AI-Driven Music Caption Generation using LLM

Project Members: Bhuvanendra Putchala, Sneha Sasanapuri (Team 84)

1. Introduction

In the recent times, the demand for multimedia processing has increased significantly due to a wide range of applications related to accessibility, media analysis, and content generation. Audio captioning, a major task in this regard, is defined as providing natural language descriptions to audio. The present project report describes how such an audio captioning system is designed using cloud-based AWS services with the aim of automating and scaling.

The system was designed to process audio files uploaded by the user on the web server, segment them into smaller chunks, and generate captions for each chunk using a pre-trained BART model. The project showcases the integration of cloud-native technologies and state-of-the-art machine learning models to address practical challenges in multimedia processing.

2. Goals

This project covers the design, implementation, and deployment of an AI-powered system for generating captions from music clips using Large Language Models (LLMs). The architecture leverages several AWS services, including S3, SageMaker, Lambda, and CloudWatch, in processing music clips and generate meaningful captions. The report describes the technical approach, tools employed, and deployment methodology to arrive at a scalable and efficient solution.

The primary objectives of the project were:

- Automate the caption generation workflow using event-driven triggers
- Ensure scalability to handle varying workloads
- Provide a modular architecture to facilitate future enhancements

3. Components

Hardware

Elastic Beanstalk (EBS):

Elastic Beanstalk was used to host our web application, providing an easy-to-use platform for deploying and managing the application environment. Its primary advantage is that it abstracts infrastructure management, allowing concentration on the code while AWS handles provisioning, load balancing, and scaling. However, EBS has its disadvantages, such as limited customization compared to manually configured infrastructure and higher costs for complex applications. We chose EBS because it offered An easy way to deploy and scale the web application, according to the project's needs of speed for agile development and scalability.

S3 Buckets

AWS S3 was used as the backbone for object storage in our project, handling uploaded music clips, .npy files, captions, and model artifacts. The benefits of S3 include very low bounds on storage, high duruability, and easy integration with other AWS services such as Lambda and SageMaker. However, it has its downsides, such as extra cost for frequent data retrieval and vulnerability to data breach. S3 was selected because it is highly scalable and secure, maintaining a variety of data formats in a reliably stored and accessible way for all project components.

Lambda Functions

AWS Lambda functions were employed for automating various tasks, including file processing, invoking the SageMaker processing job, and making API calls to OpenAI. The advantages of Lambda include serverless execution, automatic scaling, and cost-effectiveness. However, Lambda has limitations such as execution time limits (15 minutes per invocation), and layer capacity restriction to only 50 MB in compressed format. We chose Lambda for its light-weightiness and agility while integrating with other AWS services.

SageMaker Processing Job

The SageMaker Processing Job served as the hosting environment for our fine-tuned LLM-based caption generation model. Its advantages include native support for deploying and managing machine learning models at scale. It also integrates seamlessly with S3 and other AWS services. However, SageMaker can incur high costs for prolonged usage and requires expertise in configuring containerized environments. In our project, we seamlessly integrated this with Lambda functions, and the simplicity and efficiency of this integration made it an ideal choice for our needs.

CloudWatch

AWS CloudWatch was used for monitoring and logging all processes, providing valuable insights into system performance and aiding in debugging. Its advantages include real-time monitoring and detailed metrics. However, CloudWatch's detailed logging can cost for high-volume logs. We chose CloudWatch because it offered centralized monitoring and debugging capabilities, ensuring smooth operations throughout the project pipeline.

CodeBuild

CodeBuild was utilized for creating custom Docker images with all the necessary model dependencies. Its advantages include fully managed build environments and support for multiple programming languages. To ensure compatibility with SageMaker's amd64 architecture, which is not inherently supported when building in Docker (arm64), CodeBuild provides an excellent built-in mechanism to adapt and optimize the build process for SageMaker's architecture. We used CodeBuild because it simplified the process of creating reproducible environments for which our containerized applications could run seamlessly across a variety of AWS services.

ECR Registry

Amazon Elastic Container Registry (ECR) was utilized to store our project's custom Docker images. ECR's advantages include tight integration with AWS services, support for private repositories, and scalability. However, ECR may involve additional costs compared to other container registries, and this dependency on AWS services might be limiting when dealing with hybrid environments. We chose ECR for its smooth integration with CodeBuild, SageMaker, among other AWS tools for seamless container Management workflow.

