Project 3

AudioMitra

Submitted By
TEAM_11
Vilal, Hanuma, Shriom, Madan

20/04/2024

An Innovative Content Transformation Platform

1. Introduction

AudioMitra is a Content Transformation Platform to provide authentic and trustworthy transformations using the computational power of artificial intelligence and cognitive power of human in the loop.

2. Problem Statement

Audiomitra stands as an innovative solution within the realm of language processing, aiming to streamline human endeavors through technological advancement. Functioning as a content transformation system, Audiomitra facilitates the transition from digital images into text through Optical Character Recognition (OCR), and then converting text into audio using a text-to-speech (TTS) system.

3. Abstract Overview of the Proposed Solution

3.1. Text Extraction (OCR):

1. **Open-Source API Utilization:** Extract text content from images using the Open-Source OCR APIs.
2. **Flexibility:** Adaptable to other open-source OCR APIs or tesseract engine.
3. **Output:** Text files, with optional reassembly to match original document layout.

3.2. Content Validation:

1. **Text Accuracy:** Validate OCR'd content for text accuracy.
2. **Human Intervention:** Optionally allow for text editing and refinement.
3. **Output:** Validated text content.

3.3. Text to Speech/Audio (TTS):

1. **Open-Source API Utilization**: Convert extracted/validated content into speech/audio.
2. **Flexibility:** Adaptable to other open-source TTS APIs.
3. **Output:** Ready Audio file for play.

3.4. Text to Speech/Audio (TTS):

1. **Accessibility:** Accessible via a web browser, with username and password protection.
2. **User Roles:** Support for various roles, including validators, authors, and voice-over artists.
3. **Collaboration:** Facilitates collaboration and specialization within the platform.

4. Requirements for the System

4.1. Functional Requirements:

1. **FR1: Image to Text conversion:** Convert Images into Text using OCR.
2. **FR2: Content validation and editing:** Validate and edit text content for accuracy.
3. **FR3: Text to audio conversion:** Convert text content into audio formats using TTS.
4. **FR4: Multilingual audio support:** Support multiple languages in audio conversion.
5. **FR5: User management:** Support multi-user access with different roles.
6. **FR6: Authentication and authorization:** Secure user access to the system.
7. **FR7: Human in the Loop:** Enable collaboration between human and machine intervention.
8. **FR8: User-friendly interface:** Provide a web interface for accessing the system.
9. **FR9: API integration:** Integrate with open-source OCR and TTS APIs.
10. **FR10: Flexibility with APIs:** Adapt to different OCR and TTS APIs.

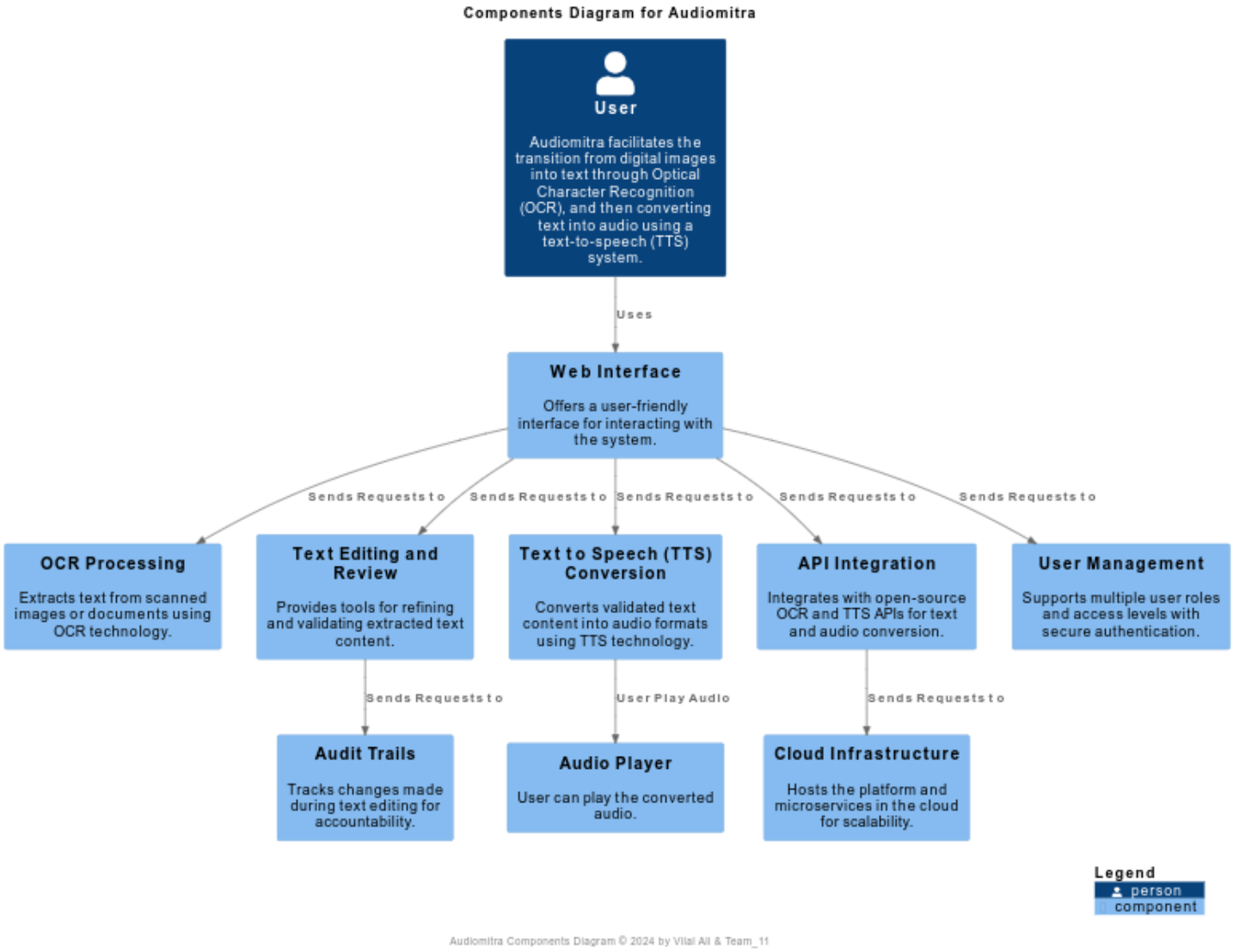
4.2. Non-Functional Requirements:

1. **NFR1: Performance and scalability:** Handle large volumes of data efficiently.
2. **NFR2: High availability:** Ensure minimal downtime and fault tolerance.
3. **NFR3: Reliability:** Provide consistent and dependable service.
4. **NFR4: Security:** Use encryption and access controls for user data.
5. **NFR5: Comprehensive documentation:** Provide user guides and help materials.
6. **NFR6: Standardized APIs:** Use APIs for integration with third-party applications.
7. **NFR7: Modular architecture:** Facilitate the addition of new features.

