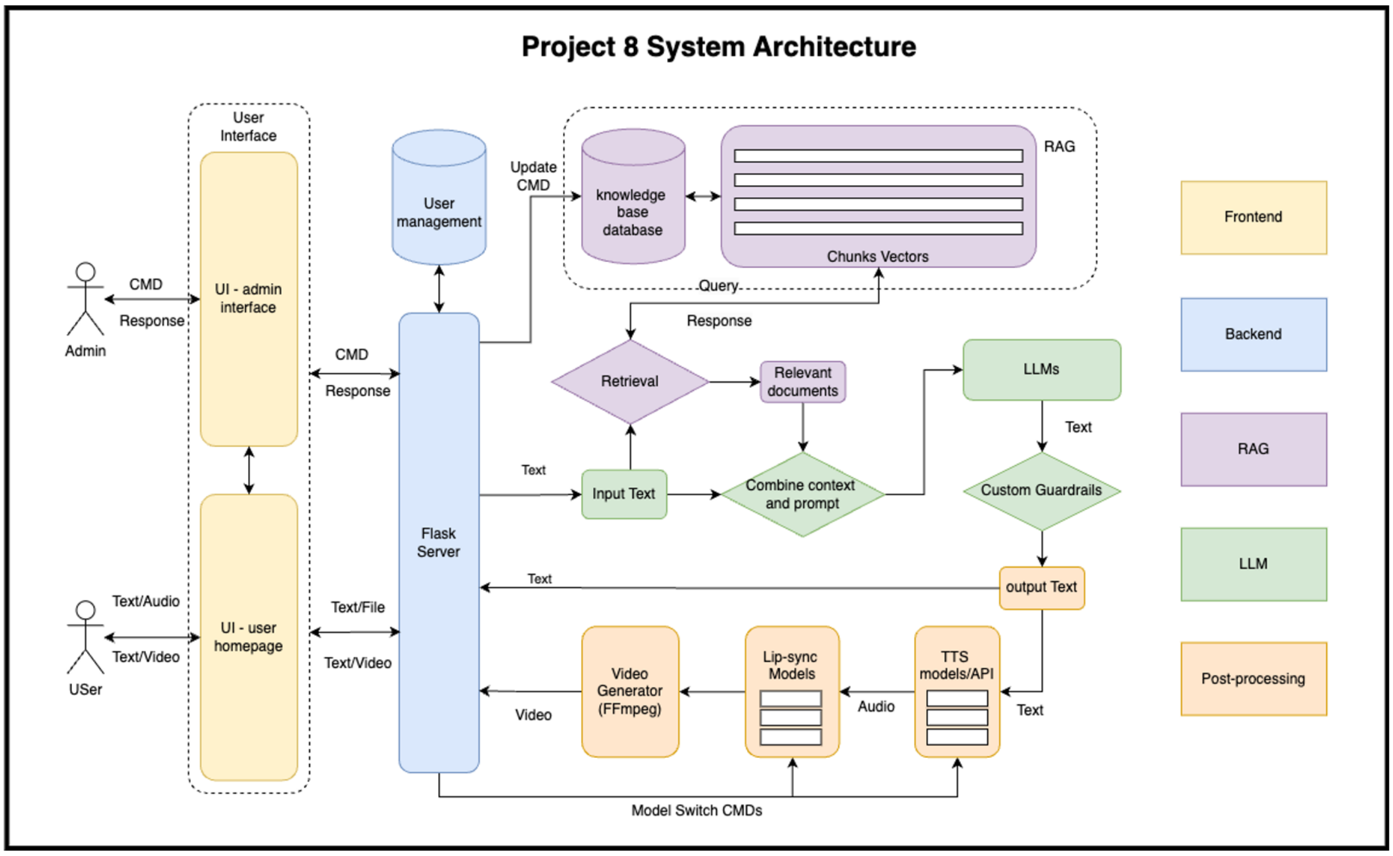
Technical Documentation

System Architecture Description

This system is an intelligent dialogue platform built upon multimodal input, knowledge augmentation, and large language models (LLMs). Users can interact with the system through text, speech, and images. The system supports capabilities such as natural language understanding, knowledge retrieval, speech synthesis, and video generation.

System Architecture Diagram



Brief Descriptions of Each Module

User Interface (User Homepage)

- Input Modes
  - Text
  - Voice
- Output Modes
  - Text
  - Video

- Personal Information Management
- User Customization
    - Theme / Background
    - Tutor Avatar

Admin Interface

- User Management
    - User information list
    - Edit user information
- Background Knowledge Base Management
    - Manage basic knowledge base
    - Manage user-specific knowledge base
- Tutor Avatar Management
    - Create or delete tutor avatars

Flask Backend (Core Orchestration Module)

- Handles all requests from users and administrators
- Routes inputs to various modules (ASR, TTS, RAG, LLM, etc.)
- Invokes Guardrails to verify content compliance
- Reads from and writes to the SQLite user database
- Coordinates data flow and result integration across modules

User Database (SQLite)

- Stored content:
    - Basic user information
    - Interaction logs
    - User input/output history
- Interface:
    - Accessed and managed by the Flask backend

RAG: Background Knowledge Vector Store (Milvus)

- Stored content:
    - Vectors generated from processed documents (see RAG module for details)
- Functions:
    - Supports multi-user knowledge vector databases and file management
    - Efficiently retrieves document content most relevant to a given query

    - Provides course-specific knowledge to downstream LLMs

Large Language Models (LLMs)

- Receive the synthesized prompt
- Generate intelligent responses based on background knowledge and user input
- Output responses are subject to post-generation Guardrails checks

Content Moderation Module (Custom Guardrails)

- Performs two-stage moderation: input and output
- Functions:
    - Detects and blocks sensitive, offensive, or non-compliant content
    - Supports customizable rules, keyword lists, or integration with moderation APIs

Text to Speech (TTS Model)

- Converts speech to text
- Provides transcribed text as input to the LLM
- Supported models/APIs:
    - EdgeTTS (default)
    - Tacotron
    - GPT-soViTs
    - CosyVoice2

Lip-sync Model

- Aligns speech with character lip movements for animation
- Supported models:
    - Musetalk
    - Lip2Wav

Video Generation / Streaming

- Uses TTS models to convert LLM responses into audio
- Applies lip-sync models to align lip animations with the audio
- Streams the synchronized video to the client using RTC technology
- Based on WebRTC (compatible with Chrome/Edge)

Audio and Video Model Management

- Start, stop, or switch the currently running TTS service

- Modify configuration settings for the TTS model
- Start or stop the active video synthesis service
- Change the tutor avatar bound to the selected lip-sync model

System Workflow Diagram

```
User input (text/audio/image)
↓
Flask Server processes the request
↓
(Audio input → ASR converts to text)
↓
Input Guardrails safety check
↓
RAG retrieves background knowledge (via Redis vector database)
↓
Merge retrieved documents + input text → Prompt
↓
Send to LLM to generate response text
↓
Output Guardrails safety check
↓
Depending on needs:
→ Output plain text
→ Generate video (TTS + lip-sync + FFmpeg)
↓
Result returned to user interface
```

Module Descriptions

UI - Homepage

Module Functions

- Intelligent dialogue interface: Enables real-time interaction between users and the AI tutor
- Multimedia support: Accepts various input formats including text, audio, images, and documents
- Session management: Create, select, favorite, and delete chat sessions
- Theme switching: Supports light and dark themes
- File upload: Supports PDF, Word, Excel, TXT, and other document formats
- Voice input: Real-time speech recognition and transcription
- Responsive design: Adapts to different screen sizes

Input and Output

Input

- User text messages

- Audio files (recorded via microphone)
- Document files (PDF, DOC, DOCX, XLS, XLSX, TXT)
- Session selection operations

Output

- AI tutor's text responses
- Conversation history
- Downloadable file links
- Real-time speech-to-text transcription results

Dependencies

- React 19.1.0 – Frontend framework
- Material-UI 7.1.2 – UI component library
- Axios 1.10.0 – HTTP client
- React Router DOM 7.6.3 – Routing management
- Web Speech API – Speech recognition
- File API – File processing
- Canvas API – Audio visualization

