Report: Audio Signal Processing / Tune Scout

Softwaredesign (MECH-M-DUAL-1-SWD)

Master Mechatronik

1. Semester

Course Instructor: Peter Kandolf, PhD

Class: MA-MECH-24-BB

Author: Tobias Obwexer, BSc; Lennard Pöll, BSc

17. Februar 2025

Table of contents

List of illustrations

1 Introduction

This laboratory report presents the development and implementation of a web application for music upload, secure streaming, and audio equalization using Streamlit. The application integrates Amazon Web Services (AWS), specifically Amazon S3 for secure storage and Amazon DynamoDB for metadata management and duplicate detection via audio fingerprints.

The project demonstrates how cloud-based storage and database systems can be effectively utilized in a music streaming application while ensuring security, efficiency, and user-friendly interaction.

2 Objektives

The primary objective of this project is to develop a functional web application that allows users to seamlessly upload, manage, and stream music files. By integrating cloud-based technologies, the system aims to provide a scalable and secure solution for handling digital audio content.

Through the use of audio fingerprinting, the application ensures that identical songs are not uploaded multiple times, thereby optimizing storage usage and maintaining data integrity. In addition, the system should allow users to input metadata such as the title, artist, and album of each track, ensuring structured and organized data management.

Beyond file storage and duplication control, the project seeks to enable secure and controlled music streaming. To achieve this, pre-signed URLs generated via Amazon S3 will be used, ensuring that music files remain private and accessible only to authorized users.

Another significant goal of this project is to integrate an interactive audio equalizer. By allowing users to adjust various frequency ranges such as bass, mid, and treble. Furthermore, the system should enable users to save and apply custom equalizer settings for future use.

Lastly, the project aims to demonstrate the practical benefits of cloud computing in digital music management. By leveraging AWS services such as Amazon S3 and DynamoDB, the application is expected to showcase how modern cloud storage and database solutions can enhance efficiency, security, and usability in an online music platform.

3 Theoretical Background

Audio fingerprinting is an advanced technology used to identify audio files based on their unique acoustic features. This method is commonly utilized by applications such as Shazam to quickly and accurately recognize music tracks. The process of audio fingerprinting consists of several sequential steps, including spectrogram generation, peak identification, hashing of these peaks, and finally, matching against a database of known audio files. A more detailed implementation of these steps is discussed in Chapter 5.3.

3.1 SPECTROGRAM GENERATION

A spectrogram is a visual representation of the frequency components of an audio signal over time. It is created by applying the Fourier Transform to small segments of the audio signal. This technique allows for the analysis of how the frequency content of a signal evolves over time. In particular, the spectrogram helps highlight the key characteristics of a music track and makes them accessible for further processing. [1]

3.2 PEAK IDENTIFICATION

In a spectrogram, the most prominent points are those that exhibit the highest amplitudes within certain frequency ranges at a given time. These so-called peaks are particularly robust against background noise or distortions, making them stable reference points for fingerprinting. To identify these peaks, a maximum filter is often applied, highlighting local maxima in the spectrogram. This ensures that only the most significant frequency components are isolated. [1]

3.3 HASHING

Once the peaks in the spectrogram have been identified, the next step is hashing. This involves grouping the peaks into distinctive pairs, considering not only their frequency values but also their temporal spacing. This pairing increases the uniqueness of the fingerprints and reduces the probability of false detections. The generated hashes serve as compressed representations of the audio signal and are stored in a database. [1]

3.4 MATCHING

During the matching process, the hash set of an unknown audio signal is compared with a database of hashes from known audio files. By pairing peaks and considering time differences between them, the search can be significantly accelerated. If a match is found in the database, the corresponding music track can be identified with high precision. [1]

The detailed implementation of these techniques is further explained in Chapter 5.3.

4 Methodology

4.1 TECHNOLOGIES USED

The application is developed using Python and incorporates a combination of frontend, backend, and cloud technologies to ensure efficient audio management, streaming, and equalization.

Frontend: The user interface is built using Streamlit, a lightweight and interactive Python framework designed for web application development. It allows users to upload songs, adjust equalizer settings, and manage their music library effortlessly.

Cloud Services:

- **Amazon S3:** Securely stores audio files, ensuring scalable object storage and controlled access via pre-signed URLs.
- **Amazon DynamoDB:** A database used to store song metadata and fingerprints, enabling efficient deduplication and retrieval.