5. Audiomitra System Stakeholders

6. Concerns

7. Components Diagram



8. Architectural Design Decisions

These architectural design decisions ensure that Audiomitra is robust, scalable, maintainable, and efficient. The combination of modern frontend technologies like ReactJS and Tailwind CSS, a lightweight and flexible backend with Python Flask, and reliable data storage with MySQL provides a strong foundation for the system's architecture.

8.1. Use of React for Frontend Development

- **Decision:** Utilize Tailwind CSS for frontend styling.
- **Justification:** Tailwind CSS offers a utility-first approach that accelerates the development process and allows for highly customizable designs without the overhead of managing dense style sheets. It supports responsive design natively, which is crucial for modern web applications.

8.2. Integration of Tailwind CSS for Styling

- **Decision:** Design robust and well-documented APIs for seamless integration with external systems, such as weather forecast providers, parking lot sensors, and ticketing platforms.
- **Justification:** Well-designed APIs enable interoperability, allowing the system to leverage external services and data sources effectively. This enhances functionality, accuracy, and reliability, enabling features like weather forecast integration, parking lot availability updates, and ticket booking.

8.3. Python Flask for backend development

- **Decision:** Utilize Python Flask for backend development.
- **Justification:** Flask is a lightweight and flexible web framework ideal for rapid backend development. It provides strong support for building RESTful APIs and integrates seamlessly with a microservices architecture, allowing for scalability and easy maintenance.

8.4. MySQL for database storage

- **Decision:** Utilize MySQL for database storage.
- **Justification:** MySQL offers reliable relational data storage for structured data such as user information and relationships. It provides strong transaction support, security features, and optimization options for performance and scalability.

8.5. Adoption of Containerization and Orchestration Technologies

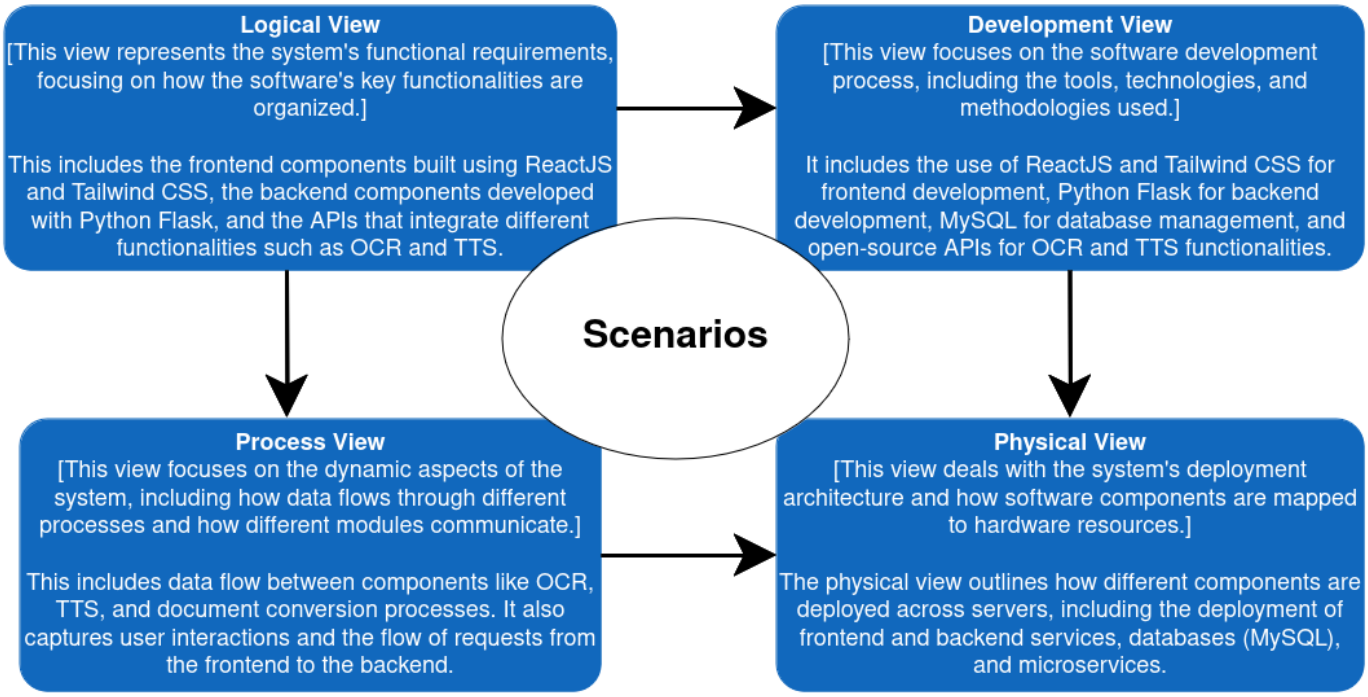
- **Decision:** Deploy both frontend and backend using Docker, with Kubernetes for orchestration.
- **Justification:** Containerization encapsulates the application's environment, ensuring consistency across different deployment stages and platforms. Kubernetes provides automation in scaling, management, and deployment of containerized applications, which is essential for maintaining high availability and facilitating microservices architecture.

8.6. Choice of Microservices Architecture

- **Decision:** Implement the backend using a microservices architecture.
- **Justification:** This architecture supports scalability and flexibility, allowing independent development and deployment of services, easier scaling, and effective load balancing. It facilitates the integration of diverse technologies tailored to specific services (e.g., Python for data processing and Node.js for API management).