Technical Details

- **State management**: Utilizes React Hooks for component state control
- **API communication**: Interacts with backend via RESTful APIs
- **File upload**: Supports chunked uploads for large files with progress indication
- **Audio processing**: Analyzes and visualizes audio using Web Audio API
- **Caching mechanism**: Locally caches chat messages to improve user experience
- **Error handling**: Unified error handling and user feedback
- **Theming system**: Dynamic theme switching with light/dark mode support
- **Responsive layout**: Flexbox-based layout to support various devices

---

UI - Admin Page

Module Functions

- User management: View, create, edit, and delete user accounts
- Model management: Manage AI model configurations and runtime status
- Avatar management: Create and configure virtual avatars
- Knowledge base management: Maintain and monitor knowledge base content
- Access control: Role-based access control (RBAC)

- Data analytics: View user activity and system usage statistics
- Bulk operations: Perform batch actions on user accounts

Input and Output

Input

- User search and filter parameters
- Form data for creating/editing users
- File uploads (avatar images)
- Pagination parameters

Output

- User list data
- Model configuration details
- Avatar configuration data
- Knowledge base list

Dependencies

- React 19.1.0 – Frontend framework
- Material-UI 7.1.2 – UI component library
- Axios 1.10.0 – HTTP client
- React Router DOM 7.6.3 – Routing management
- File API – File upload handling
- Form Validation – Input validation

Technical Details

- **Table components**: Configurable data tables with sorting and filtering support
- **Modal dialogs**: Pop-up interfaces for user creation and editing
- **File upload**: Drag-and-drop support, accepts multiple image formats
- **Form validation**: Real-time validation of user input
- **Access control**: Interface rendering based on user roles
- **Data caching**: Local caching of user data to reduce API requests
- **Error handling**: Centralized error feedback and handling mechanism
- **Responsive design**: Admin UI adapts to various screen sizes
- **Theming system**: Light and dark mode support

Server - User Login

Module Functions

- Handles user login, registration, verification code delivery, password updates, and logout
- Implements JWT-based authentication
- Enforces a single-token-per-user login mechanism
- Automatically stores login state in Redis for session binding and state management

Input

- Email + password (for login/registration)
- Verification code (for registration/password update)
- Session ID (optional, for external service binding)
- Password update form data
- JWT token (for session validation and logout)

Output

- JWT token upon successful login
- Status of verification code delivery
- Remaining token validity (in seconds/minutes)
- Confirmation messages for registration and password updates

Dependencies

- Flask 2.x – Web framework
- Flask-JWT-Extended – JWT authorization middleware
- Redis – Stores login tokens, verification codes, and session IDs
- SQLAlchemy – ORM for user data operations

Custom Modules:

- `verification_cache` : Manages rate-limiting and validation of verification codes
- `email_service` : Handles email delivery
- `token_utils` : Verifies and manages tokens within Redis

Technical Details

- Old tokens are cleared on login to enforce forced logout (single-login policy)
- Verification codes are rate-limited with a 10-second cooldown to prevent abuse
- All session states are stored in Redis with keys: `token:{jti}` and `user_token:{user_id}`
- All sensitive actions (e.g., password updates) require verification code validation
- Failed requests return a unified JSON error structure
- Compatible with external service session binding via `sessionid`

Server - User Management

Module Functions

- Manage user accounts: query, create, update, delete
- Maintain user profile information, support avatar uploads, and account deactivation
- Tutor administrators can access all user records and system logs
- Integrated user activity logging system to track modification history

Input

- User search and filter parameters (pagination, role, status, keywords)
- Form data for user creation/editing (email, password, role, bio, etc.)
- File uploads (avatar images)
- Verification code (for account deactivation)
- Log query parameters for tutor administrator actions

Output

- Paginated and filtered user list
- Detailed user information including profile data
- Avatar image access URL after upload
- User activity logs (timestamp, target user, changed fields)
- Simulated model configuration and knowledge base data (placeholder for actual integration)

Dependencies

- Flask 2.x – Backend framework
- Flask-JWT-Extended – Authorization and permission validation
- SQLAlchemy ORM – Modeling and operating on user, log, and notification data
- Redis – Temporary session ID storage and logout support
- File system – Avatars stored under `/static/avatars`

Technical Details

- All user operations are recorded in the `UserActionLog` table for auditability
- Tutor-level permission is strictly enforced on the backend (`user.role` based check)
- Avatar upload supports jpg/png/jpeg formats, renamed to `user_<id>.jpg` automatically
- Form updates implement differential logging (only changed fields are recorded)
- Account deactivation requires a verification code and supports pre-deletion snapshots
- All APIs return standardized JSON responses for frontend compatibility

- User queries support pagination, fuzzy search, and filtering by role/status

---

LLMs

System Architecture

The RAG AI Assistant is built on a sophisticated LangGraph workflow that orchestrates multiple AI components to provide intelligent conversational responses. The system follows a modular architecture with clear separation of concerns.

High-Level Architecture

```
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   Web Client    │    │   API Server    │    │   LangGraph     │
│                 │◄──►│   (Flask)       │◄──►│   Workflow      │
└─────────────────┘    └─────────────────┘    └─────────────────┘
                                │                       │
                                ▼                       ▼
                       ┌─────────────────┐    ┌─────────────────┐
                       │   Milvus API    │    │   Ollama LLM    │
                       │   (Vector DB)   │    │   (Local)       │
                       └─────────────────┘    └─────────────────┘
                                │                       │
                                ▼                       ▼
                       ┌─────────────────┐    ┌─────────────────┐
                       │   Tavily API    │    │   SQLite        │
                       │   (Web Search)  │    │   (Sessions)    │
                       └─────────────────┘    └─────────────────┘
```

Core Components

1. API Interface (`api_interface.py`)

Purpose : Flask-based REST API server that handles HTTP requests and manages streaming responses.

Key Features :

- Streaming chat interface with Server-Sent Events (SSE)
- Model management and switching
- Health check endpoints
- Session management with SQLite checkpointing

Main Endpoints :

- `POST /chat/stream` : Main streaming chat interface
- `GET /health` : Health check
- `POST /activate_model` : Model switching

Implementation Details :

```python
# Streaming response generation
def generate_stream():
    q = queue.Queue()

    async def async_worker():
        async with AsyncSqliteSaver.from_conn_string("checkpoints.db") as checkpointer:
            builder = create_assistant_workflow()
            app_workflow = builder.compile(checkpointer=checkpointer)
            async for event in app_workflow.astream(inputs, config={"configurable":
{"thread_id": session_id}}, stream_mode=["custom"]):
                chunk_data = {
                    "chunk": event[1]['chunk'],
                    "status": "streaming",
                    "timestamp": datetime.now().isoformat(),
                }
                q.put(f"data: {json.dumps(chunk_data, ensure_ascii=False)}\n\n".encode('utf-
8'))
```

2. AI Assistant Workflow (`ai_assistant_final.py`)

Purpose: Core LangGraph workflow that orchestrates the entire conversation flow.

Workflow Nodes:

Guardrail Check Node

```python
async def guardrail_check(state: AssistantState) -> AssistantState:
    """Content safety classification"""
    # Classifies queries as: normal, homework_request, harmful
    # Uses LLM to determine content safety
```

Query Rewrite Node

```python
async def query_rewrite(state: AssistantState) -> AssistantState:
    """Intelligent query reformulation"""
    # Rewrites user queries for better retrieval
    # Considers conversation history for context
```

Query Classification Node

```python
async def classify_query(state: AssistantState) -> AssistantState:
    """Determines retrieval strategy"""
    # Classifies as: no_retrieval, need_rag, need_web_search
    # Routes to appropriate processing path
```

Document Retrieval Node

```python
async def retrieve_documents(state: AssistantState) -> AssistantState:
    """Milvus API integration"""
    # Queries vector database for relevant documents
    # Supports personal and public knowledge bases
```

Web Search Node

```python
async def search_external(state: AssistantState) -> AssistantState:
    """Tavily search integration"""
    # Performs web search for real-time information
    # Fallback when RAG results are insufficient
```

Response Generation Node

```python
async def generate_response(state: AssistantState) -> AssistantState:
    """LLM-powered response generation"""
    # Generates streaming responses with context
    # Integrates retrieved documents and search results
```

3. Milvus API Client (`milvus_api_client.py`)

Purpose: Client for interacting with the Milvus vector database service.

