: The tool uses a local language model (Large Language Model) via an Ollama API integration to "invent" suitable keywords from the available information.

- A structured prompt is given to the model, providing all available metadata: Artist, Title, BPM of the song, an excerpt of the lyrics (up to ~300 characters) for contextual orientation, and optional style hints. These style hints are either automatically composed from user-selected settings (see Presets/Mood) or a manually entered prompt addition.
- Additionally, the model receives fixed rules in the prompt on how the output should look: It must generate 12–14 tags in lowercase and hyphen notation (e.g., `dark`, `bass-heavy`, `female-vocal`, etc.), must include a BPM tag (format `bpm-XXX`), a maximum of 2 genre tags, and at least one tag each from the categories Vocals, Instruments, Mood, Rap Styles. An example collection is provided in the prompt so the model can recognize the style.
- As the result of the model query, the tool receives a string with the tags. These are post-processed and filtered (nonsensical or overly long terms are removed, format is standardized) and saved as a list. It is also ensured that the BPM tag (if missing or at a later position) is inserted or sorted to the beginning.
- The final tag list is then saved in a text file `<Name>_prompt.txt` (tags comma-separated on one line). This file represents the prompt for the ACE-Step model and can, for example, be used in music generation or in a dataset as a description.

6. **Gradio Web Interface:** The tool offers an intuitively operable web UI (based on Gradio), via which all the above steps can be executed with a few clicks. When starting the tool (see Setup below), a local web service opens on `http://127.0.0.1:7860` with the following functionality:

- **File Scan & Start:** A click on "Start Tagging" scans the defined `data/` folder and automatically processes all found audio files as described above. Progress is logged live in the interface (for each song, it is displayed whether lyrics were found and whether tags were saved).
- **Lyrics Option:** A checkbox "Overwrite lyrics" controls whether existing lyrics files should be overwritten. If this option is deactivated, already existing `<Name>_lyrics.txt` files are retained (no re-downloading of lyrics), saving time if you repeat a run.
- **Genre Presets:** In a "Genre-Preset" dropdown, a predefined genre/style preset can be selected. These presets are simple text or JSON files in the `presets/` folder and contain certain keywords (e.g., a "HipHop" preset might specify terms like "90s, boom bap, underground"). The selected preset text flows as a style guide into the prompt for the model, so the generated tags are stylistically adapted.
  - **Note:** You can place your own preset files in `presets/` – they are automatically loaded upon startup.
- **Mood Slider:** The "🎭 Mood: Sad ↔ Happy" slider allows variation of the mood preset. A neutral value (0.0) means no explicit mood preset. For positive values (>0.3), the style guide is automatically supplemented with a "happy" tag, for negative values (<-0.3) with "sad". This influences, for example, the resulting mood tags (whether "chill"/"happy" or "dark"/"sad" is generated).

- **Prompt Addition:** A "  Prompt addition (optional)" text field allows manually entering additional keywords or hints that are also passed to the model. If text is entered here, it completely overrides the above Presets/Mood settings – so you can specifically provide your own descriptions (e.g., "orchestral, melancholic, strings, 80s") to steer the tag generation in a certain direction.
  - **Export Function:** After successful processing, all generated files can be exported collectively via the interface. An input field allows selecting a destination folder, and the "Export" button copies all relevant files there (i.e., the audio files as well as the respective `_lyrics.txt` and `_prompt.txt`). This allows you, for example, to create a complete dataset folder without altering the original files.
7. **Batch Scripts & Extensions:** For advanced users, the project also includes a console entry point (see Installation), so the tool can be called via script (`ace-data` command). Additionally, helper scripts exist in the `tools/` directory, e.g., for audio conversion (`mp3_to_wav.py`, `Batch wav_to_mp3.bat`) or experimental approaches to genre classification with pre-trained models (`classify_music.py` uses a HuggingFace model for genre recognition). These are optional and not essential for the main pipeline. The core data pipeline focuses on the steps described above: Lyrics → BPM → LLM Tagging.

## Installation & Setup

The following describes setting up the development environment and installing the ACE-Step Data-Tool. It is assumed that Python and git are installed. We recommend using Miniconda to manage the Python environment.

1. **Get Repository:** Clone this GitHub repository to your computer:

```
git clone https://github.com/methmx83/ace-data_tool.git
```

(Alternatively, you can download the project as a Zip.)

2. **Python Version & Environment:** Ensure a current Python version is installed. The tool requires Python  $\geq 3.7$ ; Python 3.13 is recommended (tested on 3.13 in the target environment). Create a new Conda environment (optional, but recommended) with the appropriate Python version:

```
conda create -n ace-data_env python=3.13
conda activate ace-data_env
```

This ensures that dependencies are installed isolated and compatibly.