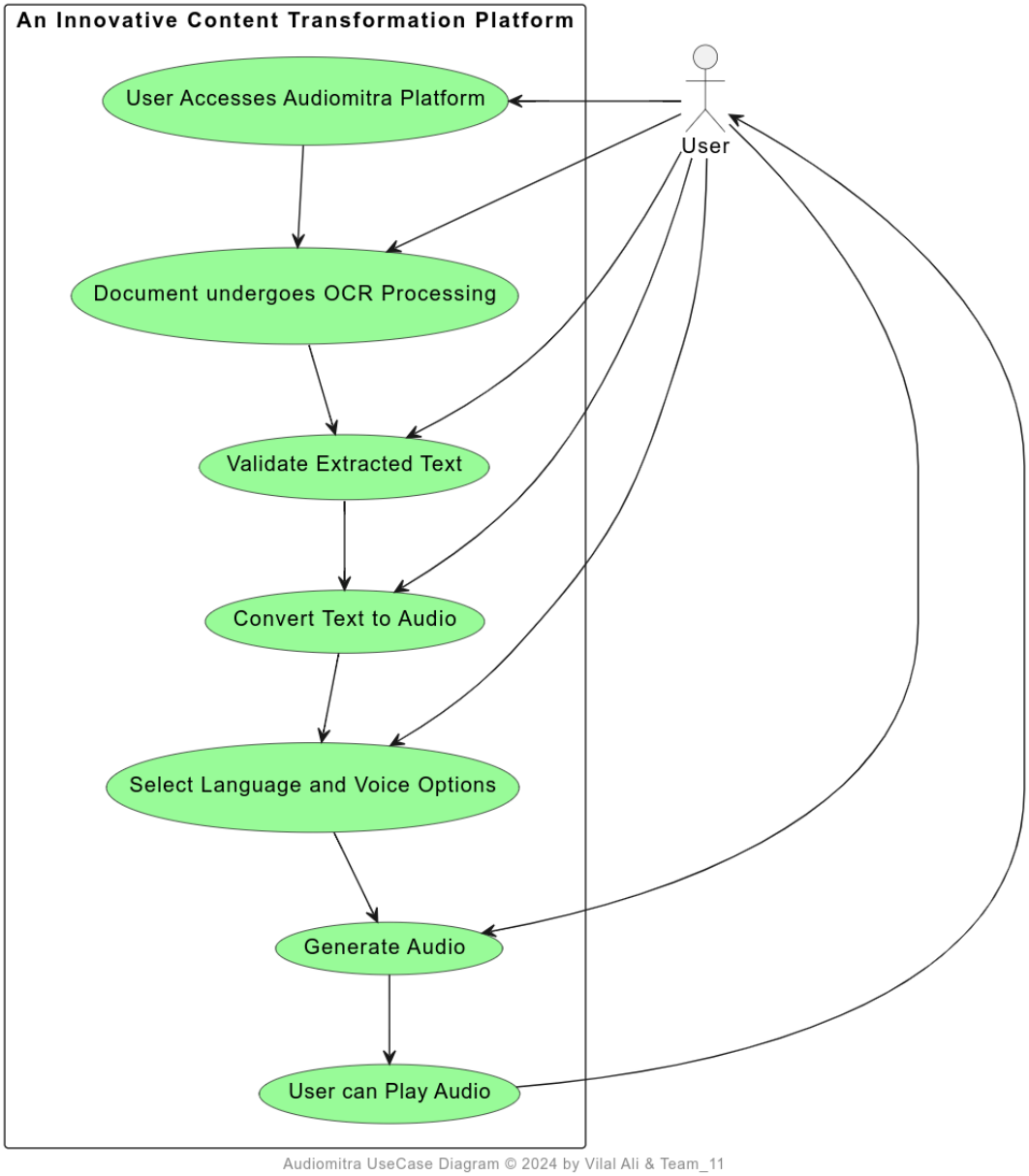
9. Views

The 4+1 architectural view model provides a comprehensive approach to understanding the structure and behavior of a software system from multiple perspectives. The four views represent different aspects of the system (logical, process, physical, and development views), and the "plus one" refers to the scenarios or use cases that tie the other views together. Here's how you might apply the 4+1 model to your project:



10. Scenario and Use Case

Description: This view ties the other four views together and represents the different use cases or scenarios the system supports.
Examples: Scenarios could include converting a printed document into an audiobook, digitizing a library's archive, or providing multilingual support for educational content.
Interactions: This view illustrates how the system supports user interactions across different scenarios, integrating various components and flows outlined in the other views.

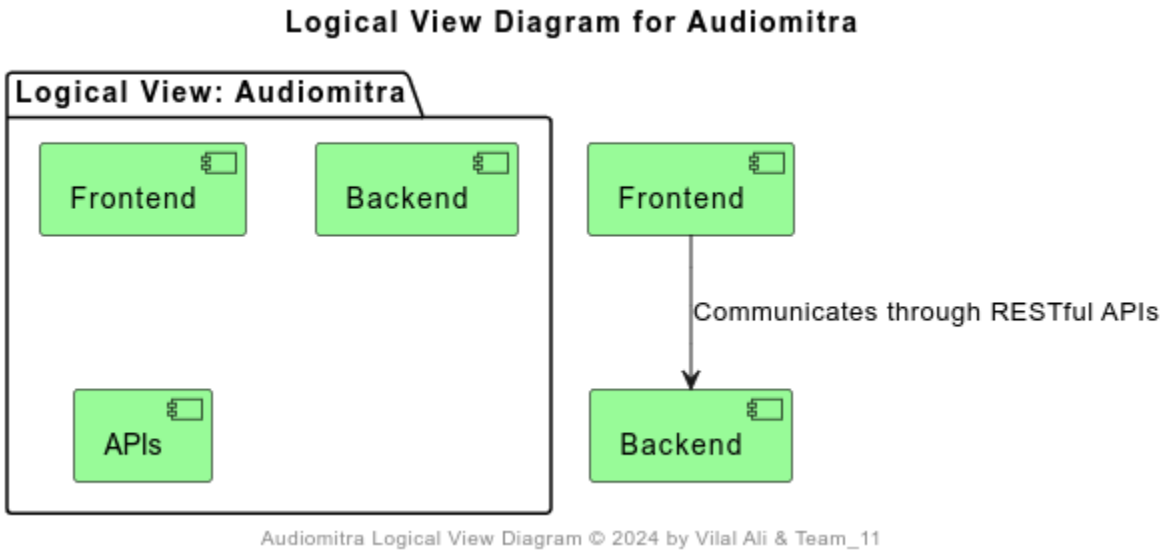


- **User Interaction:**
  - Users access the Audiomitra platform through the web interface.
  - They initiate the conversion process by uploading a scanned image or PDF of a printed document.
- **Document Processing:**
  - The uploaded document undergoes Optical Character Recognition (OCR) processing, extracting text from the scanned images.
  - The extracted text is validated for accuracy, with the option for the user to review and edit if necessary.
- **Text to Audio Conversion:**