Key Features:

- HTTP-based API client for Milvus service
- Configurable parameters for retrieval
- Error handling and retry logic
- Health check functionality

API Integration:

```python
def query(self, question: str, user_id: str, personal_k: int = 5,
          public_k: int = 5, final_k: int = 5, threshold: float = 0.5):
    """Query Milvus database with parameters"""
    request_data = {
        "question": question,
        "user_id": user_id,
        "personal_k": personal_k,
        "public_k": public_k,
        "final_k": final_k,
        "threshold": threshold
    }
```

4. Configuration Management (`milvus_config.py`)

Purpose: Centralized configuration management for the system.

Configuration Options:

- Milvus API settings (URL, timeout, endpoints)
- Default retrieval parameters
- Environment variable support

Implementation:

```python
class MilvusConfig:
    MILVUS_API_BASE_URL: str = os.getenv("MILVUS_API_BASE_URL", "http://localhost:9090")
    MILVUS_API_TIMEOUT: int = int(os.getenv("MILVUS_API_TIMEOUT", "30"))
    DEFAULT_PERSONAL_K: int = int(os.getenv("DEFAULT_PERSONAL_K", "5"))
    DEFAULT_PUBLIC_K: int = int(os.getenv("DEFAULT_PUBLIC_K", "5"))
    DEFAULT_FINAL_K: int = int(os.getenv("DEFAULT_FINAL_K", "10"))
```

Data Flow

1. Request Processing Flow

```
User Input → API Server → LangGraph Workflow → Response Generation → Streaming Response
```

2. Detailed Workflow Steps

1. **Input Validation**: API server validates required fields
2. **State Initialization**: Creates AssistantState with user input
3. **Guardrail Check**: Content safety classification
4. **Query Rewrite**: Intelligent query reformulation
5. **Query Classification**: Determines retrieval strategy
6. **Retrieval/Search**:
    1. If RAG needed: Query Milvus API
    2. If web search needed: Query Tavily API
    3. If no retrieval: Skip to generation
7. **Response Generation**: LLM generates streaming response
8. **Session Persistence**: Save conversation state to SQLite

3. State Management

The system uses `AssistantState` TypedDict for state management:

```python
class AssistantState(TypedDict):
    user_id: str
    session_id: str
    input: str
    messages: Annotated[List[Any], operator.add]
    model: str
    safety_classification: str
    is_safe: bool
    classification: str
    retrieved_docs: List[Dict[str, Any]]
    search_results: List[Dict[str, Any]]
    reranked_results: List[Dict[str, Any]]
    final_response: str
    sources: List[str]
    rewritten_query: str
```

External Dependencies

1. Ollama LLM Service

Purpose: Local LLM inference engine

Models:

- `mistral-nemo:12b-instruct-2407-fp16` (default)
- `llama3.1:8b-instruct-q4_K_M`

Integration:

```python
@lru_cache(maxsize=1)
def get_llm(model:str="mistral-nemo:12b-instruct-2407-fp16"):
    return ChatOllama(
        temperature=0.4,
        disable_streaming=False,
        model=model,
        base_url="http://127.0.0.1:11434"
    )
```

2. Milvus Vector Database

Purpose: Vector similarity search for document retrieval

API Endpoints:

- Query: `POST /retriever`
- Health: `GET /retriever`

Query Parameters:

- `question`: Search query
- `user_id`: User identifier
- `personal_k`: Personal knowledge results count
- `public_k`: Public knowledge results count
- `final_k`: Total results count
- `threshold`: Similarity threshold

3. Tavily Search API

Purpose: Web search for real-time information

Integration:

```python
search_tool = TavilySearch(max_results=5)
```

Security and Safety

1. Content Safety System

The system implements a three-tier content safety approach:

- **Guardrail Classification**: LLM-based content classification
- **Response Filtering**: Automatic blocking of inappropriate content
- **Academic Integrity**: Prevention of homework cheating

2. Input Validation

- Required field validation
- Type checking
- Length limits
- Sanitization

3. Error Handling

- Comprehensive exception handling
- Graceful degradation
- Detailed error logging
- User-friendly error messages

Performance Considerations

1. Streaming Responses

- Real-time response generation
- Chunked data transmission
- Connection management
- Error recovery

2. Caching Strategy

- LLM instance caching with `@lru_cache`
- Session state persistence
- Query result caching

3. Concurrency

- Async/await pattern for I/O operations
- Thread-based streaming
- Non-blocking API calls

Monitoring and Logging

1. Logging Strategy

- Structured logging with timestamps
- Error tracking and stack traces
- Performance metrics
- Request/response logging

2. Health Monitoring

- API health checks
- External service monitoring
- Model availability checks
- Database connectivity

Deployment Considerations

1. Environment Setup

- Python 3.8+ requirement
- Ollama service installation
- Milvus API service
- Environment variable configuration

2. Scaling Considerations

- Load balancing for API servers
- Database connection pooling
- Model serving optimization
- Caching strategies

3. Security Measure

- API authentication (future enhancement)
- Rate limiting (future enhancement)
- Input sanitization
- CORS configuration

Testing Strategy

1. Unit Testing

- Individual component testing
- Mock external dependencies
- Error condition testing

2. Integration Testing

- End-to-end workflow testing
- API endpoint testing
- External service integration

3. Performance Testing

- Load testing
- Response time measurement
- Memory usage monitoring

Future Enhancements

1. Planned Features

- Authentication and authorization
- Rate limiting
- Advanced caching

2. Scalability Improvements

- Microservices architecture
- Containerization
- Kubernetes deployment
- Horizontal scaling

3. Advanced Capabilities

- Multi-modal support
- Advanced RAG techniques
- Custom model fine-tuning
- Analytics and insights

RAG

Overview

Introduction

1. Retrieval-Augmented Generation (RAG) enhances the performance of LLMs by incorporating external knowledge into the generation process. This has two critical purposes:(1) it mitigate the “hallucinations” by providing LLM factual and up-to-date context.(2)it enables LLM to respond correctly to course-specific questions.

2. In this project, RAG provides TutorNet with a multi-tenant, course-specific knowledge support.
3. When a student asks a course-related question, TutorNet's RAG will retrieves relevant content from both the public and the student's personal knowledge bases and supplies it to the downstream LLM, which then generates a precise, context-aware answer.

Key features

Multi-Tenant Vector Knowledge Bases

- The system maintains two types of vector stores: a public knowledge base overseen by administrators and isolated personal knowledge bases for individual users. Each user's knowledge base is kept separate; users can manage their own personal store, while administrators control the shared public repository.Users/Administrator can upload and delete documents stored in their knowledge base independently.

Pure-Text Scenario

- The default RAG mode encodes document text into embeddings and stores them in text-only collections. Queries are vectorized and matched against these text embeddings for full-text search and fine-grained chunk-level re-ranking. This mode delivers low-latency, GPU-efficient retrieval ideal for purely textual content.

