Multilingual Podcast Translation Service

Team Members:

- *Saketh Reddy Dodda*
- *Navya Chalamalasetty*

Responsibilities:

1. **Saketh Reddy Dodda**

    - Led the development of separate backend microservices, such as REST API, STT, Translation, TTS, Storage, and Redis Cache.
    - Responsible for the design and implementation of the system architecture, including Kubernetes cluster setup and ingress configurations.
    - Worked with backend services to integrate the frontend user interface.

2. **Navya Chalamalasetty**

    - Worked together to design the system architecture. Using Google Pub/Sub, we ensured seamless interaction between components using Google Pub/Sub.
    - Responsible for migrating the individual backend services into Kubernetes and leveraging ingress to seamlessly send the requests.
    - Contributed to the development of the frontend, auto-scaling and detailed testing between microservices to ensure reliability.

Project Goals:

The Multilingual Podcast Translation Service aims to offer podcasters a quick and automated way by translating and localizing their podcasts, so their content can be accessible to the global audience. In order to accomplish this, the service uses an end-to-end system that can process audio in any language, convert it to text, translate the text into the target language, and then return high-quality audio in the translated language. The project makes use of cutting-edge cloud-based technologies to guarantee user convenience, scalability, and dependability.

By implementing this system, the project has successfully achieved the following goals:

1. **Enhanced Accessibility:**

    - Made it possible for podcasters to communicate with a wide range of international audiences by overcoming language barriers.
    - To serve users of text-based content, transcripts in the target languages were made available to download.

2. **Automated Translation Workflow**:

    - Automated the complicated processes of speech-to-text transcription, language translation, and text-to-speech audio generation.
    - Podcasters saves time and effort by not having to re-record or perform manual translations.

3. **Scalability for Multilingual Content Distribution:**

    - Developed a microservices-based, cloud-native architecture with Kubernetes to effectively manage numerous requests.
    - Kubernetes' horizontal auto-scaling feature was implemented, allowing the system to scale dynamically in response to workload and guarantee steady performance even during times of high demand.
    - Ensured the system can adapt to changing demand and accommodate podcasters with different audiences and scales.

4. **Seamless Integration and User Experience:**

    - Rendered an interactive web-based frontend where users could choose languages, upload audio files, and monitor their progress.
    - Integrated caching mechanisms for repetitive requests, thereby reducing latency and enhancing system performance.

Potential Use Cases:

- **Global Audience Expansion:**
    - Podcasters can expand their audience to increase the reach to international listeners by delivering their content in multiple languages without requiring extra manual work.

- **Enhanced Accessibility:**
    - Translated audio and downloadable text transcripts make information more accessible to people with disabilities and those who prefer to read text.

- **Efficient Content Localization:**
    - Podcasters can easily adapt their content to different regions, allowing regionalized podcast episodes to enter the market more quickly.

Software and Hardware Components

1. **REST API (REST Server Service)**

    - **Purpose**: Serves as the gateway to backend services and handles user requests. It allows users to upload audio files, choose source and target languages and retrieve translated text and audio files.

- **Role**: Acts as bridge between the user interface and backend services, guaranteeing that the requests are handled efficiently and responses are accurate.

2. **Google APIs**

- **Google Speech-to-Text API:** Converts podcast audio into text in the source language.
- **Google Translation API:** Translates transcribed text into the target language.
- **Purpose**: Leverages Google's robust and accurate speech and text processing technologies to provide speech-to-text and text-to-speech capabilities.

3. **gTTS Python Library**

- **Purpose**: Used in the **TTS (Text-to-Speech)** Service to generate text-to-speech audio when performing translation tasks.
- **Role**: Offers text-to-speech processing that is both lightweight and adaptable for producing audio in target languages.

4. **Message Queues (Google Pub/Sub)**

- **Purpose**: Allows microservices to communicate with one another by managing data flow and queuing tasks sequentially.
- **Role in Workflow:**
  - The REST API publishes tasks to the first queue after receiving user input.
  - Each service ensures sequential and organized task processing by processing the data independently and publishing it to the next queue up to queue 6 until cache. It is implemented in a way this whole process follows the flow step-by-step in this order: STT → Translation → TTS → Big Query → Storage → Redis Cache.

5. **Key-Value Store (Redis Cache)**

- **Purpose**: Provides a caching layer for translations that already have been processed, allowing quick retrieval of results for frequently requested inputs.
- **Role**:
  - Reduces processing load on backend services.
  - Improves the system responsiveness by utilizing key-value pairs to cache translated text and audio files.