Backend: The backend utilizes several Python libraries to implement core functionalities:

- **boto3:** AWS SDK for Python, managing interactions with S3 and DynamoDB.
- **pandas and numpy:** Used for data manipulation and fingerprint generation.
- **Jinja2:** Facilitates dynamic template rendering if needed.

4.2 SYSTEM ARCHITECTURE

The system consists of three main modules: Song Upload and Metadata Handling, Secure Streaming, and Audio Equalization.

(A) Song Upload and Metadata Handling

- Users can upload MP3 or WAV files via the Streamlit interface.
- Metadata such as artist, title, and album can be manually entered or assigned default values.
- The system generates an audio fingerprint for each uploaded file, ensuring duplicates are detected before storage in DynamoDB.

(B) Secure Streaming

- All uploaded songs are listed within a user-friendly interface.
- Secure streaming is enabled through pre-signed URLs from AWS S3, preventing unauthorized access.

(C) Audio Equalization

- Users can enhance playback quality by adjusting bass, mid, and treble frequencies.
- Custom equalizer settings can be saved and applied to other tracks for a personalized listening experience.

4.3 FEATURES OVERVIEW

Song Upload:

- Upload MP3/WAV files with metadata input.
- Automatically generate fingerprints for deduplication.

Duplicate Detection:

- Prevent duplicate entries by checking stored fingerprints in DynamoDB.

Streaming:

- List all uploaded songs.
- Stream audio securely via AWS S3 pre-signed URLs.

Audio Equalizer:

- Adjust bass, mid, and treble frequencies.
- Save and apply custom equalizer settings.

This methodology ensures a scalable and secure approach to audio file management, leveraging cloud computing and efficient data processing techniques to provide a seamless user experience.

5 Implementation

5.1 APPLICATION DEVELOPMENT

The frontend is a Python-based web application implemented using the Streamlit framework. It supports features such as user authentication, audio file uploads, streaming, music recognition, and comparisons. The application connects to Amazon Web Services (AWS) components like DynamoDB and S3 for database and storage purposes and uses auxiliary modules and features for audio file processing.

5.1.1 Main Functionalities

The core functionalities of the application include:

- **User Authentication:** Handles login and sign-up processes using hashed passwords with the `bcrypt` library. Credentials are managed in a DynamoDB table using the `UserManager` class.
- **Song Uploading:** Allows users to upload MP3 or WAV files, input metadata, and store the songs in an S3 bucket. Fingerprints for audio files are generated and stored in DynamoDB tables.
- **Song Recognition:** Provides methods to compare uploaded or recorded audio files against the existing database using audio fingerprints.
- **Song Streaming:** Fetches and streams stored songs from the S3 bucket using a presigned URL.
- **Audio Equalizer:** Provides an interface for song equalization using the external `equalizer_features` module.

5.1.2 Class Description: `StreamlitApp`

The `StreamlitApp` class implements the features described above. Below is an explanation of its key methods:

5.1.3 `authenticate_user()`

This method handles user login by verifying the provided username and password against the encrypted credentials stored in DynamoDB. The `session_state` is used to persist login status for the user.

5.1.4 `sign_up_user()`

Facilitates user registration by hashing passwords using `bcrypt` and creating new user entries in the DynamoDB user table. Error handling ensures unique usernames.

5.1.5 `upload_song_with_metadata()`

This method enables users to upload audio files along with metadata (e.g., title, artist, album). The process includes:

- File and metadata input validation.
- Conversion of uploaded files (e.g., MP3) to WAV format for compatibility with the audio fingerprinting system.

- Fingerprinting of the uploaded file using the `fingerprint_file()` function.
- Database storage of the file metadata and fingerprints using DynamoDB tables (`SongsFingerprints` and `Hashes`).
- Uploading the audio file to an S3 bucket.

The method also checks for duplicates in the database using generated fingerprints and returns matching results if any.

5.1.6 `compare_uploaded_song()`

Allows users to upload an audio file to compare against the existing database. Key steps:

- Converts uploaded files to WAV format, if necessary.
- Fingerprints the file and compares the hashes against the database using the `find_song_by_hashes()` method.
- If a match is found, retrieves metadata of the matching song and displays it to the user; otherwise, informs the user of no matches.

5.1.7 `compare_recorded_song()`

This method uses the `record_audio()` function to capture audio directly from the user. The recorded audio is fingerprinted and compared with the database. The results (either a match or no match) are displayed similarly to the comparison of uploaded songs.