Multimodal Scenario (Optional)

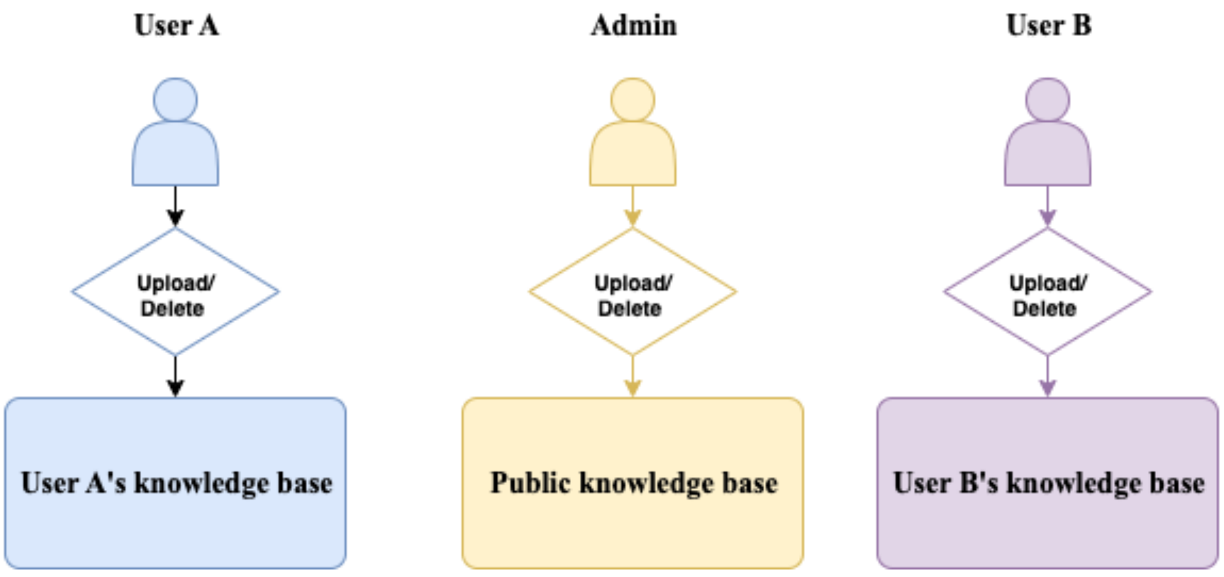
- An optional multimodal pipeline can be enabled in `config.py` when multimodal retrieval is required. It encodes documents into page-level multimodal embeddings(capturing text images, diagrams and charts) and chunk-level text embeddings.Retrieval then runs in two stages : a coarse page search on multimodal embeddings, followed by a fine-grained chunk search on text embeddings.Results include text snippets paired with the most relevant slides images. Multimodal processing can be toggled based on content characteristics and GPU resource availability.
- Currently, the RAG module supports both pure-text and multimodal scenarios; however, the multimodal functionality has not yet been integrated into the overall system due to GPU limitations.

Query-Relevant Content Augmentation

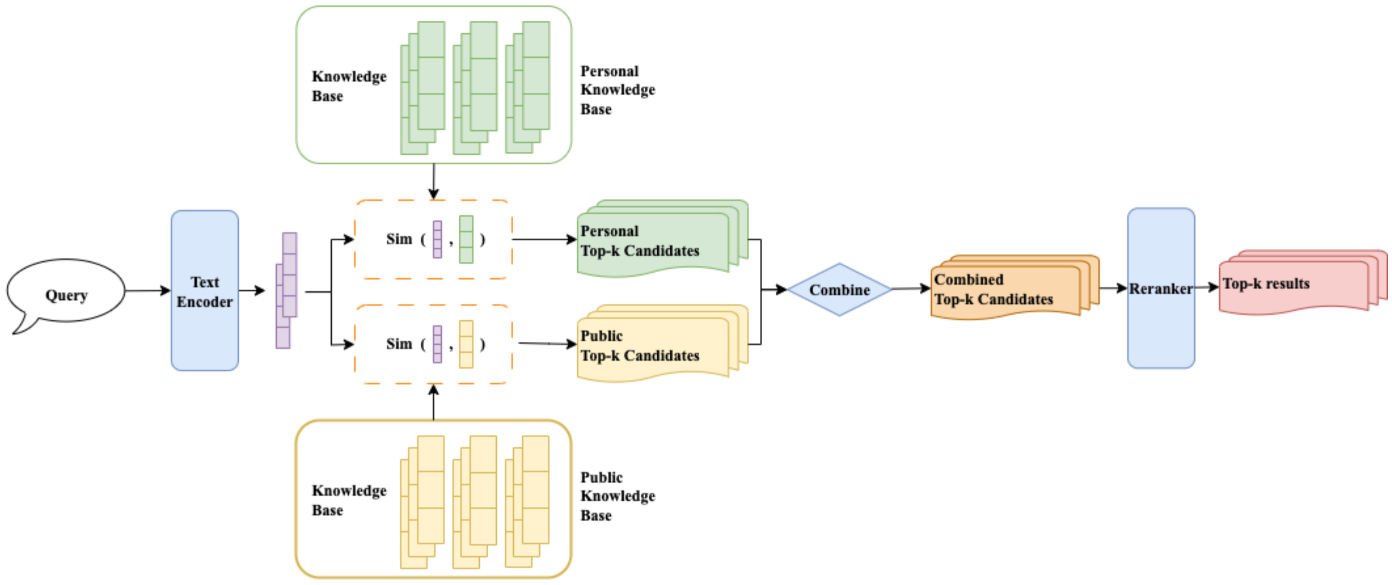
- For any incoming query, the system retrieves the top-k most relevant chunks from both public and personal knowledge bases, then supplies these precise context snippets to the downstream LLM. This grounding strategy reduces hallucinations and ensures accurate, course-specific answers.

Pipeline Diagram

Multi-tenant Knowledge Base Architecture



RAG Query-Retrieval Pipeline



Dependencies

- pymilvus >= 2.5.14
- pymilvus-model >= 0.3.2
- sentence-transformers >=4.1
- torch >2.7.1
- pdf2image>=1.16.0
- pillow>=8.0.0
- PyMuPDF>=1.22.0
- regex>=2022.10.31

Technical Details

Multi-tenant knowledge base

Vector Database Selection

This project's knowledge base base in Milvus for following reasons:

- **Multi-tenant isolation:** All tenants share the same Milvus cluster yet operate in fully independent data environments. Through a dedicated interface, users can create their own Collections as their personal knowledge base, upload or delete files. This guarantees total isolation of each user's data and indexes and supports hot updates.
- **Flexibility:** Milvus supports a variety of deployment models and provides a comprehensive integration of different search and indexing strategies.
- **Multimodal extensibility:** Milvus supports multimodal data by offering a unified infrastructure and tools to integrate, process, and analyze diverse data types such as text, images, audio, and sensor data.
- **High scalability**： Milvus is designed to handle massive vector datasets and support GPU acceleration.

Knowledge Base Architecture

The multi-tenant knowledge in this project is based on Milvus Collection-Oriented multi-tenancy strategy. It is organized into Public and Personal knowledge base.

- **Public knowledge base:** Managed by administrators(upload/delete files). Users are not allowed to manage a public knowledge base.
- **Personal knowledge base:** Each user manages their own knowledge base (upload/delete files).

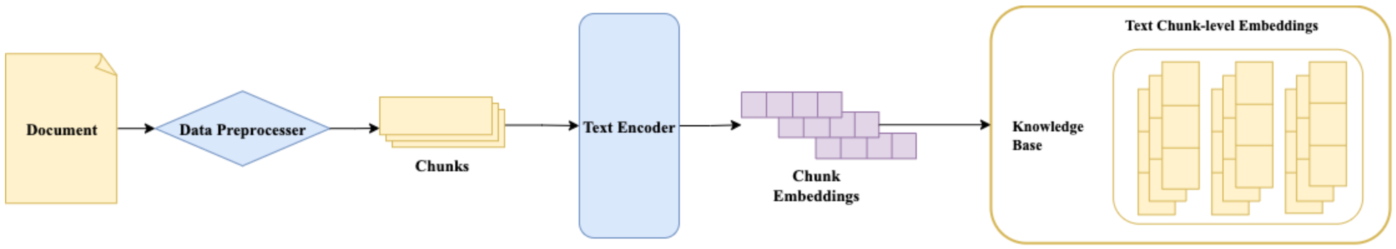
Collections settings for different scenarios:

- **Plain-Text Scenario:** 1 Collection per administrator or user to store chunk-level text embeddings.
- **Multimodal Scenario:** 2 collections per administrator or user. One to store multimodal page-level embeddings and the other to store chunk-level text embeddings.