5. **Database (BigQuery)**

- **Purpose**: Tracks and retrieves metadata for every podcast translation request.
- **Data Stored:**
  - Metadata such as input/output file names, duration, source, output audio length, translated text length and target languages.
  - Status of each translation request is stored at every step in the pipeline.

- Additionally, file paths for transcriptions, translations, audio outputs, and original audio files will be stored.

6. **Google Cloud Storage (GCS)**

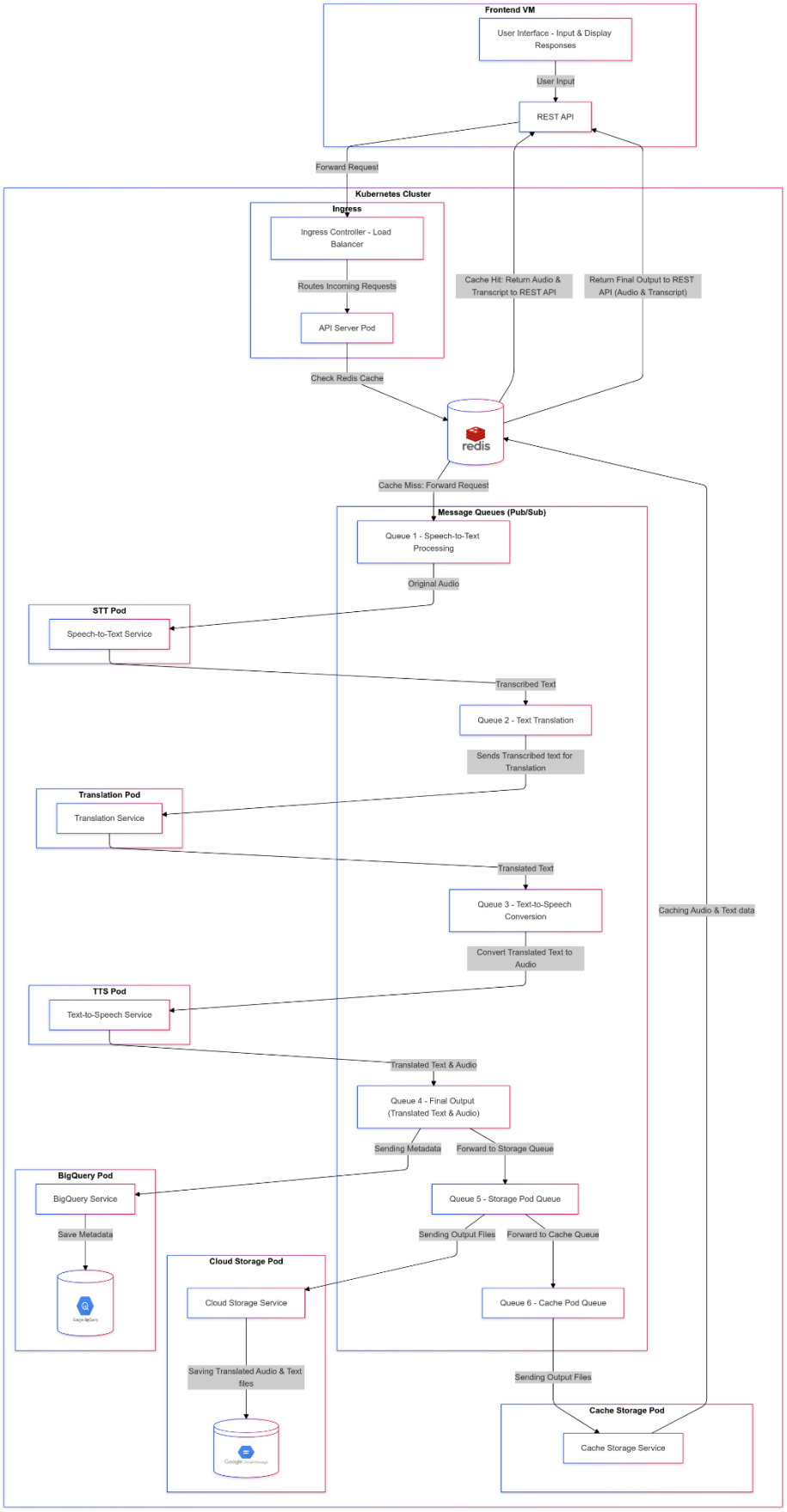
- **Purpose**: Provides scalable storage for large files, such as audio inputs and outputs, ensuring secure and accessible data storage.
- **Data Stored:**
  - Uploaded audio files.
  - Transcriptions from the Speech-to-Text service.
  - Translated text and audio outputs generated by the Text-to-Speech service.