3. **Install Dependencies:** Change to the project directory and install the required Python packages. All dependencies are defined in `setup.py` – you can install the tool directly as a package:

```
cd ace-data_tool
pip install -e .
```

This will install all required libraries (see below) and register the console script `ace-data`.

Alternatively, you can install the packages manually via `pip install -r requirements.txt` if such a list is provided.

4. **External Requirements (Ollama & Model):** For tag generation, an Ollama server with the required language model must be running. Therefore, install Ollama (version 0.9.6 or newer, see Ollama Docs) on your system. Then, load a compatible language model into Ollama. In the default config file (`config/config.json`), the model name `"deep-x1_q4:latest"` is entered – this indicates a local model (e.g., a quantized Llama derivative). Ensure that either this model is available or adjust the `model_name` in the config to an installed model. Start the Ollama service so the API is accessible at `http://localhost:11434` (standard port for Ollama).
  - **Note:** Ollama is cross-platform (Windows may require WSL or a native porting, version 0.9.6 supports Windows natively). Without a running LLM server, tag generation cannot occur – in this case, the tool would stop after lyrics for each song.

5. **Prepare Folder:** Create a `data` folder in the project directory (if not already present) and copy your audio files there. You can also maintain the subfolder structure if the songs are already sorted by albums/genres etc. – the scan runs recursively.
6. **Start the Tool:** After successfully installing the packages and setting up the Ollama model, you can start the application. Under Windows, the `ace-data` script was registered during installation via `pip`. Execute in your activated environment:

```
ace-data
```

This starts the Gradio web interface. After a few seconds, the console output `Running on local URL: http://127.0.0.1:7860` should appear. Open this address in your browser to access the user interface.

- **(If the command is not found, check the installation or alternatively use `python -m webui .app` in the project folder.)**

7. **Usage via UI:** In the web interface, you can now make the desired settings (see above: Overwrite Lyrics, Preset, Mood, Prompt) and then click "Start Tagging".  
The tool then processes all files in the `data/` folder. Monitor the log outputs to detect any problems (missing lyrics, model errors, etc.). After a successful run, you will find two new files for each track in the `data/` directory: `<Name>_lyrics.txt` and `<Name>_prompt.txt`. Optionally, use the Export function in the UI to copy all results collectively to another folder (practical for transferring the audios plus tags, e.g., into a training dataset).

## Dependencies

The project uses current Python libraries to implement audio processing, web scraping, and ML integration. A shortened overview of the most important dependencies and tools:

- **Python  $\geq 3.10$**  (recommended 3.13) – newer versions are recommended to ensure compatibility with all libraries.
- **Gradio (v5.35 or higher)** – for the WebUI. Enables the simple creation of interactive interfaces to control the workflow.
- **Librosa** – for audio analysis (especially tempo/BPM detection). Internally, Librosa uses `numpy`, `scipy`, `soundfile`, etc., to load and process audio data.
- **SoundFile** – for loading audio files (Librosa backend, requires the system-wide library `libsndfile` – when installed via Conda, this is automatically installed, with `pip` it is included in the `pysoundfile` package).
- **Requests + BeautifulSoup4** – for downloading song text web pages and parsing the HTML content from Genius.
- **Tinytag** – for reading audio file metadata (artist, title, album, etc. from common formats like MP3, FLAC, M4A).
- **NLTK (Natural Language Toolkit)** – for sentiment analysis and text normalization of lyrics. The tool downloads the sentiment data (`vader_lexicon`) and stopword lists upon first use.
- **numpy, scipy, matplotlib** – common packages for numerical calculations; `matplotlib` is hardly used within the tool itself (was possibly intended for debugging).
- **Ollama** – external service for LLM queries. This is not a Python library, but a runtime that can execute LLMs locally (similar to a local ChatGPT server). It is addressed via HTTP. Please install Ollama separately, as described above. Alternatively, you could theoretically also use your own API endpoints of another LLM (e.g., OpenAI API), but the tool is preconfigured for the Ollama protocol.
- **CUDA 12 (optional, for GPU support in LLM operation)** – If the language model used utilizes GPU acceleration (e.g., via Ollama/llama.cpp with CUDA support), the corresponding NVIDIA drivers and CUDA toolkit should be installed. The recommended setup uses CUDA 12.9 under Windows for optimal performance with newer NVIDIA cards.

The installation of Python dependencies is done, as shown above, via `pip`/Conda. Make sure to install in the appropriate environment. For Windows users, it is advisable to install Visual Studio 2022 Build Tools (as provided in the recommended configuration), as some Python packages (if no wheel is available) require a

C++ compiler. In our case, however, most libraries are installed as precompiled wheels, so no manual compilation should be necessary.