Knowledge Base File Management

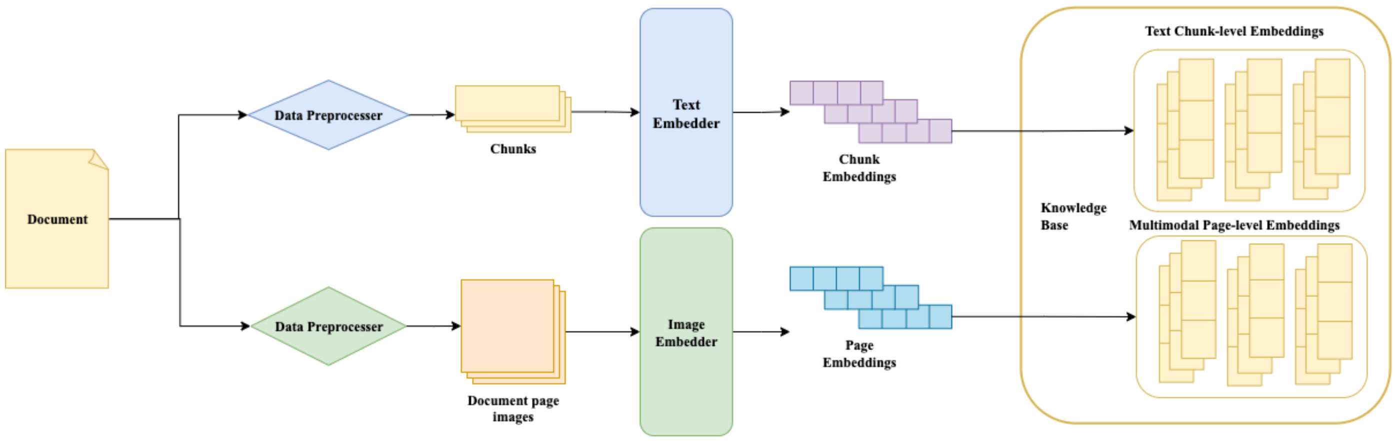
File Upload

- **Pure-text scenario**



The raw documents are ingested into the knowledge base via the workflow shown in the diagram; in addition to the embeddings themselves, all associated metadata is also stored alongside them.

- **Multimodal scenario**



The raw documents are ingested into the knowledge base via the workflow shown in the diagram; in addition to the embeddings themselves, all associated metadata is also stored alongside them.

File Delete

When a filename is provided for deletion, the system looks up that filename in the indexed metadata, identifies all embeddings associated with it, and removes them from the knowledge base.

Data Preprocess & Data Embeddings

Pure-Text Scenario

- Data Cleaning
  - Read raw text lines from the document
  - Detect and identify recurring headers and footers
  - Remove all instances of those headers and footers
  - Perform basic text cleaning (e.g., normalize bullet symbols, collapse excess whitespace)
  - Split the cleaned text into sentences at multiple granularity levels
  - Group sentences into semantically coherent chunks of controlled size with configurable overlap
  - Package each chunk into a standardized `Document` structure
- Data Embedding
  - After benchmarking several embedding models, RAG selected all-MiniLM-L6-v2 pretrained model from the SentenceTransformers library as the text encoder due to its strong semantic retrieval performance and lightweight inference characteristics.

Multimodal Scenario

In the multimodal scenario, document data preprocess and embedding are handled in two parallel pipelines:

- Multimodal Page-level Pipelines
  - Page rendering (convert each PDF page into an image)
  - Data Embedding
  - Colpali-v1.3 is used for its outstanding ability to capture multimodal information (text, images, and layout).
- Text Chunk-level Pipelines

The same as in Pure-Text Scenario

Retrieval

Pure-Text Scenario

Two-stage Retrieval Pipeline

Stage-1: Initial Retrieval

- Input: `query`
- Process:
- Encode the query into a vector (`query_embedding`).
- Perform a vector-based search and do similarity calculations in two separate knowledge bases:
    1. Search in personal knowledge base and return top-k most similar results `personal_hits`
    2. Search in public knowledge base and return top-k most similar results `public_hits`
- Combine `personal_hits` and `public_hits` and sort the combined results.
- Get the top k results from the combined results `combined_hits` .

Stage-2:Re-ranking

- Input: `candidates` (the `combined_hits` in stage-1) and orginal `query`
- Process:
- For each candidate, form a (query, candidate) pair and score it with a cross-encoder, producing a fine-grained `cross_score` .
- Fuse the stage-1 `retrieval_score` and the `cross_score` to get the `final_score`
- Sort all candidates by `final_score` to produce the final ranked top-k results.

Similarity Metric

The whole process uses **cosine similarity**

Cross-encoder Details

`ms-marco-MiniLM-L-6-v2` as cross-encoder.

Score Fusion Details

1. Score fusion has two steps: normalization and weighted combination.
2. Normalization: applying Min-Max scaling to both scores `retrieval_score` , `cross_score` .
3. Weighted Combination:
4. Compute a weighted sum.(alpha is set to 0.4)

```
final_score = α × normalized_retrieval_score + (1 − α) × normalized_cross_score
```

Multimodal Scenario

Two-stage Retrieval Pipeline

- Stage 1: Page-level Retrieval
    - Input: `query`
    - Process:

- Encode the query with Colpali encode to obtain `query_embedding`
    - Perform a vector-based search and do similarity calculations in two separate knowledge bases:
        1. Search in personal knowledge base and return top-k most similar results `personal_hits`
        2. Search in public knowledge base and return top-k most similar results `public_hits`
    - Deduplicate the combined hits to form a set of candidate pages.
    - From these candidates, select the overall top k pages `page_hits` by applying a Max-similarity and get the `page_score` .
- Stage 2: Chunk-level Retrieval
    - Input: `query` `page_hits`
    - Process:
    - For each of the selected pages in `page_hits` ,retrieve all of its constituent text chunks as candidate.
    - For these candidate chunks, get the similarity scores `chunk_score` with query .
    - Fuse the stage-1 `page_score` and `chunk_score` to get the `final_score`
    - Sort all candidates by `final_score` to produce the final ranked top-k results and return the most relevant slides images.

Score Fusion Detail

Score fusion has two steps: normalization and weighted combination.(The same as in the Pure-Text Scenario)

Similarity Metric

Stage 1(b) and stage 2(b) use cosine similarity

Experiment & Evaluation

Customized Dataset

Dataset

To evaluate the RAG system's performance, this project generated a question–answer dataset of 128 questions based on the COMP9331/COMP3331 lecture slides PDFs.

Each example in the 128-pair dataset is represented as a JSON object with the following fields:

`"question"`	**The full text of the question being asked.**
`"answer"`	A list of key terms (or short tokens) that together form the concise answer.
`"page_ids"`	An array of page numbers in the source PDF where the answer can be found.
`"source"`	The filename of the document from which this Q&A pair was extracted.

Evaluation

Metric

- A retrieval is considered successful if the page containing the answer is retrieved.
- Recall@k is used to evaluate retrieval performance.

Experiment

	recall @1	recall @3	recall @5	recall @10
`BGE-M3`	48.4%	64.1%	70.3%	76.6%
`all-MiniLM-L6-v2`	51.6%	67.2%	73.4%	78.9%
`all-MiniLM-L6-v2` + `ms-marco-MiniLM-L-6-v2`	52.3%	72.7%	77.3%	81.2%
`Colpali` + `all-MiniLM-L6-v2` (cascade)	53.9%	70.3%	79.7%	82.0%
`Colpali` + `all-MiniLM-L6-v2` (hybrid)	50.0%	71.9%	78.1%	81.2%

`Colpali` + `all-MiniLM-L6-v2` (cascade) is the pipeline introduced in Retrieval-Multimodal Scenario

`Colpali` + `all-MiniLM-L6-v2` (hybrid) is the fusion pipeline. It independently queries both ColPali (page-level) and all-MiniLM-L6-v2 (chunk-level), then merges their results into a single combined ranking.

Analysis

- Baseline comparison
  - Switching from BGE-M3 → all-MiniLM: +3.2 pct at R@1 and +2.3 pct at R@10.
  - Adding MS-MARCO cross-encoder brings a large +5.5 pct lift at R@3 and pushes all recall metrics above 80%.
- Cascade vs. Hybrid
  - For multimodal pipeline, cascade pipeline achieves the highest recall at R@1, R@5, and R@10, indicating its strength for precise page localization.