5.1.8 `stream_uploaded_song()`

Fetches songs dynamically from the database and provides an interface for users to stream these songs. Utilizes presigned URLs from S3 to enable secure playback within the Streamlit application.

5.1.9 Additional Utilities and Modules

The application integrates multiple external modules and libraries, as listed below:

- `pipeline.fingerprinting`: Provides methods for generating audio fingerprints from files and streams.
- `Databank.Amazon_DynamoDB` (`AmazonDBConnectivity`): Manages interaction with the DynamoDB tables.
- `Databank.Amazon_S3` (`S3Manager`): Handles file storage and retrieval operations for the S3 bucket.
- `pipeline.equalizer`: Contains the `equalizer_features()` function for audio file equalization.
- `pipeline.record`: A utility for recording audio directly from the user's microphone.
- `pipeline.audioconverter`: Provides the `convert_to_wav()` function for converting audio files to WAV format.

5.1.10 Application Workflow

The application is structured around the following main flow:

1. User authentication (handled via the login and signup methods).
2. Selection of the desired functionality (uploading, comparing, or streaming songs) through a sidebar menu.
3. Execution of the user-selected method (e.g., `upload_song_with_metadata()`, `compare_uploaded_song()`).
4. Display of results and feedback to the user.

5.1.11 Technology Stack

The `app.py` file is built using the following technologies and libraries:

- **Frontend:** Streamlit framework for creating an interactive web interface.
- **Backend:** Python modules for AWS services (e.g., `boto3`) and audio processing.
- **Database:** Amazon DynamoDB for metadata and fingerprint storage.
- **Storage:** Amazon S3 for storing and streaming audio files.
- **Audio Processing:** Custom-built fingerprinting, file conversion, and recording utilities.

5.2 DATABASE INTEGRATION

The project integrates AWS services, such as DynamoDB and S3, to manage song metadata, fingerprints, file storage, and retrieval. The integration is implemented through dedicated Python classes: `AmazonDBConnectivity` for DynamoDB operations and `S3Manager` for S3 interactions.

5.2.1 Amazon DynamoDB Integration

Amazon DynamoDB is a NoSQL database used to store the following:

- Song metadata, such as title, artist, album, and associated S3 storage information.
- Song fingerprints, which enable music recognition through hash-based comparisons.

The `AmazonDBConnectivity` class provides a streamlined abstraction for performing common operations on DynamoDB tables, such as inserting, fetching, and updating items. It supports custom features for managing song metadata and fingerprints.

5.2.2 Key Methods in `AmazonDBConnectivity`

- `__init__()`: **Initialization**
  Establishes a connection to DynamoDB using `boto3`. It initializes the high-level resource and low-level client, allowing flexible database operations.
- `insert_item(item)`: **Insert Items**
  Inserts a new item (e.g., song metadata or fingerprints) into the specified DynamoDB table.
- `fetch_item()`: **Fetch Items**
  Scans the DynamoDB table to retrieve all records, returning a list of items for further processing.

- `update_item(key, update_expression, ...)`: **Update Items**
  Updates an existing record in the table based on the primary key. Supports expression attributes for defining complex updates.
- `delete_item(key)`: **Delete Items**
  Deletes an item from the database using its primary key.
- `store_song(song_data, hashes)`: **Store Songs and Fingerprints**
  Combines metadata and fingerprints into a single workflow. It first checks if the song already exists (by comparing fingerprints) and assigns a new `SongID` if the song is unique. Metadata is stored in the `Songs` table and fingerprints in the `Hashes` table.
- `get_latest_song_id()`: **Retrieve Latest** `SongID`
  Fetches the maximum `SongID` from the `Songs` table to ensure unique IDs for all new song entries.
- `find_song_by_hashes(hashes)`: **Find Songs by Fingerprints**
  Searches the `Hashes` table to locate a song by comparing its hash values. Returns metadata of the matching song if found.

5.2.3 Amazon S3 Integration

Amazon S3 is used to store and retrieve the MP3 or WAV files uploaded by users. The `S3Manager` class provides functionalities to interact with the S3 bucket, including uploading files, downloading files, and generating pre-signed URLs for secure access.

