SyncInterview

A

Major Project (IIS4270) Report

Submitted in the partial fulfillment of the requirement for the award of

Bachelor of Technology

in

Computer Science and Engineering
By:

Name: Manan Bajaj
Reg No.: 219311042
Section: A

Under the supervision of:

Dr. Vijay Kumar Sharma

Type of project: External

05/2025

Department of IoT and Intelligent Systems
School of Computing and Intelligent Systems (SCIS)
Manipal University Jaipur
VPO. Dehmi Kalan, Jaipur, Rajasthan, India – 303007
Bachelor of Technology

Department of IoT and Intelligent Systems
School of Computing and Intelligent Systems (SCIS), Manipal University Jaipur,
Dehmi Kalan, Jaipur, Rajasthan, India- 303007

STUDENT DECLARATION

*I hereby declare that this project (**SyncInterview**) is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the University or other Institute, except where due acknowledgements has been made in the text.*

Place: Manipal University Jaipur
Date: 05/05/2025

Manan Bajaj
(219311042) |Section: A
B.Tech (IOT & IS) 8th Semester

Department of IoT and Intelligent Systems
School of Computing and Intelligent Systems (SCIS), Manipal University Jaipur,
Dehmi Kalan, Jaipur, Rajasthan, India- 303007

Date: 15/05/2025

CERTIFICATE FROM SUPERVISOR

*This is to certify that the work entitled "**SyncInterview"** submitted by **Manan Bajaj**(219311042) to **Manipal University Jaipur** for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** is a bonafide record of work carried out by him under my supervision and guidance from 15/01/2025 to 10/05/2025*

Dr. Vijay Kumar Sharma
Department of IoT and Intelligent Systems
Manipal University Jaipur

ACKNOWLEDGMENTS

I express my sincere gratitude to Dr.Geeta Rani, Head of the Department of Internet Of Things AND Intelligent Systems, for facilitating a conducive research environment. I am deeply thankful to my project supervisor, Dr. Vijay Kumar Sharma, for their patient guidance and consistent support throughout the development of this project.

I also acknowledge the contributions of the open-source community whose tools—Groq, FastAPI, Celery, Redis, and others—were integral to this work. Special thanks to my peers and testers whose feedback improved SyncInterview's performance.

ABSTRACT

This report details the design and implementation of a robust backend system for an AI-powered interview application. The application aims to provide users with a precise, live interview experience by analyzing their resume against a target job description. The core challenge addressed is enabling a seamless, real-time conversational flow with a Large Language Model (LLM), despite the inherent latency in LLM generation. The proposed architecture utilizes FastAPI for API endpoints, WebSockets for live communication, and Celery for asynchronous task processing. A key innovation is the real-time processing strategy for user speech, where transcribed text is chunked and sent to the LLM every two seconds, prompting partial or tentative responses. Upon user pause, the final accumulated text is sent for a definitive response, prioritizing the last available output if the final generation is delayed. The system incorporates background tasks for document parsing (JD and Resume analysis), a parallel mini-LLM for conversational fillers and surprises, and an analysis module utilizing LLM agents and function calling for evaluating user responses, including code snippets for technical roles. The report covers the system architecture, detailed design of key components, the specific real-time interaction logic, task management using Celery, data handling, and the analysis framework. The provided file structure serves as the blueprint for the modular implementation. The report concludes with a summary of achievements and potential future enhancements

LIST OF FIGURES

LIST OF TABLES

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 PROJECT BACKGROUND

In today's competitive job market, effective interview preparation is crucial for candidates. Traditional methods often involve mock interviews conducted by humans, which can be resource-intensive and lack consistency. The advent of Artificial Intelligence, particularly Large Language Models (LLMs), presents an opportunity to automate and enhance the interview preparation process. An AI interviewer can offer personalized feedback, simulate real-world scenarios, and provide objective analysis based on specific job requirements and candidate profiles. This project focuses on building a robust backend system for such an application, capable of handling document analysis, managing live conversational interviews, and providing detailed post-interview analysis. The core challenge lies in enabling a natural, low-latency interaction experience with the LLM during the live chat, simulating a human interviewer's responsiveness despite the computational overhead of AI model inference.