Software

The software components of this project are critical to enabling the integration of machine learning, cloud services, and user interaction. Below is a comprehensive list of software tools utilized:

Web Application Framework

The web application framework provides the backbone for the project's interface through which users can upload.mp3 files with ease. Flask was chosen because it is a light, Python-based web framework that is simple, flexible, and supported by an extremely large community. Because of this, it perfectly integrates with AWS S3 for efficient uploading of files and provides robust file handling. Furthermore, Flask supports fast development and deployment, making it perfect for this project. However, Flask has some drawbacks in that it is less functional than heavier frameworks such as Django; it might take more effort when building a larger project. Regardless, Flask's scalability and focus on minimalism created the perfect combination for this development of a user-approachable platform that bridges between the user and the systems doing the processing in the backend.

Frontend tools

The user-facing webpage for uploading files and displaying captions was created using HTML, CSS, and JavaScript. These tools gave full control over the design of the interface, ensuring simplicity and accessibility for all users. HTML and CSS allowed for a clean layout, while JavaScript added interactivity, such as real-time status updates on the processing pipeline. We used standard web technologies to ensure the application remains lightweight and responsive, accessible across devices, and aligns with the scalability goals of the project.

Pre-trained model and Machine Learning Framework

The project utilized the BART (Bidirectional and Auto-Regressive Transformers) model for generating natural language captions from segmented audio. BART is a powerful pre-trained model that excels in tasks requiring contextual understanding and generation, making it ideal for audio captioning. *PyTorch* was selected as the machine learning framework due to its dynamic computation graph, ease of use, and libraries like Transformers for handling BART. Also, *NumPy* and *FFmpeg* were used for data manipulation and audio pre-processing respectively. While *PyTorch* offers great flexibility, its learning curve can be steep for beginners. Despite this, its extensive documentation and active community provided the necessary support. The combination of BART and *PyTorch* ensured a high-quality captioning pipeline, leveraging cutting-edge machine learning capabilities.

Audio Pre-processing Tools

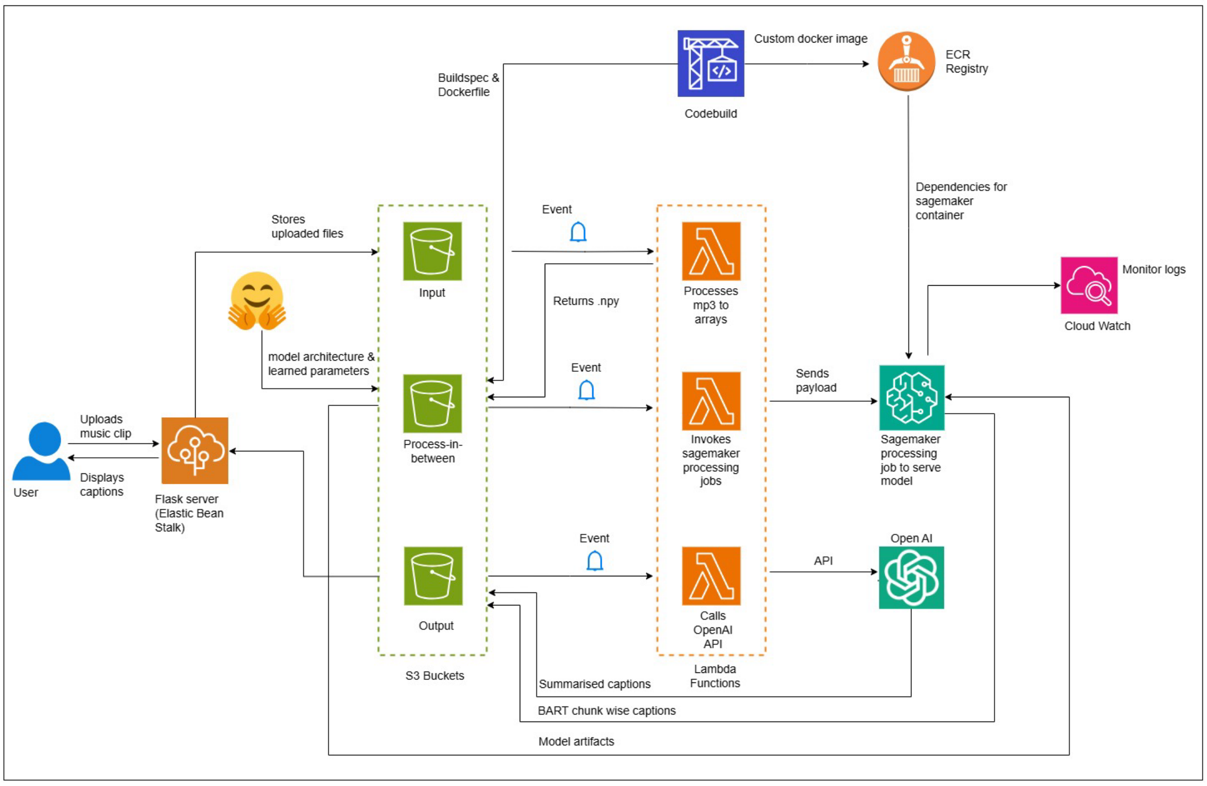
The audio pre-processing pipeline primarily relied on *FFmpeg* for converting .mp3 files into mel spectrograms. *FFmpeg* was chosen for its high-performance audio decoding capabilities, support for a wide range of formats, and efficient processing speed. It enabled us to handle audio data with precision, ensuring that it met the model's requirements. While *FFmpeg* requires command-line proficiency and lacks Python-native functionality, its robustness outweighed these drawbacks. *FFmpeg* was preferred for its faster processing speed and broader format compatibility. The use of *FFmpeg* streamlined the conversion process, providing high-quality inputs for the machine learning model.