5.2.4 Key Methods in `S3Manager`

- `__init__()`: **Initialization**
  Establishes an S3 client connection using `boto3`, configured with AWS credentials, region, and bucket name.
- `upload_file(file_name, object_name=None)`: **Upload Files**
  Uploads a local file to the specified S3 bucket. Optionally allows renaming the file in the bucket using the `object_name` parameter.
- `download_file(object_name, file_name=None)`: **Download Files**
  Downloads a file from the S3 bucket to the local file system. The file can be renamed locally using the `file_name` parameter.
- `get_presigned_url(s3_key, expiry=3600)`: **Generate Pre-Signed URLs**
  Generates a temporary URL to securely access a specific file in S3. The generated URL is valid for a configurable duration (`expiry`). This is primarily used for streaming songs directly from the S3 bucket.

5.2.5 Usage of Database Integration in the Application

Both the DynamoDB and S3 integrations are extensively used in the application workflow:

1. **Uploading Songs:** When a user uploads a song, a new record is created in DynamoDB with metadata and fingerprints. The song file is uploaded to S3 for storage.
2. **Fetching Songs:** The application retrieves song metadata from DynamoDB, which is displayed to users. S3 is used to stream the corresponding audio file.
3. **Checking for Duplicates:** Fingerprints allow the application to check for existing entries in the database before adding a new song. This prevents duplicates.
4. **Streaming Songs:** Pre-signed URLs enable secure playback of audio files stored in S3.

5.2.6 Error Handling

Both classes include mechanisms to handle AWS-specific errors. For example:

- **Credentials Issues:** Errors like `NoCredentialsError` and `PartialCredentialsError` are caught and logged.
- **Client Errors:** Exceptions such as `ClientError` are caught during operations, e.g., when fetching or updating data in DynamoDB or S3.

Proper error messages are logged or displayed, ensuring incomplete operations do not block the application workflow.

5.2.7 Technology Summary

The integration uses the following AWS services:

- **DynamoDB:** NoSQL database for storing and querying song metadata and fingerprints.
- **S3:** Scalable object storage for preserving and retrieving uploaded audio files.

The `boto3` library serves as the primary interface for AWS operations, enabling secure connections and streamlined interaction with AWS resources.

5.3 PIPELINE PROCESSING

The pipeline consists of multiple components responsible for processing audio files, extracting fingerprints, and ensuring efficient song recognition. The primary modules include:

5.3.1 Audio Conversion

Before fingerprinting, all audio files are converted into a standardized format using the `audioconverter.py` module. This ensures consistency in sample rate and mono-channel processing, improving compatibility across different input sources. The conversion is handled by FFmpeg through a subprocess call:

```python
def convert_to_wav(input_path, output_path):
    command = [
        "ffmpeg", "-y", "-i", input_path,
        "-ac", "1",
        "-ar", str(settings.SAMPLE_RATE),
        output_path
    ]
    result = subprocess.run(command, stdout=subprocess.PIPE, stderr=subprocess.
        PIPE)
    if result.returncode != 0:
        raise RuntimeError(f"ffmpeg error: {result.stderr.decode()}")
```

5.3.2 Fingerprinting

The `fingerprinting.py` module extracts audio fingerprints by computing spectrograms and identifying significant peaks. The steps include:

- Loading the audio file and ensuring the correct sample rate.
- Computing the spectrogram using the Short-Time Fourier Transform (STFT).
- Identifying prominent peaks using a maximum filter.

- Generating hashes by pairing frequency-time peak pairs.

```python
def fingerprint_file(filename):
    f, t, Sxx = extract_spectrogram(filename)
    peaks = find_spectrogram_peaks(Sxx)
    peak_points = convert_to_tf_pairs(peaks, t, f)
    hashes = generate_hashes(peak_points, filename)
    return hashes
```

5.3.3 Spectrogram Computation and Peak Detection

The spectrogram transforms an audio signal into a time-frequency representation, enabling peak extraction. Peak detection ensures that only the most prominent frequency components are used in fingerprinting, improving recognition accuracy. The peaks are determined using a maximum filter:

```python
def find_spectrogram_peaks(Sxx):
    data_max = maximum_filter(Sxx, size=settings.PEAK_BOX_SIZE, mode='constant',
        cval=0.0)
    peak_mask = (Sxx == data_max)
    y_peaks, x_peaks = peak_mask.nonzero()

    peak_values = Sxx[y_peaks, x_peaks]
    sorted_indices = peak_values.argsort()[::-1]
    peaks = [(y_peaks[idx], x_peaks[idx]) for idx in sorted_indices]

    total_area = Sxx.shape[0] * Sxx.shape[1]
    peak_limit = int((total_area / (settings.PEAK_BOX_SIZE ** 2)) * settings.
        POINT_EFFICIENCY)

    return peaks[:peak_limit]
```