MMDocIR(selected)

Dataset

The MMDocIR dataset is a comprehensive multi-modal retrieval benchmark covering ten distinct domains. Originally comprising 313 documents drawn from a variety of modalities (text, tables, figures, charts, etc.), I selected a balanced subset of 50 documents to accommodate our time and computational resources. In choosing these 50 and maximized the diversity of question modalities (e.g. text-only, chart-based, layout-driven). Due to ongoing resource constraints, current evaluation focuses on a comparative performance analysis of text retrieval model and multimodal retrieval model: BGE-M3 and Colpali.The total number of questions is 409.

Results

	recall @3	recall @5	recall @10
`BGE-M3`	16.5%	24.2%	37.2%
`Colpali`	23.9%	32.7%	50.1%

Analysis

On the dataset with various modalities, Colpali outperforms BGE-M3 significantly due to its capability o capturing images,charts,tables and layout information.

Conclusion & Future Outlook

Conclusion

For the project's needs, indexing the curriculum-specific course materials can be handled by a lightweight, pure-text scenario well. These documents are narrowly scoped and benefit from minimal compute and fast response times, so encoders like `all-MiniLM-L6-v2` strike the ideal balance of speed and accuracy.

In highly diverse, large-scale collections that include text, images,charts, tables and complex layouts, a multi-modal scenario consistently delivers the highest recall by exploiting both visual and structural cues.

Future Outlook

- **Accelerated Multi-Modal Indexing:** At present, retrieval in multimodal scenario takes roughly eight seconds, largely due to limited GPU resources. Since Milvus natively supports GPU-accelerated search, I anticipate that offloading vector search and index building to GPUs will dramatically reduce this latency.
- **Vision-Language Model Integration:** Resource constraints have so far prevented us from employing vision-language models (VLMs) in the downstream LLM stage. Enabling a multi-modal RAG setup that couples ColPali (or similar) with a VLM would unlock richer cross-modal reasoning and further improve retrieval accuracy.
- **Scalability with Milvus Standalone:** Our current setup uses Milvus Lite, which is ideal for small to medium workloads. If a larger knowledge base is needed, migrating to Milvus Standalone will provide enhanced throughput, fault tolerance, and cluster-level GPU acceleration, ensuring robust performance on big data.

TTS

Introduction

This project builds a unified Text-to-Speech (TTS) service platform, consisting of a central controller module (`tts.py`) and four independent subservice modules (`server.py`), each running a different TTS model. The design goals are as follows:

1. **Centralized orchestration and decoupled management**: The main controller (`tts.py`) provides unified interface management and scheduling for all model services.
2. **Multi-model support**: Each submodule runs independently and loads a different model. Some models support voice cloning, while others support voice style switching.
3. **Decoupled architecture & extensibility**: Submodules can be independently deployed, making it easy to update, replace, or add new models.
4. **FastAPI-based communication**: FastAPI is used for inter-module communication, allowing the main controller to interact with submodules and return unified responses.

Module Functions

- Text-to-speech conversion
- Used for voice responses and video generation
- Voice style customization via parameters
- Voice cloning support in certain models

- Model parameter configuration
- Start and stop TTS services
- Switch between running TTS services

Key Features

TTS Controller Module

- Provides a unified external interface
- Dispatches incoming requests to the appropriate model service based on parameters
- Aggregates and returns output from model services
- Highly extensible — easily add or remove subservice modules

Model Subservices (`server.py`)

- Each `server.py` deploys an independent TTS model service
- Responsible for service startup, audio output, and configuration changes
- Independently executed with resource isolation, enabling parallel processing and easy maintenance

System Architecture

```
                       +------------------+
                       |   Client Request |
                       +--------+---------+
                                |
                                v
                       +--------+---------+
                       |     TTS server   |
                       +--------+---------+
                                |
        +-----------+-----------+-----------+-----------+
        |           |                       |           |
        v           v                       v           v
+---------------+ +---------------+ +---------------+ +---------------+
|  Model Server | |  Model Server | |  Model Server | |  Model Server |
|    (A)        | |    (B)        | |    (C)        | |     (D)       |
+---------------+ +---------------+ +---------------+ +---------------+
```

Models and Technical Details

EdgeTTS (Default)

API Overview

EdgeTTS is a text-to-speech service provided by Microsoft, based on Azure Cognitive Services. It uses the same backend as the reading-aloud feature in Microsoft Edge browser. It supports multiple languages, a wide range of voices, and emotional styles.

Advantages

- Extremely fast speech synthesis (RTF < 0.1)

- Supports multiple voice styles

Limitations

- Does not support voice cloning or fine-tuning
- Requires internet access (no offline deployment)

Deployment

Refer to `edge` service deployment documentation.

Parameters

- `tts_text` (str): Text to be synthesized
- `prompt_text` (str): Official EdgeTTS voice style name
- `prompt_wav` : None

Output

- 24000 Hz synthesized audio in WAV format

Tacotron

Model Overview

Tacotron is an end-to-end text-to-speech model that converts text directly into speech features (e.g., Mel spectrograms), which are then turned into natural audio via vocoders such as WaveNet. The model includes:

1. **Encoder** : Transforms text/phoneme sequences into high-dimensional context representations (includes embedding, CNN, Bi-RNN).
2. **Attention Mechanism** : Aligns input text with output audio steps.
3. **Decoder** : Autoregressively generates Mel spectrograms using RNNs.
4. **Post-Net** : Enhances spectrogram quality for better naturalness.
5. **Vocoder** : Converts spectrograms to waveform using Griffin-Lim, WaveNet, or HiFi-GAN.

Tacotron simplifies the TTS pipeline and significantly improves voice quality.

Advantages

- High synthesis speed (RTF ~ 0.2)

Limitations

- Does not support voice cloning; limited fine-tuning
- Lacks proper sentence boundary handling; can result in unnatural transitions

Deployment

Refer to `tacotron` service deployment documentation.

Parameters

- `tts_text` (str): Text to be synthesized
- `prompt_text` (str): Integer (0–100), controls voice scale (passed as string)
- `prompt_wav` : None

Output

- 24000 Hz synthesized audio in WAV format

GPT-SoVITS

Model Overview

GPT-SoVITS combines GPT’s text generation capabilities with So-VITS's speech synthesis. It accepts text and a short reference voice sample to generate speech mimicking the speaker's timbre. It includes:

1. **GPT Module**: Converts text into intermediate speech features
2. **Speaker Encoder**: Extracts speaker characteristics
3. **So-VITS Decoder**: Synthesizes final audio from features and speaker info

Supports zero-shot voice cloning and multi-speaker synthesis, making it ideal for virtual assistants and AI avatars.

Advantages

- Moderate synthesis speed (RTF ~ 0.5)
- Supports zero-shot voice cloning

Limitations

- Less natural output in short sentences; occasional timbre distortion
- Requires warm-up on cold start

Deployment

Refer to `tacotron` service deployment documentation.

Parameters

- `tts_text` (str): Text to be synthesized
- `prompt_text` (str): Transcription of reference audio
- `prompt_wav` (WAV): Reference audio (≤15s, 24000 Hz)

Output

- 24000 Hz synthesized audio in WAV format

CosyVoice2

Model Overview

CosyVoice2, developed by Alibaba's Tongyi Lab, is a multilingual speech synthesis model supporting Chinese, English, Cantonese, and Japanese. It features zero-shot voice cloning, emotional control, and both streaming and offline modes.

CosyVoice uses LLMs (e.g., Qwen) for semantic encoding, FSQ encoders for high-fidelity modeling, and a hybrid streaming/offline architecture with first-packet latency as low as 150ms — making it suitable for real-time applications.

Advantages

- Excellent voice quality and naturalness
- Supports zero-shot voice cloning

Limitations

- Slower synthesis speed (RTF ~ 0.8)
- Weaker performance for long or continuous outputs

Deployment

Refer to `tacotron` service deployment documentation.

Parameters

- `tts_text` (str): Text to be synthesized
- `prompt_text`: None
- `prompt_wav` (WAV): Reference audio (≤15s, 24000 Hz)

Output

- 24000 Hz synthesized audio in WAV format

Conclusion & Future Outlook

Conclusion

The system features a clear structure and decoupled modules, making it easy to maintain and extend. With a unified entry point and independent model services, users can flexibly invoke different speech task models, offering strong scalability.