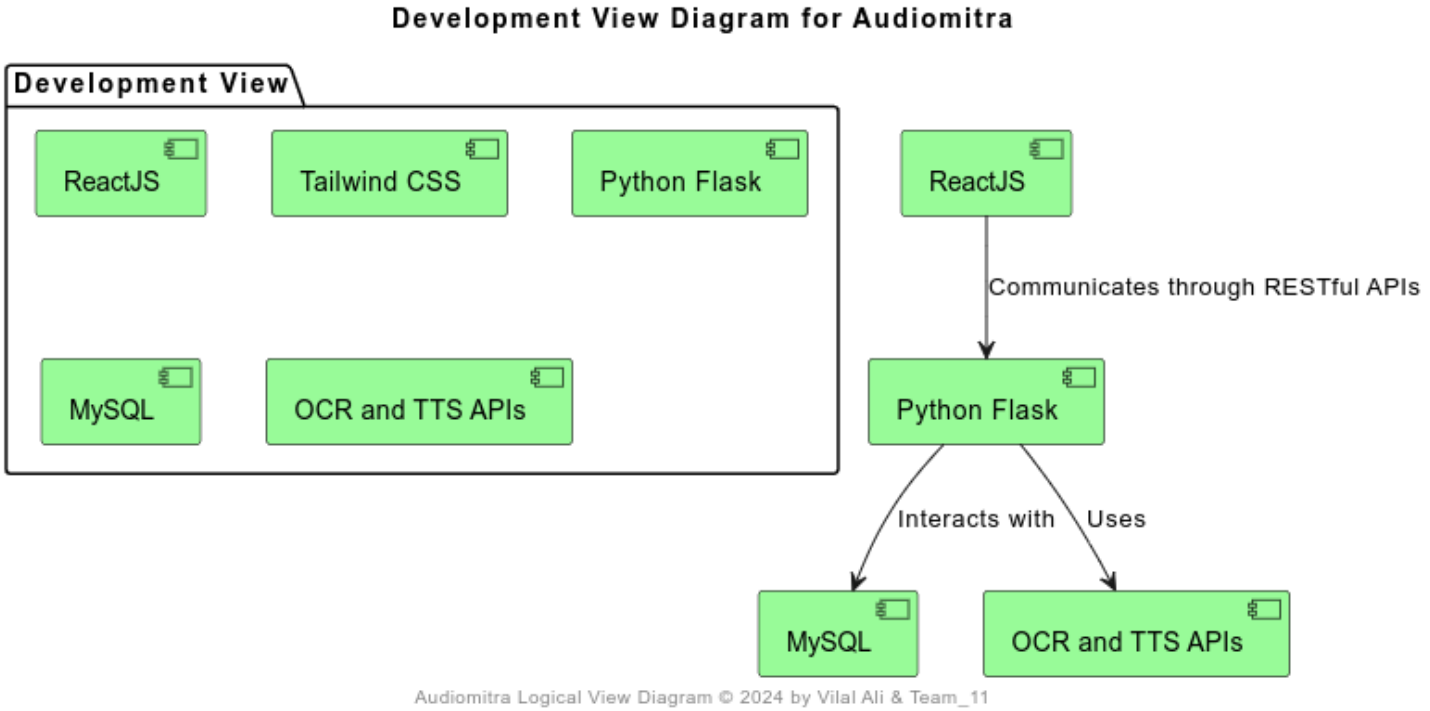
- Validated text content is then converted into an audio format using Text-to-Speech (TTS) technology.
- The user selects their preferred language and voice options for the audiobook.

- **Audio Output:**
  - The converted audiobook is generated and made available for download or streaming.
  - The user can listen to the audiobook on their preferred device, enhancing accessibility and convenience.