5.3.4 Hash Generation

Hashes are generated by pairing detected peaks into frequency-time pairs. This ensures that the system can efficiently compare and match audio fingerprints, making song recognition robust against noise and distortions:

```python
def generate_hash(p1, p2):
    return hash((p1[0], p2[0], p2[1] - p1[1]))
```

5.3.5 Target Zone Computation

To enhance robustness, peaks are paired based on a defined target zone. This zone allows for frequency-time matching that accounts for variations in recording conditions:

```python
def compute_target_zone(anchor, points, width, height, offset):
    x_min = anchor[1] + offset
    x_max = x_min + width
    y_min = anchor[0] - (height * 0.5)
    y_max = y_min + height

    for point in points:
        if y_min <= point[0] <= y_max and x_min <= point[1] <= x_max:
            yield point
```

5.3.6 Live Audio Stream Processing

In addition to processing static audio files, the pipeline supports live fingerprinting for real-time audio streams. This enables applications such as live song recognition and monitoring:

```python
def fingerprint_audio_stream(frames):
    f, t, Sxx = compute_spectrogram(frames)
    peaks = find_spectrogram_peaks(Sxx)
    peak_points = convert_to_tf_pairs(peaks, t, f)
    hashes = generate_hashes(peak_points, "streamed_audio")

    return hashes
```

This pipeline ensures that audio files and live streams are processed efficiently, fingerprints are generated accurately, and music recognition remains fast and scalable.

5.4 AUDIO EQUALIZER

The audio equalizer module provides users with the ability to adjust bass, midrange, and treble frequencies in uploaded WAV audio files. This feature enhances the listening experience by allowing real-time frequency adjustments.

5.4.1 Equalizer Features

The equalizer is implemented in the `features.py` module, where users can upload a WAV file and apply different frequency adjustments using sliders. The signal modifications are calculated as follows:

```python
def equalizer_features():
    st.header("Equalizer")
    uploaded_file = st.file_uploader("Upload a WAV file", type=["wav"])
    if uploaded_file:
        bass_gain = st.slider("Bass Gain (dB)", -10, 10, 0)
        midrange_gain = st.slider("Midrange Gain (dB)", -10, 10, 0)
        treble_gain = st.slider("Treble Gain (dB)", -10, 10, 0)
```

The uploaded file is processed, and the equalizer gains are applied to the signal, modifying its amplitude:

```python
equalized_signal = (
    audio_signal * (1 + bass_gain / 100) +
    audio_signal * (1 + midrange_gain / 100) +
    audio_signal * (1 + treble_gain / 100)
)

equalized_signal = np.clip(equalized_signal, -32768, 32767)
```

A waveform visualization of the equalized signal is then displayed using Matplotlib:

```python
fig, ax = plt.subplots(figsize=(10, 4))
ax.plot(time, equalized_signal, label="Equalized Signal")
ax.set_xlabel("Time (s)")
ax.set_ylabel("Amplitude")
ax.legend(loc="upper right")
st.pyplot(fig)
```

5.4.2 Filtering Mechanism

The `filters.py` module implements frequency filtering using Butterworth filters to apply low-pass, high-pass, and band-pass filtering:

```python
def butter_lowpass_filter(data, cutoff, fs, order=5, gain=1.0):
    nyquist = 0.5 * fs
    normal_cutoff = cutoff / nyquist
    b, a = butter(order, normal_cutoff, btype='low', analog=False)
    filtered_data = lfilter(b, a, data)
    return filtered_data * gain
```

Similarly, a high-pass filter and band-pass filter are implemented:

```
def butter_highpass_filter(data, cutoff, fs, order=5, gain=1.0):
    nyquist = 0.5 * fs
    normal_cutoff = cutoff / nyquist
    b, a = butter(order, normal_cutoff, btype='high', analog=False)
    filtered_data = lfilter(b, a, data)
    return filtered_data * gain
```

For applications requiring selective frequency enhancement, a band-pass filter is useful. It allows frequencies within a specified range to pass while attenuating frequencies outside this range:

```
def butter_bandpass_filter(data, lowcut, highcut, fs, order=5, gain=1.0):
    nyquist = 0.5 * fs
    low = lowcut / nyquist
    high = highcut / nyquist
    b, a = butter(order, [low, high], btype='band', analog=False)
    filtered_data = lfilter(b, a, data)
    return filtered_data * gain
```

These filtering techniques enable the user to enhance specific frequency ranges while suppressing unwanted noise, making the equalization process more effective.