7. **Kubernetes Cluster**
   - **Components:**
     - **Separate Pods for Microservices:**
       - Each pod handles a distinct core function (e.g., Speech-to-Text, Translation, Text-to-Speech, Storage, Cache) for isolated and efficient processing.
     - **Ingress Controller:**
       - Manages load balancing and route incoming API requests to the appropriate Kubernetes cluster services.
       - Ensures optimized workload distribution for a reliable and responsive system.
     - **Separate Pods for Storage Services:**
       - Dedicated pods manage interactions with Google Cloud Storage to store and retrieve processed data in an organized manner.
   - **Scalability:**
     - Horizontal auto-scaling dynamically adjusts the number of pod replicas depending on the workload, ensuring the system handles high demand efficiently.

8. **Google Cloud Platform (GCP)**

- **Virtual Machines:**
  - Used to host the frontend service, which allows user interaction outside of the Kubernetes cluster while maintaining isolation and allocating resources as efficiently as possible.
- **Kubernetes Engine (GKE):**
  - Machine Type: e2-standard-2
  - Runs the backend microservices and manages orchestration, scaling, and health checks.

Architectural Diagram:



Interaction of the Different Software and Hardware Components

The **Multilingual Podcast Translation Service** is an integrated system that seamlessly integrates several software and hardware components to deliver automated podcast translation services. Every component in the architecture plays a specific role and communicates with other components through well-defined channels. This section provides a detailed description of the connectivity between software and hardware components along with their interactions.

Frontend Interaction with Backend:

1. **User Interface**:

    - Hosted on a **separate virtual machine**, the frontend provides podcasters an easy-to-use interface for uploading audio files, choosing source and target languages, and interacting with the backend.
    - The frontend uses sends **HTTP requests** to send user inputs (audio files and language preferences) to the backend REST API.
    - The frontend also polls the backend for job status and upon completion, it displays the results (translated text and audio).

2. **REST API (REST Server)**:

    - The REST Server serves as the entry point for all user requests from the frontend.
    - It communicates with the Redis Cache to check for cached results and creates a distinct job ID for every request to track processing.
    - If a cache hit occurs, the REST API replies to the user directly after retrieving the cached information (translated text and audio) from Redis.
    - If a cache miss occurs, the REST publishes a message to **Queue 1 (Pub/Sub)** to forward the job to the backend pipeline.

3. **Ingress Controller:**

    - It acts as the main gateway for all incoming HTTP/HTTPS requests to the backend services hosted in the Kubernetes cluster.
    - Routes incoming requests from the **Frontend Virtual Machine (VM)** to the REST API based on predefined rules (/translate endpoint or /results/<job_id> endpoint).

Backend Microservices and Their Interactions:

1. **REST API to STT Service (Queue 1)**:

    - The REST API sends the audio file URL, source language, target language, and job ID as a message to **Queue 1** using Google Pub/Sub.
    - The **STT Service** (hosted in the STT Pod within the Kubernetes cluster) listens to Queue 1 for new messages.

- The STT Pod uses the **Google Speech-to-Text API** to process the audio file, converts it to text, and then sends the output to **Queue 2**.

2. **STT Service to Translation Service (Queue 2)**:

    - The **Translation Service** subscribes to **Queue 2** to get the metadata (e.g., job ID, source language, target language) and transcribed text.
    - The Translation Pod converts the transcribed text into the target language using the **Google Cloud Translate API** and publishes the translated text to **Queue 3**.

3. **Translation Service to TTS Service (Queue 3)**:

    - The **Text-to-Speech (TTS) Service** listens to **Queue 3** for translated text and metadata.
    - The TTS Pod produces high-quality audio in the target language, utilizing the **Google Text-to-Speech API.**
    - In order to store the metadata, the produced audio and metadata are sent to **Queue 4.**

4. **TTS Service to BigQuery Service (Queue 4)**:

    - The **BigQuery Service** subscribes to **Queue 4** to retrieve metadata about the processed job, such as input/output file names, audio duration, text length, and job ID.
    - The metadata is stored in **Google BigQuery** for structured data organization and future analysis.
    - Once metadata is stored, the BigQuery Pod forwards the message to **Queue 5** for file storage.

5. **BigQuery Service to Storage Service (Queue 5)**:

    - The **Storage Service** listens to **Queue 5** for metadata and signed URLs of the translated files.
    - Stores metadata for each processed job, such as input/output file names, text length, audio duration, and language preferences. Based on the job ID, it uploads the translated audio and text files to **Google Cloud Storage (GCS)** in an organized folder structure.
    - Once the files are stored, the service forwards a message to **Queue 6** for caching.