**Hardware Recommendation:** The tool was tested under Windows 10 Pro x64 with an Intel i9 (24 threads) and an NVIDIA RTX 4070 (12 GB VRAM). 64 GB RAM was available. Similar or slightly weaker configurations should work. At least 8–12 GB of graphics memory is advisable so that the LLM model fits into the GPU memory – see the section below on model compatibility.

## Usage Examples

In normal use, no additional coding is necessary – the tool handles everything at the push of a button via the UI. Nevertheless, a few application examples and typical outputs are shown here to illustrate the behavior:

1. **Basic Run (process all files):** After placing your MP3s in `data/`, start `ace-data`. In the browser UI, you can, for example, set the genre preset to "HipHop", the mood slider to slightly positive (+0.5), and then click Start Tagging. Assuming there is a file `Song1.mp3` in the folder (Artist: Imagine Dragons, Title: Believer).
  - The tool will: load the lyrics, calculate BPM (e.g., ~125), and generate tags. The log display shows something like:

```
Song1.mp3 - ✓ Saved lyrics and prompt
BPM: 125 - TAGS: bpm-125, male-vocal, rock, drums, energetic,...
```

This signals that the files were successfully created. In the file system, you now find:

- `Song1_lyrics.txt` – the complete lyrics to "Believer", beginning with, e.g., "Artist: Imagine Dragons\nTitle: Believer\n\nFirst things first...".
- `Song1_prompt.txt` – the tags, e.g.:

```
bpm-125, male-vocal, rock, guitars, aggressive, anthem, pop-rock, upbeat,
```

(This is a fictional example of ~10 tags; the actual list may vary slightly. The important thing is the format: lowercase, comma-separated, possibly hyphens instead of spaces.)

2. **Overwrite/Retry Lyrics:** If a song found no lyrics on the first run (the log shows something like "X No lyrics found."), you can manually correct (e.g., improve the file's ID3 tags or place the lyrics manually as `<Filename>_lyrics.txt` in the folder) and then run the tool again. In this case, activate the "Overwrite lyrics" option so the tool tries again to retrieve the text online (or overwrites your manually created file) despite an existing lyrics file. Without this option, it would retain existing `_lyrics.txt`.
3. **Usage as Python Module:** All core functions are also programmatically usable. For example, you can access the modules in your own Python script if you want to automate something:

```
from scripts.bpm import get_bpm
bpm_val = get_bpm("audio/path/song.wav")
print("Detected BPM:", bpm_val)

from scripts.lyrics import fetch_and_save_lyrics
fetch_and_save_lyrics("Imagine Dragons", "Believer", "output_folder/Believer_lyrics.t

from scripts.tagger import generate_tags, save_tags
tags = generate_tags("output_folder/Believer.mp3", prompt_guidance="rock, energetic")
save_tags("output_folder/Believer.mp3", tags)
print("Generated Tags:", tags)
```

This exemplary code would determine the BPM of a song, load lyrics from Genius, and generate tags, without using the Gradio interface. Note that `generate_tags` requires access to the running LLM server – so make sure Ollama with the model is running, otherwise a timeout/error will occur. The

`save_tags` function saves the tags in the `<Song>_prompt.txt` file and ensures that the lyrics file is cleaned of technical artifacts (removal of special characters/placeholders from the text).

4. **Exported Dataset Structure:** If you use the Export function or manually copy the `data/` folder elsewhere after processing, you get a dataset that can be used directly for training purposes. A possible excerpt of the folder structure after processing might look like this:

```
data/
├── Album1/
│   ├── Track01.mp3
│   ├── Track01_lyrics.txt
│   ├── Track01_prompt.txt
│   ├── Track02.mp3
│   ├── Track02_lyrics.txt
│   └── Track02_prompt.txt
└── Album2/
    ├── SongA.mp3
    ├── SongA_lyrics.txt
    └── SongA_prompt.txt
```

Each song now lies with its associated lyrics and prompt text file. This text data can, for example, be fed into the training of an ACE-Step model (lyrics as the desired vocal text, tags as musical context/style). Note that the tool does not extract actual audio features – it generates only descriptive meta-information.

## Model Compatibility & Hardware Requirements

The quality and speed of tag generation strongly depend on the language model (LLM) used. The ACE-Step Data-Tool was deliberately built to not require an external cloud API, but to work with a local model – so copyrighted lyrics also remain on your own computer. By default, the model name `deep-x1_q4` is set in `config/config.json`. This should be a compatible model in Ollama format (e.g., a derivative of an LLaMA-2 model with ~7–13B parameters, quantized to 4 bits for lower VRAM consumption).