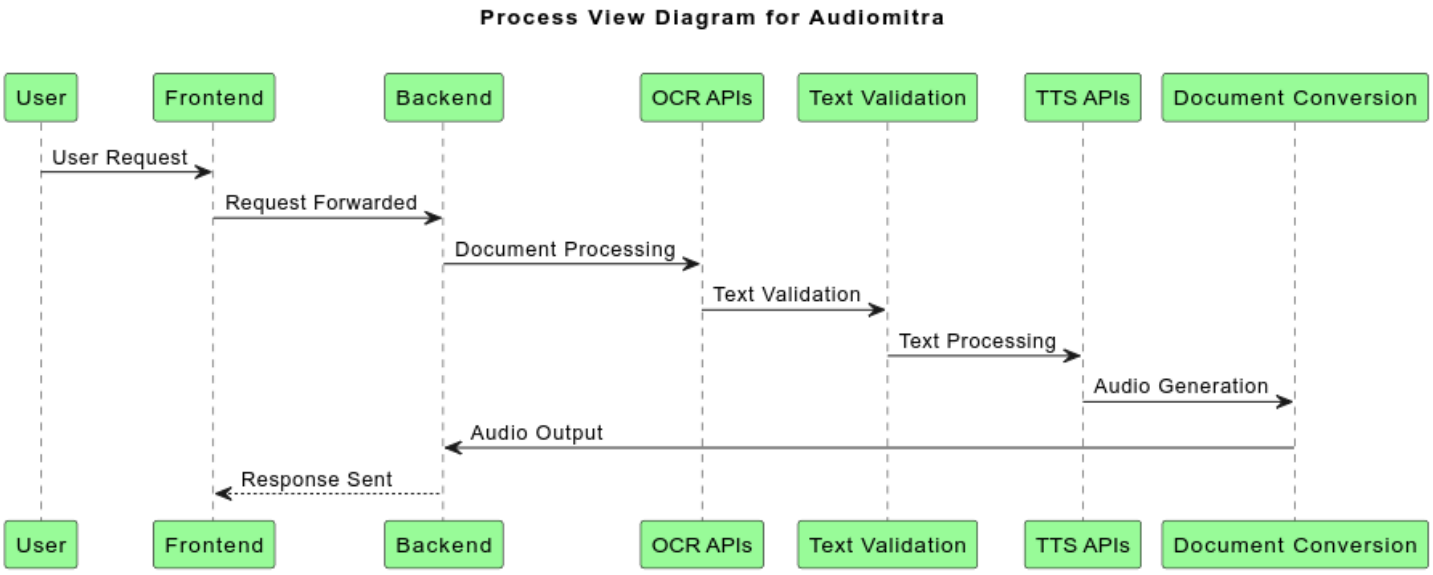
11. Logical View



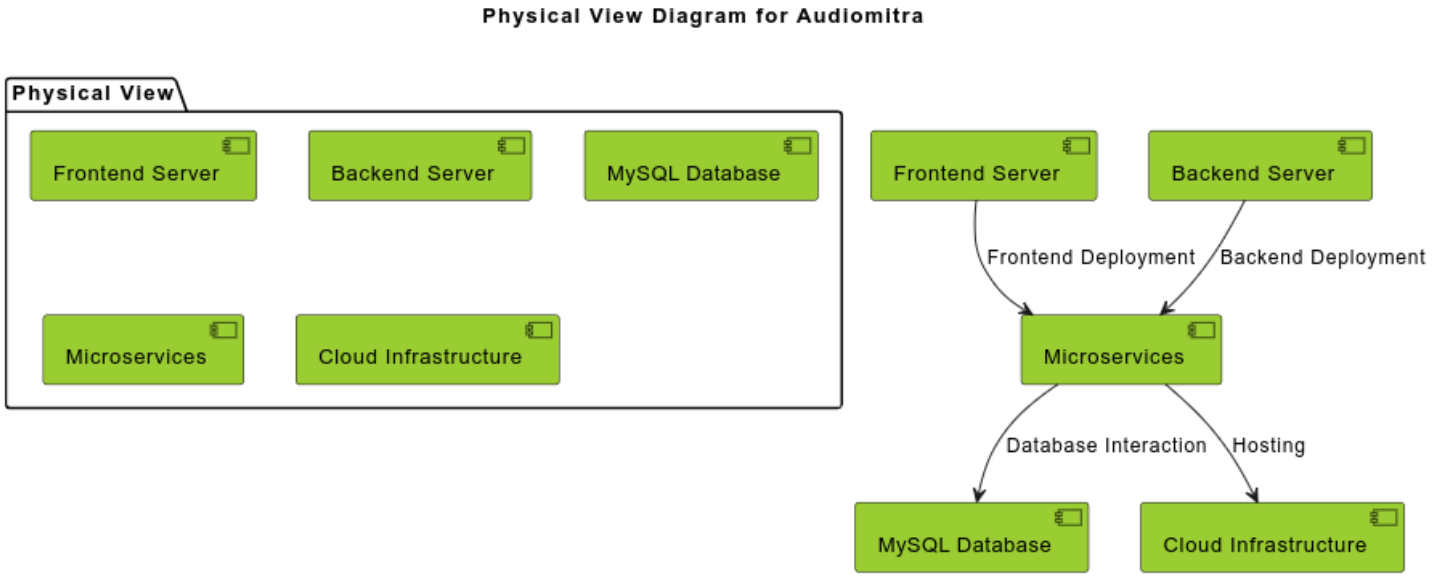
12. Development View



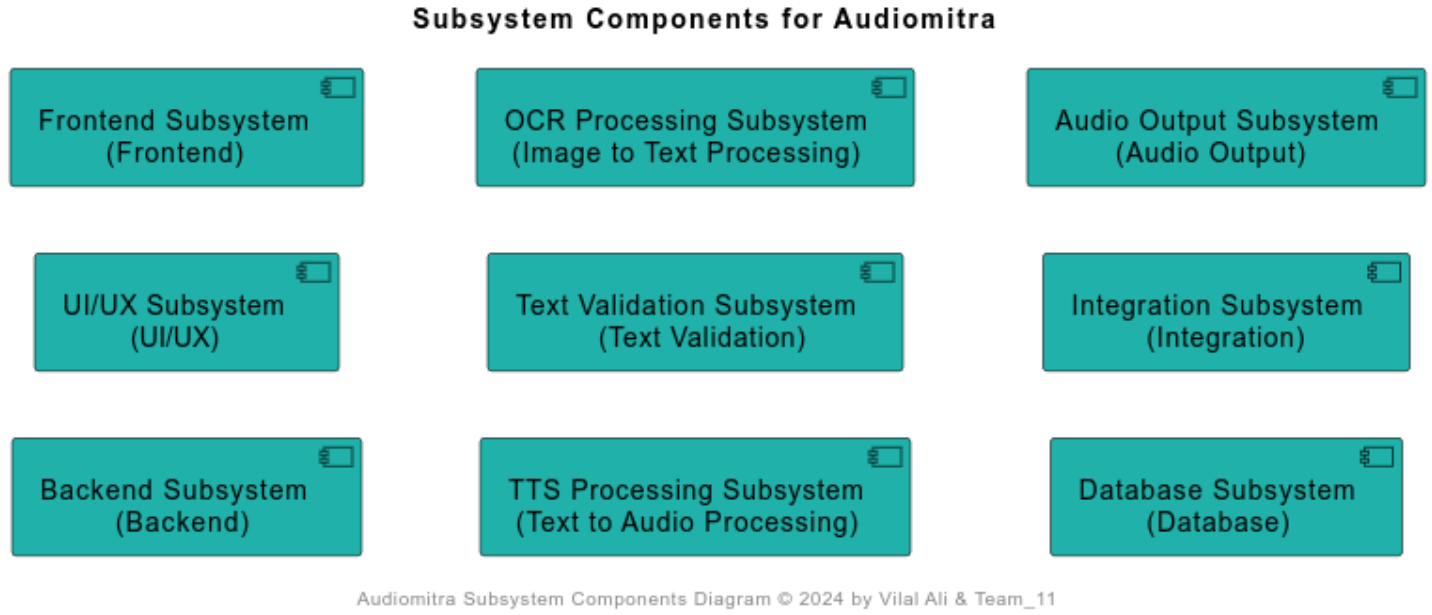
13. Process View



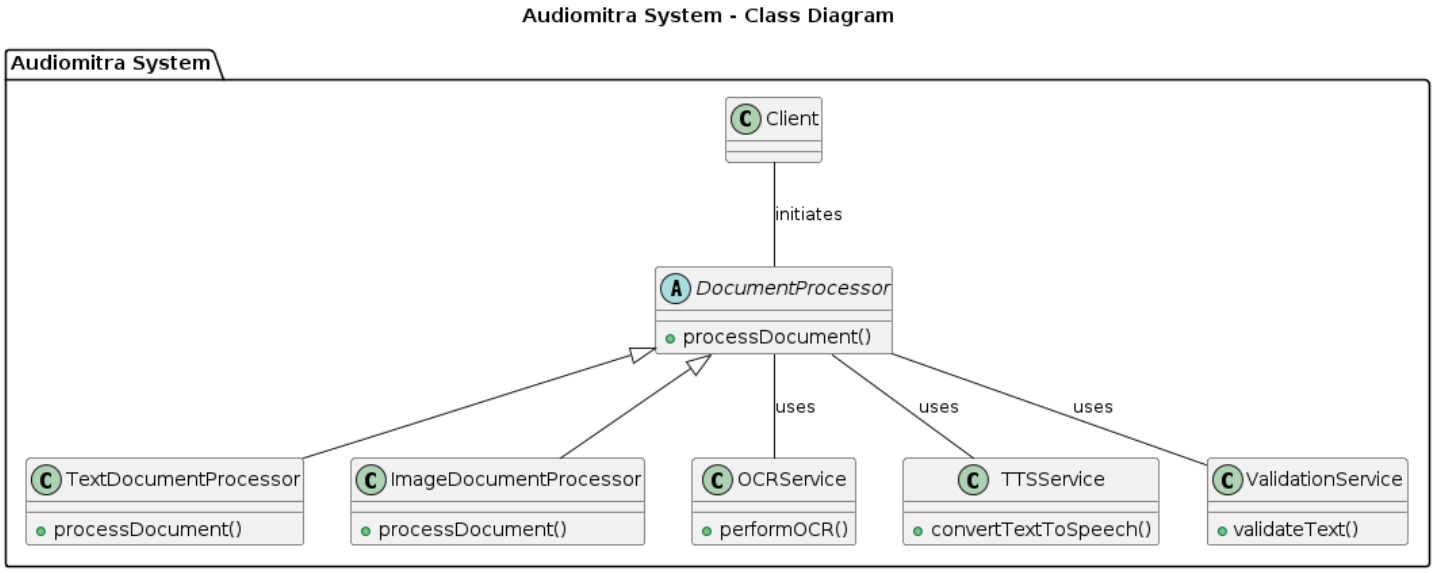
14. Physical View



15. Subsystems



16. UML Class Diagram



17. Microservices

These microservices are designed to be modular, independent, and scalable, allowing for easier development, deployment, and maintenance of the Audiomitra system.:

17.1. OCR Processing Microservice:

- Responsible for handling document processing using the OCR (Optical Character Recognition) extracting Text.
- Responsible for handling document processing tasks, including Optical Character Recognition (OCR) and Text-to-Speech (TTS) conversion.
- Communicates with external OCR and TTS APIs to extract text content and generate audio files.
- Validates extracted text and ensures accuracy before audio conversion.

17.2. Text Validation Microservice:

- Responsible for handling Validates extracted text and ensures accuracy before audio conversion by Human in the Loop.

17.3. Text Validation Microservice:

- Responsible for handling communicates with external TTS APIs to take text content and generate audio files for multilanguage.

17.4. User Management Microservice:

- Manages user authentication, authorization, and session management.
- Handles user registration, login, and profile management functionalities.
- Enforces access control policies and manages user roles and permissions.

17.5. File Storage Microservice:

- Stores and manages uploaded documents, audio files, and other multimedia assets.
- Ensures secure and scalable storage of user data and content.
- Provides access control and permissions for file management operations.

17.6. Integration Microservice:

- Handles communication with external APIs and services for OCR and TTS functionalities.
- Orchestrates data exchange between the system's components and external services.
- Ensures seamless integration and interoperability with third-party services.

18. Architectural Tactics that are employed in the Audiomitra

1. **Microservices Architecture:**
   - Decompose the system into smaller, independently deployable services.
   - Each microservice handles a specific set of functionalities, such as document processing, user management, and file storage.
   - Enables scalability, flexibility, and ease of maintenance.
2. **Caching:**
   - Implement caching mechanisms to store frequently accessed data and reduce latency.
   - Utilize caching for metadata, processed documents, and audio files to improve system performance and responsiveness.
   - Employ caching strategies such as time-based expiration and least recently used (LRU) eviction policies.
3. **Asynchronous Processing:**
   - Utilize asynchronous processing for long-running tasks and non-blocking operations.
   - Queue-based messaging systems like Kafka or RabbitMQ can be employed for event-driven processing and task orchestration.
   - Enhances system responsiveness, scalability, and fault tolerance.
4. **Load Balancing:**
   - Distribute incoming requests across multiple instances of microservices to evenly distribute the workload.
   - Implement load balancing strategies such as round-robin, least connections, or weighted distribution.
   - Improves system performance, availability, and fault tolerance by preventing overloading of individual services.
5. **Horizontal Scaling:**
   - Scale the system horizontally by adding more instances or nodes to handle increased loads.
   - Utilize containerization technologies like Docker and orchestration platforms like Kubernetes for automatic scaling and management of microservice instances.
   - Ensures system scalability, elasticity, and resilience to handle fluctuating workloads.

19. Implemented Architectural Patterns

We are using Flask, a lightweight web framework for Python. The architecture of our backend can be influenced by the Flask framework itself, as well as the specific needs of our application. Here are some architectural patterns and principles that can be applied to our Flask backend.

Implemented Architectural Patterns in "Document Processing System":

For Document Processing System, which involves handling tasks like Optical Character Recognition (OCR) and Text-to-Speech (TTS) conversion, and communicates with external APIs, let's consider two architectural patterns that could be particularly effective:

A. **Microservices Architecture (MA)**

B. **Event-Driven Architecture (EDA)**

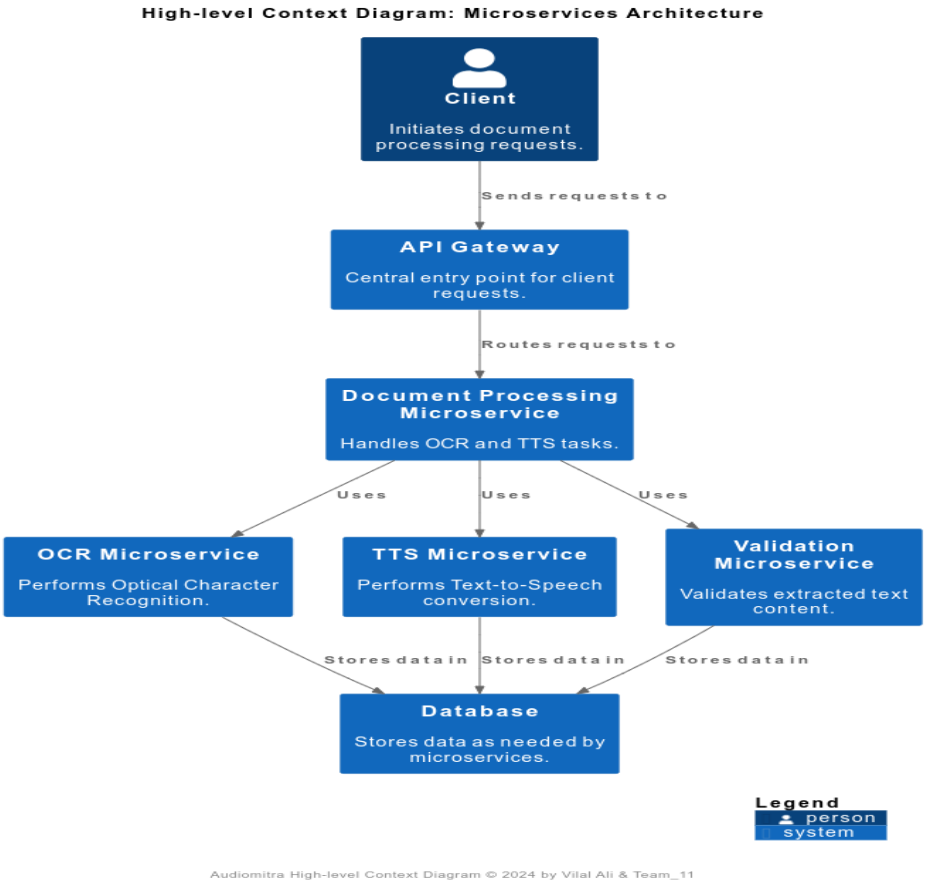
19.1. Microservices Architecture (MA) Pattern:

- **Service Segmentation:** Each task (OCR, TTS, validation) can be a separate microservice.
- **API Gateway:** A central entry point for client requests, routing them to the appropriate microservice.
- **Data Storage:** Each microservice can have its own database or share a common database, depending on the data consistency requirements.