6. **Storage Service to Redis Cache Service (Queue 6)**:

    - The **Redis Cache Service** subscribes to **Queue 6** to store the encoded text and audio for quicker retrieval in the future.
    - The job ID is used as the key to store the encoded data.
    - The storage service uses signed URLs to communicate with the backend services, allowing for safe file downloads and uploads.

7. **Google Kubernetes Engine (GKE)**:

    - Hosts all backend microservices in a distributed environment, ensuring high availability and scalability.
    - The Kubernetes cluster is set up with **horizontal auto-scaling**, which helps it manage different workloads by dynamically adding or removing pods according to resource usage.

Debugging and Testing

We implemented a structured debugging and testing approach, covering local tests, unit tests, integration tests, error handling and end-to-end tests to ensure the reliability, accuracy and scalability which are stated below-

1. **Local Testing:**

    - **Backend Service Construction Without Cloud Components:** All backend services were developed and tested locally without utilizing cloud resources. Several methods were assessed for the text-to-speech (TTS) phase. Since some APIs were inefficient, we finalized using the gTTS Python library as a reliable alternative.
    - **Inter-Service Communication Using JSON:** Since it was not possible to implement a queue outside of the cloud environment at this time, a JSON-based structure was used to simulate message passing between microservices. This approach allowed us to validate the functionality and data flow between services locally.

2. **Unit Testing:** To verify the functionality of individual components in isolation.
    - **Components Tested:**
        - **REST API:** Tested endpoints for handling requests, sending data to the backend services, and delivering output in a format that can be downloaded.
        - **Microservices**: Prior to integration, each service was independently tested for its core functionality including accurate transcription, translation, and audio generation to ensure each of them worked as intended.
        - **Redis Cache:** Validated caching behaviour to ensure that frequently requested translations were stored and retrieved correctly.

3. **Asynchronous Processing with Job ID and Polling:**
    - To address asynchronous processing through Google Pub/Sub, we implemented a **job ID endpoint** (/results/<job_id>) in the REST server.
    - When a user submits a request, the REST server generates a unique job_id and returns it to the frontend. The JavaScript frontend uses this job_id to poll the /results/<job_id> endpoint regularly, checking if the output is available in the Redis Cache.
    - This mechanism allows the REST server to decouple from the backend services and handle user requests without waiting for backend responses, ensuring seamless asynchronous processing while maintaining real-time responsiveness for the user.

4. **Error Handling and Logging:** To ensure robustness by handling and logging errors at each stage of the workflow.

    - **Error Handling:**
        - Implemented try-catch blocks in each microservice to carefully handle exceptions (e.g., API failures, network issues).
        - Established fallback mechanisms to carry out tasks if a service failed temporarily.
    - **Logging**:
        - Collected detailed logs from every component, including user requests, data flow through message queues, API responses, cache retrievals, and errors encountered, using a centralized logging system such as Google Cloud. These logs offered an in-depth understanding of the workflow and assisted in quickly identifying and debugging the issues.

5. **Integration Testing:** To validate interactions between components and ensure seamless data flow throughout the pipeline.

    - **Message Queue Integration**: Verified whether the Google Pub/Sub queues transferred the data sequentially between each microservice without any delays or information loss.
    - **Kubernetes Pod Communication:** Tested to see if the data communication between each service's pods was received correctly and responded within the expected timeframes.
    - **Storage and Cache Interaction:** Verified that data was appropriately stored, cached, and retrieved by examining the interactions between the microservices, Redis Cache, and Google Cloud Storage.

6. **End-to-End Testing:** To validate the entire workflow from user input to the final output.
    - Simulated the entire request cycle, from user uploading an audio file, chosing the source and target languages, and then receiving the translated text and audio output in a downloadable format.
    - Validated that the functionality of each processing step (transcription, translation, and text-to-speech conversion) is as expected and making sure that the final output fulfilled the quality and accuracy standards.

Working System: Capabilities, Workload Handling, and Potential Bottlenecks

Working System:

The **Multilingual Podcast Translation Service** is designed as a reliable, step-by-step pipeline that effectively automates the podcast audio transcription, translation, and localization. Below is a detailed explanation of how the system works:

1. **Frontend:**
    - Users interact with the system via a web-based frontend where the users can upload an audio file, select the source and target languages, and start the translation process.
    - After the user submits the request, the frontend sends the audio file and metadata for processing to the backend REST API (/translate endpoint) via an ingress controller.