6 Future Work

ACKNOWLEDGEMENT OF CURRENT ISSUES

The application currently experiences performance bottlenecks due to the following reasons:

- **High data processing demand:** Significant delays are caused by frequent interactions between the application and Amazon DynamoDB.
- **Redundant processing:** Repetitive computation-intensive tasks such as fingerprint generation and matching lead to slower response times.
- **Inefficient queries:** DynamoDB queries and lookups are not optimized for the current workload, resulting in slow data retrieval.

The following set of improvements should be considered to address these issues and enhance application performance in the future.

FUTURE IMPROVEMENTS

1. **Optimize Interaction with Amazon DynamoDB**
   - Use *batch operations* like `BatchGetItem` and `BatchWriteItem` to reduce the number of requests made to DynamoDB.
   - Implement *caching mechanisms*, such as Amazon ElastiCache or in-memory frameworks like `functools.lru_cache`, to store frequently accessed data (e.g., fingerprints and metadata) and reduce round-trip queries.
   - Restructure DynamoDB tables by employing composite keys or secondary indexes to optimize query performance.
2. **Introduce Background Processing for Intensive Tasks**
   - Integrate a background job queue using tools like AWS Lambda, Amazon SQS, or Celery to perform tasks such as fingerprint generation and song matching asynchronously.
   - This will allow users to interact with the application while computationally heavy operations run in the background.
3. **Database Partitioning and Sharding**
   - Partition DynamoDB tables to handle increasing data more effectively and improve query efficiency.
   - Consider introducing sharding techniques to distribute fingerprint data across multiple smaller databases.
4. **Improve File Upload and Conversion**
   - Accelerate file format conversions by offloading them to AWS services such as AWS Lambda or preprocessing them locally during the upload process.
5. **Streamlined Fingerprint Matching**
   - Optimize fingerprint matching algorithms and consider breaking fingerprints into smaller chunks for quicker comparisons.
   - Use search libraries like Elasticsearch for high-performance fingerprint lookups.
6. **Dedicated Audio Processing Service**
   - Delegate audio processing tasks (e.g., fingerprint generation and audio matching) to a microservice that can scale independently of the main Streamlit application.

7. **Minimize File Transfers**
   - Use presigned S3 URLs for direct uploads/downloads to/from Amazon S3, avoiding the need for file data to pass through the server.
8. **Optimize Streaming Performance**
   - Pre-generate *HLS* or *Dash* streaming playlists for songs stored in S3, enabling smoother audio streaming with adaptive bitrates.
9. **Scalability and Cost Monitoring**
   - Use AWS Auto Scaling to adjust resource usage based on demand patterns.
   - Monitor costs using AWS Cost Explorer to optimize resource usage.
10. **Improved User Authentication Workflow**
    - Replace plaintext verification mechanisms with a token-based authentication system, such as JWT (JSON Web Token), for better scalability and security.
11. **Improve Error Handling and Logging**
    - Implement robust error-handling mechanisms for database queries, audio processing, and file uploads.
    - Leverage AWS CloudWatch for consistent monitoring and logging.
12. **Asynchronous Architecture**
    - Utilize Streamlit's asynchronous capabilities or integrate a framework like FastAPI for non-blocking tasks such as fingerprint generation and comparisons.
13. **Enhance User Experience**
    - Add progress bars or status indicators during long-running tasks like fingerprint generation or metadata uploads.
    - Allow users to perform other actions while background tasks are being processed.
14. **Test and Profile the Application**
    - Use profiling tools like `cProfile` or `LineProfiler` to identify slow-performing functions in the codebase.
    - Conduct regular load tests to ensure scalability as the userbase grows.
15. **Explore Alternate Database Models**
    - For fingerprint and metadata storage, consider graph databases (e.g., Neo4j) or time-series databases for faster query performance.
16. **Preprocess Metadata Aggregation**
    - Precompute and store aggregated metadata values (e.g., combinations of artist names, titles, and albums) to save computation time during user queries.

Literatur.

[1] Cameron MacLeod, “abracadabra: How does shazam work?” 19.02.2022. [Online]. Available: https://www.cameronmacleod.com/blog/how-does-shazam-work