Future Outlook

- Support continuous conversation via WebSocket.
- Add GPU resource scheduling and dynamic load balancing.
- Integrate more language and voice style models.

Lip-sync & RTC

Module Functions

LiveTalking is a real-time digital human interaction system that leverages audio/video processing and deep learning technologies to achieve the following:

1. **Real-time audio-video processing**: Converts live audio input into a talking digital avatar
2. **Digital avatar generation**: Creates personalized digital humans based on video materials
3. **WebRTC streaming**: Enables low-latency audio/video transmission in real time
4. **Multi-session support**: Allows simultaneous access by multiple users

Input and Output

Input

- Video files: Used to create digital human avatars
- Text input: Sent via API to trigger speech generation
- Audio input: Received from the TTS module to drive lip-sync video generation

- WebRTC signaling: Handles audio/video connection requests from clients

Output

- Static resource files: Avatar images and configuration files
- Real-time audio-video stream: Avatar video stream delivered via WebRTC
- API responses: For avatar creation, switching, and other service endpoints

Dependencies

Core Libraries and Frameworks

- **Audio/Video processing**:
  - `aiortc` : WebRTC implementation
  - `aiohttp` : Asynchronous HTTP server
  - `flask` : Web server framework
- **Deep learning**:
  - `PyTorch` : Model inference
  - `torch.multiprocessing` : Multiprocessing support
- **Image processing**:
  - `OpenCV` : Used in lower-level modules
  - `NumPy` : Numerical computation
- **Others**:
  - `fastapi` : API server
  - `uvicorn` : ASGI server
  - `flask_sockets` : WebSocket support

System Requirements

- Python 3.8+
- CUDA support (for GPU acceleration)
- ffmpeg (for video processing)

Model Introduction – MuseTalk

MuseTalk is a lip-sync model specifically designed for real-time digital avatars. It features:

1. **Low-latency inference**: Suitable for live streaming and interactive use cases
2. **High-precision lip synchronization**: Generates accurate lip movements from speech features, supporting multiple languages
3. **Lightweight architecture**: Requires relatively low computational resources and runs smoothly on standard GPUs

MuseTalk Workflow

1. Converts audio input into feature vectors

2. Maps features to facial animation parameters using a neural network
3. Generates facial expressions and lip motions from the parameters
4. Merges facial animations with avatar images

Technical Details

System Architecture

LiveTalking follows a layered architecture:

1. **API Layer**: Manages avatars and service control endpoints
2. **Core Processing Layer**: Handles audio/video input and renders the digital human
3. **Model Layer**: Loads and runs the MuseTalk model for lip-sync generation

Session Management

Each user session (identified by a session ID) has its own:

- Audio/video processing channel
- WebRTC connection
- Avatar state

WebRTC Implementation

WebRTC is implemented using the `aiortc` library:

- ICE servers handle NAT traversal
- WHIP protocol is supported for low-latency publishing
- Custom `MediaTrack` handles frame-level audio/video processing

Multiprocessing & Memory Optimization

- Uses `torch.multiprocessing` for multi-process inference
- Employs `spawn` to avoid CUDA context conflicts
- GPU memory optimized for performance
- Asynchronous processing prevents blocking the main thread

Avatar Creation Workflow

1. **Video processing**: Extract keyframes from uploaded video
2. **Face detection**: Identify and crop face regions
3. **Resource generation**: Create all necessary files for the avatar
4. **Configuration saving**: Generate config files for future loading

Background Processing for Avatars

1. **Semantic segmentation**: Uses the PPHumanSeg model for accurate foreground-background separation
2. **Edge smoothing**: Applies Gaussian blur to improve boundary transitions
3. **Audio-preserving video composition**: Merges processed video with original audio using FFmpeg
4. **Service deployment**: Deployed as a microservice using Jina, supporting RESTful API calls for easy integration

Performance Optimization

- **Model warm-up**: Ensures no lag on first-time inference
- **Batch processing**: Improves GPU utilization
- **Optimized video encoding**: Supports both H.264 and VP8 codecs

---

Model Management Server

Module Functions

- Manages all TTS and Lip-sync models
- Supports model startup, shutdown, configuration updates, and status queries
- Enables binding of Tutor Face and Tutor Voice into a unified Avatar
- Supports adding, deleting, and modifying Avatars

Input and Output

Input

- Model control commands
- Parameter configurations, video material, and optional audio samples for Avatar creation
- Configuration content to be modified for models or avatars

Output

- List of models with configuration and status information
- List of Avatars with status information
- Preview images of Avatars

Technical Details

Model management involves interaction with the following configuration files:

- `model_info`: Stores the list of supported TTS models and their metadata
- `config.json`: Stores the current runtime configuration of the active TTS service
- `avatar_info.json`: Stores the list of supported Avatars and their metadata

TTS Model Configuration (`model_info`)

Operations such as model querying, editing, and launching involve reading and writing `model_info`. Key fields include:

```
{
    "tacotron": {
        "full_name": "TacoTron2",
        "clone": false,
        "license": "MIT",
```

```json
        "env_path": "/home/chengxin/workspace/chengxin/conda/envs/taco2",
        "server_path": "/workspace/chengxin/tts/taco/taco_server.py",
        "status": "active",
        "timbres": [
            "Default",
            "MaleA",
            "MaleB",
            "FemaleA",
            "FemaleB"
        ],
        "cur_timbre": "Default"
    },
    ...
}
```

- `env_path` : Path to the virtual environment associated with the model
- `server_path` : Path to the TTS server script of the model

If `clone` is set to `false` , additional fields include:

- `timbres` : A list of supported timbre styles for the model
- `cur_timbre` : Currently active timbre

TTS Runtime Control (`config.json`)

Used when starting or stopping TTS models. The process is as follows:

```json
{
    "tts_server_port": 5033,
    "tts_server_use_gpu": true,
    "current_tts_server": "edgeTTS",
    "tts_server_pid": 3514099,
    "tts_timbre": "FemaleA"
}
```

- When the server receives a **model shutdown** command, it retrieves the process ID running on the port specified by `tts_server_port` , and terminates it.
- When the server receives a **model start** command, it first terminates the currently running TTS service process (if any), then starts the new model, and updates all relevant fields in `config.json` accordingly.

Avatar Management (`avatar_info.json`)

Avatar creation, parameter retrieval, and deletion all involve reading and writing `avatar_info.json` . Key fields include:

```json
{
    "test_avatar": {
        "description": "This is a test Avatar",
        "status": "active",
        "prompt_face": "/workspace/chengxin/tts/data/avatars/test_avatar_face.mp4",
        "avatar_preview": "/workspace/share/yuntao/LiveTalking/data/avatars/test_avatar/full_imgs/00000000.png",
        "avatar_blur": true,
        "avatar_model": "MuseTalk",
        "clone": false,
        "tts_model": "edgeTTS",
        "timbre": "Default",
        "prompt_voice": "/workspace/chengxin/tts/data/voice/test_avatar_voice.wav"
    },
    ...
}
```

- `prompt_face` : File path to the reference video used for generating the Tutor Face
- `avatar_preview` : File path to the preview image of the Avatar
- `prompt_voice` (optional): File path to the reference audio used for the Tutor Voice

Deployment Method Instructions

Lip-sync Model Deployment

Technical Framework

LiveTalking is built using the following technology stack:

- **Backend Framework**: Dual-service architecture based on Flask and aiohttp
- **Audio & Video Processing**: `aiortc` handles WebRTC communication, and `FFmpeg` is used for video processing
- **Deep Learning Framework**: `PyTorch` for model inference with CUDA acceleration support
- **Service Protocols**: Supports both HTTP API and WebSocket/WebRTC streaming

Server Ports

Server Ports

Service	Port	Description
WebRTC Main Service	8105	Default main service port, provides WebRTC and HTTP API
Background Service	23002	Background blur/replacement service interface
API Service	20000	General management API port

Environment Requirements

- **GPU**: NVIDIA GPU with CUDA 11.3+ and at least 24GB of VRAM
- **Network**: Stable network connection with an upstream bandwidth of at least 5 Mbps

Environment Setup

Configure the LiveTalking Main Module

```
# create conda env
conda create -n nerfstream python=3.10
conda activate nerfstream

#install pytorch, choose the install cmd according to your cuda version by
#<https://pytorch.org/get-started/previous-versions/>
conda install pytorch==2.5.0 torchvision==0.20.0 torchaudio==2.5.0 pytorch-cuda=12.4 -c
pytorch -c nvidia

#install moudle for MuseTalk
conda install ffmpeg
pip install --no-cache-dir -U openmim
mim install mmengine
mim install "mmcv==2.1.0" #Compilation here will take up a lot of CPU, pay attention to
monitor CPU usage
mim install "mmdet==3.2.0"
mim install "mmpose>=1.1.0"

#download model file
```

After installation, modify the lip-sync.json file:

- Change `conda_init` to the Conda activation script,
- Set `conda_env` to the path of the nerfstream environment,
- Set `working_directory` to the directory where the lip-sync module is located.