2. **Backend Workflow:**
    - **REST API:**
        - The REST API receives user requests, it created a distinct job ID for the task, and uploads the audio file to **Google Cloud Storage (GCS)**.
        - The API publishes the job metadata, including the GCS URL, source language, target language, and job ID, to **Queue 1 (Pub/Sub)** for the later processing steps.
    - **Speech-to-Text (STT) Service**:
        - Using **Google Speech-to-Text API,** the STT service retrieves messages from Queue 1 and converts the audio file into text.
        - **Queue 2** then receives the transcribed text and the job metadata for translation.
    - **Translation Service**:
        - The Translation service uses **Google Cloud Translate API** to extract messages from Queue 2 and converts the transcribed text into the target language.
        - After translation, the translated text is sent to **Queue 3** for text-to-speech processing.
    - **Text-to-Speech (TTS) Service:**
        - Using **Google Text-to-Speech API**, the TTS service extracts messages from Queue 3 and transforms the translated text into high-quality audio.
        - The service uploads the audio and text files that have been translated to **GCS** and posts their URLs to **Queue 4** along with the updated metadata.
    - **BigQuery Service:**
        - The BigQuery service pulls messages from Queue 4 and saves metadata for tracking and analysis, including file details, languages, and job IDs.
        - The service publishes the message to **Queue 5** for storage processing.
    - **Storage Service:**
        - The Storage service downloads the translated files from signed URLs, organizes them into GCS folders, and republishes the metadata to **Queue 6** for caching.

- **Redis Cache Service:**
    - The Redis Cache service retrieves the final metadata from Queue 6, encodes the translated text and audio files into Base64 format, and caches them in **Redis** for quick retrieval.

3. **Frontend Result Retrieval:**
    - The frontend used the job ID to poll the REST API (/results/<job_id> endpoint) for the results. If the results are available in Redis Cache, they are returned to the user directly. If not, the user is notified that the results are still being processed.

Workload Handling:

The system is built to effectively manage concurrent and high-volume workload. Key performance metrics include:

1. **Concurrent Requests**:
    - The system is able to process up to **10 concurrent translation requests** because of the asynchronous design and **horizontal auto-scaling** of Kubernetes pods.
2. **Supported File Size**:
    - The majority of podcast formats are compatible with the system because it supports audio files up to **100MB** in size.
3. **Processing Latency**:
    - The average time taken for a request to process from end-to-end processing is approximately **5-10 minutes (**depending on file size and complexity), including transcription, translation, and audio generation.
4. **Scalability**:
    - Kubernetes horizontal auto-scaling adds more pod replicas for resource-intensive services like STT and TTS, allowing to dynamically adapt to increased workloads.
5. **Languages and Dialects Supported**:
    - The system supports **over 14 languages** and is capable of handling their respective dialects for transcription, translation, and audio generation, by utilizing the flexible configuration of Google APIs.

Potential Bottlenecks:

Despite of the system's efficiency, there are few areas that might act as bottlenecks under specific conditions. Below are the bottlenecks identified and their potential solutions:

1. **API Rate Limits**:
    - **Bottleneck**: Dependency on Google APIs (Speech-to-Text, Translate, and Text-to-Speech) increase the possibility of exceeding API rate limits during peak workloads.
    - **Solution**: Implement a retry mechanism with exponential backoff for transient errors and actively monitor API quotas using Google Cloud Monitoring.
2. **Backend Processing Delays**:
    - **Bottleneck**: Large files or complex audio inputs may cause the processing pipeline to slow down, especially during resource-intensive stages like STT and TTS.
    - **Solution**: Optimization to resource allocation for Kubernetes pods can be done by increasing memory and CPU for these services and fine-tuning auto-scaling thresholds.
3. **Cache Evictions**:
    - **Bottleneck**: Excessive memory usage in Redis Cache may cause the frequently requested results to be evicted.
    - **Solution**: One possible solution to prevent data loss is to either scale Redis horizontally or integrate a persistent storage layer.
4. **Dialect Handling**:
    - **Bottleneck**: Translation quality and TTS audio generation may differ for less commonly supported dialects, resulting in inconsistent results.
    - **Solution**: Use Google API's dialect-specific configurations where possible and extend support by training custom models for niche dialects.

Future Scope

The **Multilingual Podcast Translation Service** has significant room for growth and optimization. Future enhancements include real-time language translation for live podcasts by integrating streaming APIs, providing compatibility for additional file formats such as WAV and FLAC to reach a wider user base, and better support for low-resource languages through custom language models and datasets. Designing an analytics dashboard might help podcasters gain insights into audience preferences, user engagement and translation metrics. Furthermore, batch processing capabilities would improve the user experience by streamlining the translation of multiple podcast episodes simultaneously. These improvements will increase the system's adaptability, scalability, and user-friendliness while meeting a greater variety of user requirements.