API for Text Summarization

The OpenAI GPT API was used, which summarized the chunkwise captions into a coherent, concise description. This replaced the redundancy arising between two overlapping chunks of audio. These summaries were of high quality, well-structured, and contextual, ensuring the final captions were user-friendly. While this did increase the latency and the computational cost by relying on an external API, the advantages of auto summarization outweighed these flaws. Integration of OpenAI GPT greatly improved the project by adding a layer of coherence and polish to the generated captions.

4. Architecture

The architecture of the system is designed to provide an end-to-end solution for processing and captioning music clips. The architecture ensures seamless integration between the components and efficient orchestration of tasks.



5. Interaction of components (Workflow)

The system follows an automated pipeline comprising the following steps:

File Upload and Storage

A user initiates the workflow by uploading a *.mp3* file through a web interface hosted on EBS. The uploaded file is securely transferred to a designated Amazon S3 bucket for storage and further processing.

File Processing with Lambda Function

The S3 upload event triggers the first AWS Lambda function. This function processes the uploaded `.mp3` file by converting it into a *.npy* format, a NumPy-compatible array. The *.npy* file is then written to a second specified S3 bucket, ensuring seamless transfer for subsequent stages of processing.

Invocation of SageMaker Processing Job

The addition of the *.npy* file to the second S3 bucket triggers a second Lambda function. Acting as a response handler, this function facilitates communication with the Amazon SageMaker endpoint. It sends the *.npy* payload as a list to the SageMaker model for processing. The array is then converted into a tensor format suitable for inference.

Model Initialization and Caption Generation

Within the SageMaker environment, the container instance is initialized, and the necessary model artifacts—such as pre-trained weights and architecture—are retrieved from the S3 bucket. The inference script in the container is executed to load the BART model, process the audio data, and generate captions. Captions are generated chunk-wise to ensure temporal precision, capturing changes within specific time intervals. This approach minimizes the probability of quality loss and ensures comprehensive coverage of the audio content.

Output Storage and Summarization

The generated chunk-wise captions are written to an output S3 bucket. This event triggers a third Lambda function, which invokes the OpenAI API to produce an overall summary by consolidating all chunks. This summarization process effectively eliminates repetitive terms across chunks, leveraging large language models (LLMs) to provide a cohesive and concise representation of the audio content. Both chunk-wise captions and summarized captions are displayed on the user interface, allowing users to view a detailed breakdown as well as a high-level overview of the audio content.

6. Capabilities and Limitations

Capabilities

The project provides an automated pipeline for converting audio files into natural language captions. By integrating AWS Lambda functions with SageMaker, the workflow runs smoothly, processing each file without requiring manual intervention.

At the core of the project is the pre-trained BART model, a transformer-based architecture that excels in generating contextually accurate captions. With use of pre-trained weights, the system avoids the need for extensive training, making it highly effective in generating high-quality text descriptions. Moreover, the integration of OpenAI's GPT API enhances the output by summarizing chunk-wise captions into cohesive descriptions, providing for both detailed and consolidated insights.

The system allows chunk-wise captioning, enabling the generation of time-specific captions that improve the accuracy and relevance of the outputs. This design ensures that even complex or lengthy audio files are processed effectively. The user-facing web application further enhances accessibility by displaying both detailed and summarized captions, creating a user-friendly interface for diverse audiences.