Configure the video background blur module

```
#use nerfstream
conda activate nerfstream
cd burr
#download human_segmentation_pphumanseg_2023mar.onnx
# from https://github.com/opencv/opencv_zoo/tree/main/models/human_segmentation_pphumanseg
```

Configure the MuseTalk Module

```
#get musetalk
git clone https://github.com/TMElyralab/MuseTalk.git
cd MuseTalk

# create a conda env according to guide in musetalk
```

After installation, modify the lip-sync.json file:

- Set `muse_conda_env` to the path of the Conda environment required by the MuseTalk project,
- Set `ffmpeg_path` to the location of ffmpeg,
- Set `muse_talk_base` to the installation path of the MuseTalk project.

Configure the Video Background Blur Module

```
#use nerfstream
conda activate nerfstream
cd burr
#download human_segmentation_pphumanseg_2023mar.onnx
# from https://github.com/opencv/opencv_zoo/tree/main/models/human_segmentation_pphumanseg
```

How to start

Start burr server

```
# start burr_server, if port 23002 is bind, use other port
cd burr
python burr_server.py
```

Start api server

```
python live_server.py
```

TTS Model Deployment

Technical Framework

The TTS system is built with the following technologies:

- **Languages & Libraries**: Python 3, PyTorch
- **Web Frameworks**: FastAPI, Matcha

Server Ports

In the current version, to save GPU memory resources, only **one TTS service** can be active at a time.
The TTS service is launched by the **Model Management Server**, and the port can be specified via API parameters during startup.

- **Default Port**: `5033`

Environment Requirements

- **EdgeTTS**
  - Python 3.9 or higher
  - CPU-based
- **Cosyvoice2**
  - Python 3.10
  - GPU with ≥ 6GB VRAM
  - CUDA 11.8 or higher
- **Tacotron2**
  - Python 3.9

    - GPU with ≥ 2GB VRAM
    - CUDA 11.8 or higher
- **GPT-soViTs**
    - Python 3.9
    - GPU with ≥ 4GB VRAM
    - CUDA 11.8 or higher

Environment Setup

Currently, the four supported TTS models/APIs must each run in **separate virtual environments**.
Detailed configuration instructions can be found in:

- The `env.sh` file inside each model's folder
- The `requirement.txt` under `tts/requirements/MODEL_NAME/`

```
export MODEL_NAME=cosyvoice
pip install -r ../requirements/cosyvoice/requirements.txt
echo "Environment for cosyvoice is set up."
```

How to Start

1. Locate the `server.py` file in each model's directory, and copy its path to the `server_path` field for the corresponding model in the `tts/model_info.json` file.
2. Copy the path of the virtual environment used to run each model into the `envpath` field for the corresponding model in the same `model_info.json` file.
3. **Do not manually start or stop** any TTS models.
   All TTS model lifecycle operations (start/stop) are managed by the **Model Management Module**.

TTS servers test demo

```
# ========= 3. Test Model Startup =========
# cosyvoice
curl -X POST http://127.0.0.1:8204/tts/start \
  -F "model_name=cosyvoice" \
  -F "port=5033" \
  -F "use_gpu=true"

# edgeTTS
curl -X POST http://127.0.0.1:8204/tts/start \
  -F "model_name=edgeTTS" \
  -F "port=5033" \
  -F "use_gpu=true"

# tacotron
curl -X POST http://127.0.0.1:8102/tts/start \
  -F "model_name=tacotron" \
  -F "port=5033" \
  -F "use_gpu=true"

```

```
# sovits
curl -X POST http://127.0.0.1:8204/tts/start \
  -F "model_name=sovits" \
  -F "port=5033" \
  -F "use_gpu=true"

# ========= Test Model Voice Switching =========
curl -X POST http://127.0.0.1:8102/tts/choose_timbre \
  -F "model_name=edgeTTS" \
  -F "timbre=en-US-GuyNeural"

curl -X POST http://127.0.0.1:8102/tts/choose_timbre \
  -F "model_name=edgeTTS" \
  -F "timbre=en-US-EmmaNeural"

curl -X POST http://127.0.0.1:8102/tts/choose_timbre \
  -F "model_name=tacotron" \
  -F "timbre=55"

# ========= Test TTS Generation =========
curl -X POST http://127.0.0.1:8102/tts/response \
  -F "tts_text=Hello this is TutorNet speaking. Do you want a cup of coffee?" \
  -F "prompt_text=The course name COMP9331 is simply compained 3331." \
  -F "prompt_wav=@ref.wav" \
  --output output.pcm

# Play raw PCM audio (note the specified parameters)
sox -t raw -r 24000 -e signed -b 16 -c 1 output.pcm output.wav

# Test avatar creation
curl -X POST http://127.0.0.1:8204/avatar/add \
  -F "name=test_avatar_2" \
  -F "avatar_blur=true" \
  -F "support_clone=false" \
  -F "timbre=Default" \
  -F "tts_model=edgeTTS" \
  -F "avatar_model=MuseTalk" \
  -F "description=Avatartest2" \
  -F "prompt_face=@data/ref/ref.mp4" \
  -F "prompt_voice=@data/ref/ref.wav"

curl -X POST http://127.0.0.1:8204/tts/start \
  -F "model_name=cosyvoice" \
  -F "port=5033" \
  -F "use_gpu=true"

curl -X POST http://127.0.0.1:8204/avatar/start \
  -F "avatar_name=test_avatar"

# ======== Directly Test tts Server (need to switch env) ========
conda activate edge
python edge_server.py --model_name edgeTTS --port 8102 --use_gpu True

curl -X POST http://127.0.0.1:8102/generate \
  -F "tts_text=Hello this is TutorNet speaking. Do you want a cup of coffee?" \
  -F "prompt_text=en-US-GuyNeural" \
  -F "prompt_wav=@ref.wav" \
  -o output.wav

conda activate cosyvoice
python taco_server.py --model_name tacotron --port 8102 --use_gpu True

curl -X POST http://127.0.0.1:8102/generate \
  -F "tts_text=Hello this is TutorNet speaking. Do you want a cup of coffee?" \
  -F "prompt_text=40" \
  -F "prompt_wav=@ref.wav" \
  -o output.wav

conda activate sovits
python so_server.py --model_name sovits --port 8102 --use_gpu True

curl -X POST http://127.0.0.1:8102/generate \
  -F "tts_text=Hello this is TutorNet speaking. Do you want a cup of coffee?" \
  -F "prompt_text=The course name COMP9331 is simply compained 3331." \
  -F "prompt_wav=@ref.wav" \
  -o output.wav
```

Model Management Server Deployment

Technical Framework

- **Language & Framework**: Python 3.9+, FastAPI

Server Port

- **Default Port**: `8204`

Environment Requirements

- Python 3.8 or higher
- Required Python libraries are listed in `tts/requirements/server/requirement.txt`

Environment Setup

```
cd ./requirements/server
pip install -r requirement.txt
```

How to Start

1. start TTS Management server

```
cd tts
uvicorn tts:app --host 0.0.0.0 --port 8204
```