- **Compatible Models:** Generally, chat-optimized models (e.g., LLaMA2-Chat, GPT-J, GPT-4-X Alpaca, etc.) that can follow instructions are suitable. It is important that the model has sufficient context length ( $\geq 4k$  tokens) and competence to generate meaningful tags based on short lyrics + rules. Smaller models ( $< 7B$  parameters) might have difficulty delivering consistent genre/mood tags. In the test, a LLaMA2-based model (~7B, 4-bit) delivered satisfactory results.
  - Theoretically, any model available via Ollama can be used – to change it, adjust `model_name` in the config. Make sure the model understands the Markdown formatting and roles ("system" and "user") specified in the prompt (most newer chat LLMs do).
- **Hardware & Performance:** The intended model setup requires a powerful GPU. With the recommended NVIDIA RTX 4070 (12GB VRAM), the model named in `config.json` could be loaded quickly and queries answered within a few seconds per song. If you use a GPU with less memory (e.g., 8 GB), loading might fail or the model server automatically switch to CPU operation.
  - **Warning:** In CPU mode, tag generation takes significantly longer (several minutes per song are possible, depending on the model).
  - If only little GPU memory is available, consider:
    - a smaller or more heavily compressed model (e.g., 4-bit quantization, 7B parameters or less),
    - or use Ollama's options like `--gpu-offloading` / `--cpu-offload` to offload parts of the model.
  - Basically:  $\geq 12$  GB VRAM is recommended to work smoothly. A multi-core CPU also speeds up audio processing (`Librosa`) and HTML parsing, which is usually not the bottleneck.
- **Limitations:** Note that the quality of the automatically generated tags is not perfect – it depends on the model's knowledge and the lyrics. The tool prioritizes rap/hip-hop tags when it detects rap terms (see `Moods.md` for predefined lists of genres, moods, instruments, rap styles). For very unusual or instrumental pieces, the model might be inaccurate. Furthermore, songs without findable lyrics are currently not tagged – i.e., purely instrumental tracks receive no output except the note that lyrics are



missing. This limitation is intentional, as the tags strongly depend on lyrical content and moods. In future versions, audio feature-based tagging models could be integrated here (see `tools/classify_music.py` for approaches), but as of July 2025, the tool focuses on text-based metadata.

## Project Structure

To facilitate getting started with the code, here is a simplified overview of the repository structure:

```

ace-data_tool/
├── webui/
│   └── app.py - Gradio App (UI-Layout, Start/Stop, Button-Callbacks)
├── scripts/ - Core functions for data processing:
│   ├── lyrics.py - Lyrics Scraping (Genius API Scraper)
│   ├── bpm.py - BPM Detection with Librosa
│   ├── moods.py - Text analysis (Sentiment, Tag Cleaning, Preset Loader)
│   └── tagger.py - Tag Generator (LLM API Call, Prompt Definition)
├── include/ - Helper functions and configuration:
│   ├── preset_loader.py - Loads genre presets from the presets/ folder
│   ├── clean_lyrics.py - Cleans lyrics (Removes special characters, notes)
│   ├── metadata.py, write_metadata.py - Alternative metadata handler (JSON Export)
│   └── prompt_editor.py - (optional, UI element for prompt editing)
├── presets/ - Predefined preset files (Genres, Moods, etc., extendable by the user)
├── tools/ - Additional Tools/Optionals:
│   ├── convert_tools/ - Audio conversion (mp3 <-> wav batch scripts)
│   └── wave_processing/ - Audio analysis (e.g., genre classification with ML, experiments)
├── config/
│   └── config.json - Configuration file (Path to input folder, LLM model name, API URL, etc.)
├── README.md - (This documentation)
└── System_Specifications_music-scraper.md - Recommended system environment & hardware details

```

Each area of the code is encapsulated in modules. The UI layer (`webui/app.py`) calls functions from the scripts and mediates between user options and processing. The core logic for each processing step resides in the scripts. Particularly `tagger.py` is more complex, as this is where communication with the LLM occurs and multi-stage error handling is implemented (including reload/reset of the model in case of problems, multiple attempts up to a `retry_count` limit). The include utilities handle data pre- and post-processing, while presets and config represent the customizable parts that you can modify without code changes (e.g., adding your own presets or entering a different model in `config.json`).

## Final Words

With the ACE-Step Data-Tool, you can create a richly annotated dataset from a collection of songs in just a few steps. The tool takes the tedious manual work of collecting lyrics and creating tags off your hands and standardizes the metadata format for further use in the ACE-Step project or similar endeavors. If you have questions or problems, please consult the Issues in this repository or contact the developers.

**Tip:** Detailed information about the tested system configuration (hardware, drivers, software versions) can be found in the file `System_Specifications_music-scraper.md`. This contains the exact environment information of the development computer and can serve as a reference if you need to debug environment problems. Good luck using the ACE-Step Data-Tool!