Scalability is a key strength of the system, driven by the use of AWS services like S3, Lambda, and SageMaker. These services allow the project to handle increased workloads seamlessly, ensuring robust performance under high demand. Additionally, Elastic Beanstalk simplifies the deployment of the web interface, providing a stable hosting environment with minimal configuration overhead.

Limitations

The project, however, has some intrinsic limitations. First, by being dependent on the pre-trained BART model, the quality of the captions depends on the existing training data. While this works great for general-purpose captioning, it may be less relevant for niche audio domains and may require further fine-tuning. Another thing is that the project is computationally intensive, and the cloud costs from SageMaker processing jobs and OpenAI API calls add up, especially with large datasets.

Another limitation is the time required for processing larger audio files. As the audio file size increases, the model must handle more data chunks, which directly impacts the time taken to generate captions. This could delay the availability of results, especially in use cases involving extensive or complex audio files.

While AWS services provide scalability, the performance of SageMaker jobs is tied to the instance types used, which may limit processing speed for larger or more complex audio files. Lastly, the project is designed for asynchronous processing and does not currently support real-time captioning for live audio streams, limiting its applicability in dynamic use cases.

7. Debugging and Testing

Debugging

Debugging played a pivotal role in the successful development of our project. Throughout the implementation phase, logs generated from AWS services such as mainly SageMaker and Lambda served as the major tools for identifying and resolving issues. The detailed information for model deployment process included events such as like model loading, tensor transformations, and execution of the inference script. These logs have been very helpful to us to see the errors related to compatibility between input formats, memory allocation, and model inference logic. The Lambda logs were useful for diagnosing issues in the event-driven architecture, such as data mismanagement between S3 buckets, failed triggers, and incorrect formats.

So to make the cloud environment reliable , we conducted local testing of critical components like the process_audio.py python script. This scrip is responsible for converting audio files into a format compatible with the machine learning model, was tested on various audio inputs locally. Custom images with required dependencies were built using AWS Code Build and pushed to AWS ECR to simulate Sagemaker's in-built container instance. We created mock test cases to simulate various scenarios and validate the system's behaviour. These strategies provided a controlled environment to resolve dependency conflicts, handle circular imports, and address architecture mismatches effectively.

Testing

The Testing focused on validity of the system and accuracy of the results. We generated input .npy files from audio data of various lengths and qualities to make sure that the system could handle all sorts of scenarios. These .npy files were further processed, and the generated captions were evaluated based on coherence, relevance, and accuracy.

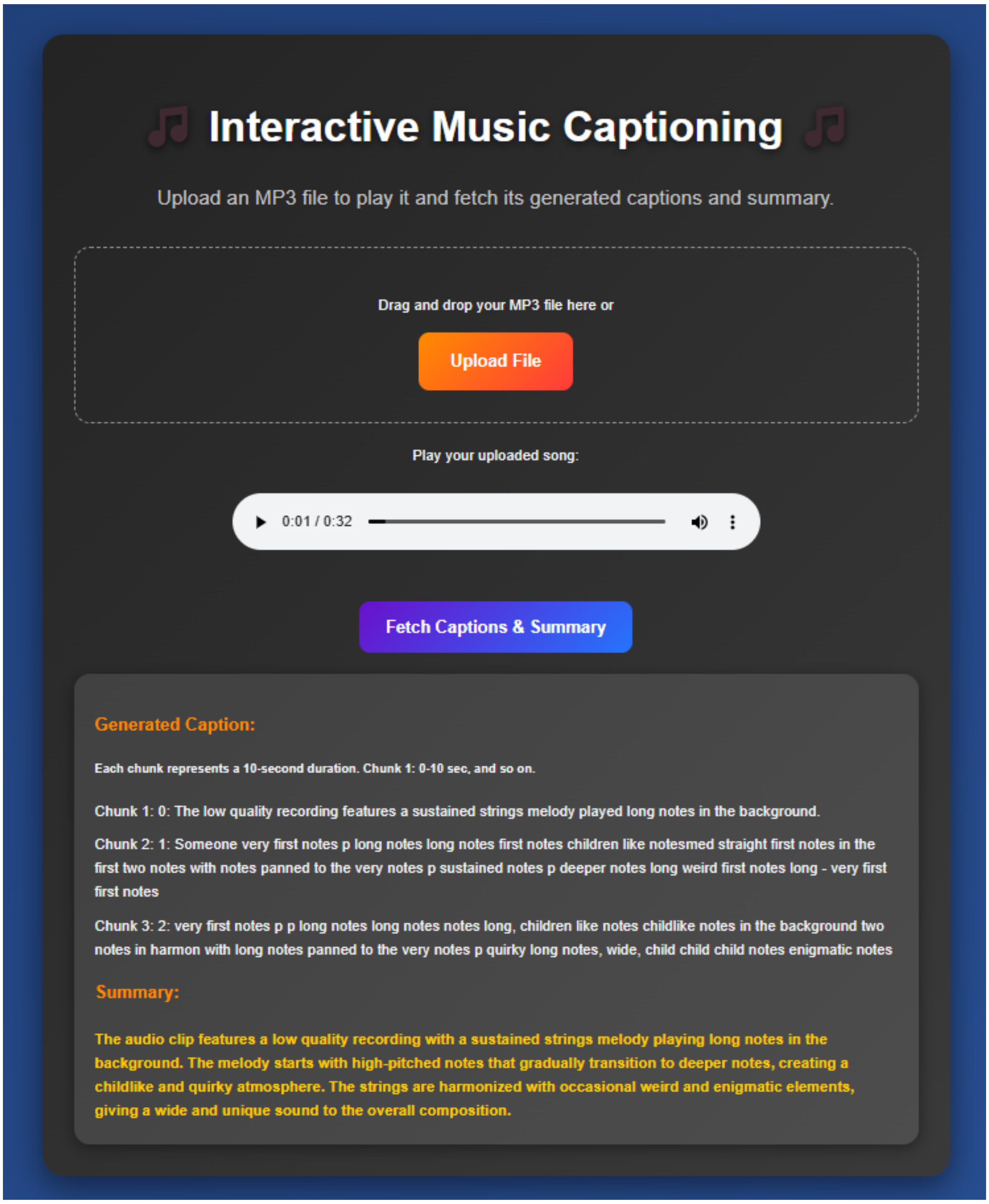
The system was tested for both functional correctness and performance under various conditions. Larger audio files were processed to evaluate the model's capability to segment and caption them accurately, although it was noted that longer files resulted in increased processing times. This limitation highlighted the need for efficient scaling solutions in future iterations. By employing a rigorous debugging and testing framework, we ensured that the system delivers high-quality captions while identifying areas for potential improvement.