19.1.1. Quality Attributes and Their Explanations

- **Scalability:** Each service (OCR, TTS, validation) can be scaled independently based on demand.
- **Flexibility:** Services can be developed, deployed, and updated independently.
- **Fault Isolation:** If one service fails, it doesn't directly impact the others.
- **Maintainability:** The ease with which the system can be maintained, modified, or enhanced by developers.
- **Reliability:** The ability of the system to consistently perform its intended functions without errors or failures.

19.1.2. High level Context Diagrams



19.1.3. Implementation: Codebase

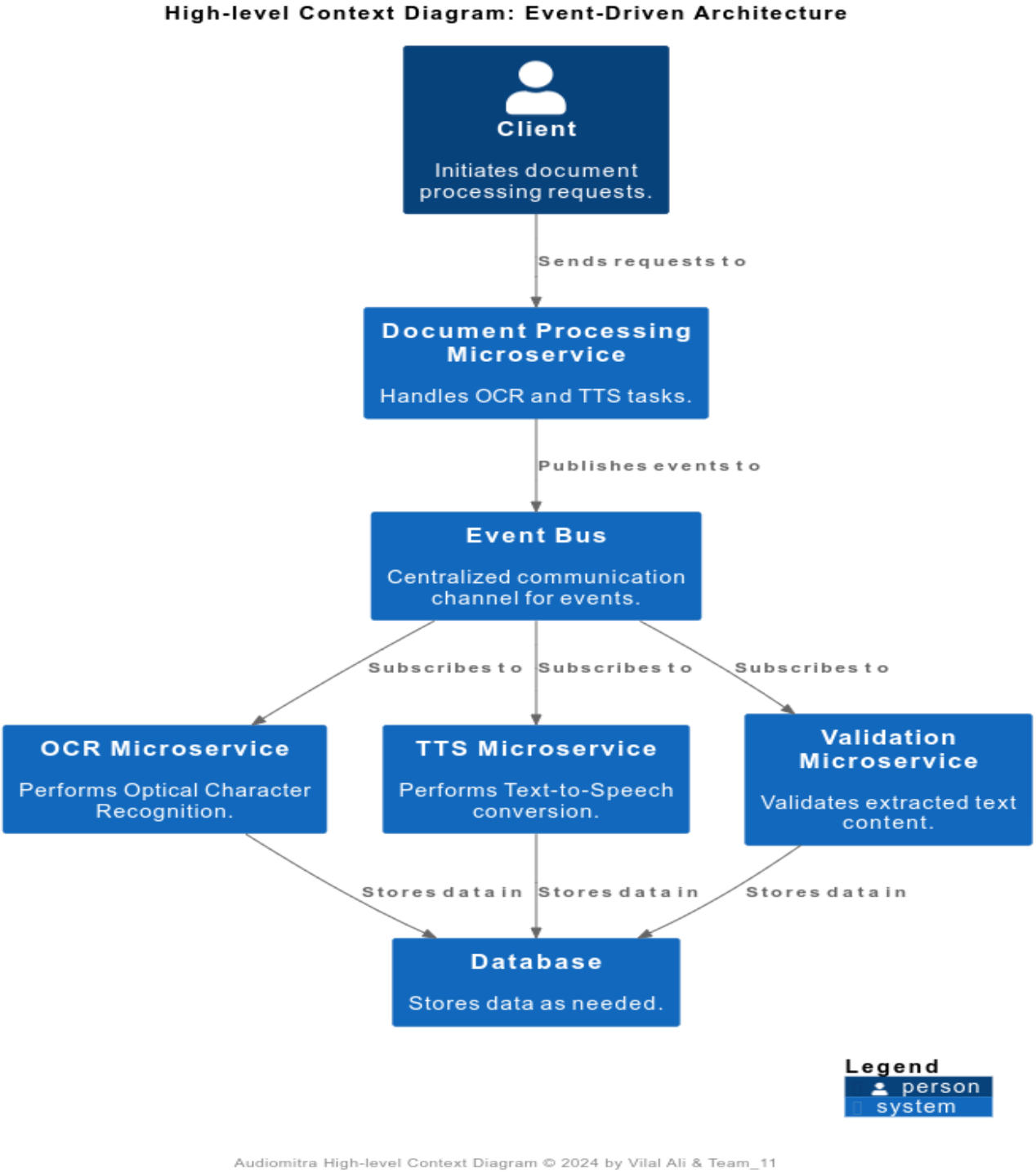
For the code snippet here is the link:
https://github.com/vilalali/audioMitra/tree/task4_architectural_patterns

19.2. Event-Driven Architecture (EDA) Pattern:

How It Applies:

- Event Production: When a document is uploaded or processed, an event is published.
- Event Consumers: Microservices subscribe to these events and perform their tasks (OCR, TTS, validation) based on the event data.
- Event Streaming: Tools like Kafka or RabbitMQ can be used to manage the event stream.

19.2.1. High level Context Diagrams



19.2.2. Quality Attributes and Their Explanations

- **Real-time Processing:** Enables the system to process events and data in real-time or near real-time, ensuring timely responses to user actions and updates.
- **Decoupling:** Components are loosely coupled, allowing for independent development, deployment, and scaling. Changes in one component do not affect others, enhancing flexibility and maintainability.
- **Scalability:** The system can easily scale horizontally to handle increasing volumes of events and data, distributing processing load across multiple instances or nodes as needed.
- **Reliability:** Ensures the reliability of event processing and delivery, minimizing the risk of data loss or processing errors. Redundancy and fault-tolerance mechanisms can be implemented to maintain system integrity.
- **Fault Tolerance:** Handles errors and failures gracefully, ensuring that the system remains operational even in the event of component failures or network issues. Redundancy and error-recovery mechanisms can be employed to minimize downtime and data loss.