8. Results

The system has handled several audio files and captioned them with high accuracy. Below are some example outputs:

Song Name: TunePocket-Sad-French-Accordion-Solo-30-Sec-Preview.mp3

Segment	**Generated Caption**
0–10s	The low quality recording features a sustained strings melody played long notes in the background
11–20s	Someone very first notes p long notes long notes first notes children like notesmed straight first notes in the first two notes with notes panned to the very notes p sustained notes p deeper notes long weird first notes long - very first first notes
21-30s	very first notes p p long notes long notes long, children like notes childlike notes in the background two notes in harmon with long notes panned to the very notes p quirky long notes, wide, child child child notes enigmatic notes


Interactive Music Captioning
Upload an MP3 file to play it and fetch its generated captions and summary.
Drag and drop your MP3 file here or
Upload File
Play your uploaded song:
0:01 / 0:32
Fetch Captions & Summary

Generated Caption:

Each chunk represents a 10-second duration. Chunk 1: 0-10 sec, and so on.

Chunk 1: 0: The low quality recording features a sustained strings melody played long notes in the background.

Chunk 2: 1: Someone very first notes p long notes long notes first notes children like notesmed straight first notes in the first two notes with notes panned to the very notes p sustained notes p deeper notes long weird first notes long - very first first notes

Chunk 3: 2: very first notes p p long notes long notes notes long, children like notes childlike notes in the background two notes in harmon with long notes panned to the very notes p quirky long notes, wide, child child child notes enigmatic notes

Summary:

The audio clip features a low quality recording with a sustained strings melody playing long notes in the background. The melody starts with high-pitched notes that gradually transition to deeper notes, creating a childlike and quirky atmosphere. The strings are harmonized with occasional weird and enigmatic elements, giving a wide and unique sound to the overall composition.

Conclusion

This project shows the potential of integrating cloud-native services with advanced machine-learning models for the processing of multimedia. It provides a scalable solution for audio captioning, with applications in accessibility, content indexing, and media analysis. Future work could focus on optimizing the model for real-time processing and extending support to multilingual captions.

References

1. Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., & Zettlemoyer, L. (2020). BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension.
2. Doh, S., Choi, K., Lee, J., & Nam, J. (2023). Automatic music captioning with LP-MusicCaps: Large Language Model based pseudo music caption dataset. Retrieved from https://huggingface.co/papers/2307.16372
3. Amazon Web Services Documentation. Available at: https://aws.amazon.com/documentation/