19.2.3. 13.2.3 Implementation: Codebase

For the code snippet here is the link:
https://github.com/vilalali/audioMitra/tree/task4_architectural_patterns

20. A comprehensive analysis for Key quality metrics:

Quality Metrics	Model-View-Controller Pattern	Event Driven Architecture Pattern
Throughput	The throughput in MVC is influenced by the efficiency of data retrieval and processing within synchronous requests. Concurrent requests can be handled effectively depending on the scalability of the underlying infrastructure.	EDA may introduce additional overhead with event processing and message handling. Complex event patterns could impact overall system throughput.
Latency	Latency in MVC can be predictable within synchronous request-response cycles. Direct control over data flow can lead to faster response times for specific use cases.	EDA can introduce non-deterministic latency due to event propagation and asynchronous processing. Event-driven systems may experience variable response times based on event complexity and message processing.
Scalability	Vertical scaling is a common approach in MVC architectures, where individual components are scaled up. Horizontal scaling may require complex load balancing and session management strategies.	EDA supports horizontal scalability by decoupling components and allowing independent scaling of event processors. Scaling based on message queues enables efficient resource allocation for specific tasks.
Availability	Availability depends on the robustness of the underlying infrastructure and fault tolerance mechanisms. Single points of failure can impact availability during high load or failure scenarios.	EDA promotes fault isolation and resilience through event-driven communication and message queues. Redundancy in message processing enhances availability by mitigating the impact of failures.

Reliability	Reliability in MVC relies on error handling within synchronous operations and transaction management. Rollback mechanisms and data consistency are critical for maintaining reliability.	EDA enhances reliability through message acknowledgment, retries, and guaranteed message delivery. Redundant event processing improves fault tolerance and data integrity.
Maintainability	Maintainability is supported by the clear separation of concerns (Models, Views, Controllers) and modular design. Changes to individual components may require careful coordination to maintain system integrity.	Maintainability in EDA is achieved through loosely coupled components and event-driven interactions. Independent services can be updated or replaced without disrupting the entire system.

21. A comprehensive analysis of which pattern is better! and why?

For the "Document Processing Subsystem", both Microservices Architecture (MA) and Event-Driven Architecture (EDA) offer distinct advantages and challenges.

Microservices Architecture (MA):

Pros:

- **Scalability:** Each microservice can be scaled independently, allowing for efficient resource allocation based on demand.
- **Flexibility:** Microservices enable independent development, deployment, and updating of components, promoting agility and innovation.
- **Fault Isolation:** Failures in one microservice do not directly impact others, enhancing system resilience and availability.
- **Maintainability:** Modular design facilitates easier maintenance, modification, and enhancement of the system over time.
- **Reliability:** Isolation of failures improves system reliability by reducing the impact of potential issues.

Cons:

- **Complexity:** Managing multiple services and their interactions can introduce complexity in deployment, monitoring, and debugging.
- **Latency:** Inter-service communication over the network can introduce latency compared to in-process communication.
- **Operational Overhead:** Managing many microservices requires robust infrastructure, monitoring, and management tools.

Event-Driven Architecture (EDA):

Pros:

- **Scalability:** EDA enables asynchronous communication through events, allowing for efficient scaling based on event volume.
- **Flexibility:** Loose coupling between components allows for independent evolution and innovation, fostering flexibility.
- **Fault Isolation:** Decoupled components reduce the impact of failures, ensuring that one component's failure does not affect others.
- **Maintainability:** Components can be replaced or updated without disrupting the entire system, enhancing maintainability.

- **Real-time Processing:** EDA enables real-time event processing, making it suitable for systems with sporadic or unpredictable events.

Cons:

- **Complexity**: Designing event schemas, handling event consistency, and managing event delivery can introduce complexity.
- **Eventual Consistency:** Ensuring eventual consistency across distributed components can be challenging and may require careful design and implementation.
- **Monitoring and Debugging:** Debugging event-driven systems can be more challenging due to the asynchronous nature of communication and event-driven behavior.

22. Future works for the Audiomitra:

In future iterations of the Audiomitra project, several enhancements and expansions could be considered to further improve its capabilities and impact. Potential areas of future work include integrating advanced machine learning algorithms for enhanced document processing accuracy, such as natural language processing (NLP) techniques for language translation and sentiment analysis. Additionally, exploring the integration of voice recognition technologies could enable users to interact with the system through speech commands, further enhancing accessibility.

Further improvements in scalability and performance could be achieved by optimizing the system's architecture for cloud-native deployment, leveraging serverless computing and auto-scaling capabilities. Furthermore, expanding language support and accessibility features, such as subtitles and closed captions for audio content, could broaden the reach and inclusivity of the platform, making educational materials more accessible to diverse audiences. Additionally, incorporating user feedback mechanisms and analytics tools could provide valuable insights for iterative improvements and user-centric development. Overall, these future directions hold the potential to elevate Audiomitra's functionality, usability, and impact in transforming document and content accessibility.

23. Conclusion:

For the "Document Processing Subsystem" system, both Microservices Architecture (MA) and Event-Driven Architecture (EDA) offer distinct advantages and challenges.

MA excels in providing control over scalability, fault isolation, and reliability, making it suitable for systems with diverse and independent components like document processing tasks. However, managing the complexity of multiple services and ensuring efficient inter-service communication are essential considerations.

EDA, on the other hand, offers benefits in terms of flexibility, real-time processing, and fault isolation. It is particularly suitable for systems with sporadic or unpredictable events, where loose coupling and asynchronous communication are advantageous. However, designing and managing event-driven systems require careful attention to event schemas, consistency, and monitoring.

So, on the requirement basis we are adopt of Microservices Architecture (MA) for the "Document Processing Subsystem" system provides a robust foundation for building a scalable, resilient, and flexible system capable of meeting the dynamic needs of document processing tasks. By leveraging the benefits of MA, we aim to develop a high-performance and reliable system that delivers exceptional value to users and stakeholders.

24. Project Contribution:

S. N.	Task Name	Implemented Design Pattern	Asignee Name	Contribution
1	Task 1: Requirements and Subsystems	Functional and Non-functional Requirements.	Vilal	Vilal 100%
		Subsystem Overview		
2	Task 2: Architecture Framework	Stakeholder Identification	Madan	Madan 60% Vilal 20% Team 20%
		Major Design Decisions		
3	Task 3: Architectural Tactics and Patterns	5 Architectural Tactics employ in Audiomitra	Shriom	Hanuma 50% Shriom 30% Vilal 10% Team 10%
		Implementation Patterns	Hanuma	
4	Task 4: Prototype Implementation and Analysis	Prototype Development	Team	Vilal 40% Hanuma 20% Madan 20% Shriom 20%
		Architecture Analysis		
5	Project Report Writing	##	Vilal	Vilal 40% Shriom 20% Hanuma 20% Madan 20%
6	Project Repo Link	https://github.com/vilalali/audioMitra		
7	Other PR Link	https://github.com/vilalali/audioMitra/tree/task4_architectural_patterns		