1.2 PROBLEM STATEMENT

Developing an AI interview application requires a backend that can process user documents (resume and job description), conduct a dynamic, question-answering interview in real-time via a conversational interface (like a chat or voice-to-text/text-to-voice), handle potential latencies from AI models, and provide comprehensive post-interview analysis. Specifically, the backend must manage:

1. Secure and efficient upload and processing of sensitive user documents.
2. Analysis of the job description and resume to tailor interview questions.
3. A real-time communication channel for the interview conversation.
4. A mechanism to interact with an LLM interviewer in a responsive manner, even when receiving partial or streaming user input.
5. Integration of background tasks for computationally intensive operations (document parsing, LLM calls, analysis).
6. Incorporation of specialized AI capabilities, such as code analysis using agents, for technical roles.
7. Generation of meaningful feedback and analysis based on the interview performance. The primary technical challenge is designing an architecture that can process streaming

transcription data in real-time, interact with an LLM, and deliver responses with minimal perceived delay, creating a "live" chat experience.

1.3 PROJECT OBJECTIVES

The main objectives of this project are:

1. Design and implement a scalable and robust backend architecture for an AI interview application using modern web frameworks and asynchronous processing.
2. Develop API endpoints for secure document (Job Description and Resume) upload and initiation of interview sessions.
3. Implement a WebSocket-based communication layer to facilitate real-time, bi-directional chat during the interview.
4. Engineer a sophisticated real-time LLM interaction mechanism that processes user speech transcription in chunks, generates tentative responses periodically, and provides the latest available response instantaneously upon user pause.
5. Integrate an asynchronous task queue (Celery) to handle background processes such as document parsing, LLM calls, and analytical tasks.
6. Incorporate a secondary LLM or process running in parallel for generating conversational fillers or 'surprises' to enhance the natural feel of the interview.
7. Develop modules for pre-interview analysis (JD-Resume matching) and post-interview performance evaluation.
8. Implement a code analysis feature using LLM agents and function calling to evaluate programming responses for relevant job roles.
9. Structure the codebase following best practices for maintainability and modularity, aligning with the provided file structure.

1.4 SCOPE OF THE PROJECT

The scope of this project is limited to the backend development of the AI interview application. This includes:

1. API development for document handling and interview session management.
2. WebSocket implementation for the live chat interface.
3. Integration with document parsing libraries (though specific implementation details might be abstracted).
4. Integration with LLM providers (e.g., OpenAI), focusing on API interaction patterns.
5. Implementation of the real-time chunking and response logic within the backend.

6. Setting up and utilizing Celery workers and a message broker.
7. Designing and implementing the interview state management logic.
8. Developing the mini-LLM filler/surprise generation logic.
9. Designing and implementing the pre- and post-interview analysis logic, including the structure for code analysis.
10. Defining data models for documents, interview sessions, messages, and analysis results.

Setting up a basic local development environment using Docker.

The project scope excludes:

1. Front-end development (user interface, audio recording, speech-to-text on the client side).
2. Speech-to-Text (ASR) or Text-to-Speech (TTS) model training or fine-tuning (integration points will be designed).
3. Deployment to a production cloud environment (conceptual deployment will be discussed).
4. Comprehensive error handling for all possible external API failures.
5. Advanced security features beyond basic API security best practices.

CHAPTER 2: LITERATURE REVIEW

2.1 OVERVIEW OF AI IN RECRUITMENT

Artificial Intelligence is increasingly being adopted across various stages of the recruitment process, from sourcing and screening to interviewing and onboarding [1]. AI-powered tools can analyze large volumes of resumes, identify potential candidates based on keywords and skills, and automate initial screening questions. AI interview platforms, the focus of this project, represent a significant advancement, offering standardized, unbiased (when designed carefully), and scalable methods for evaluating candidates [2]. These platforms can conduct interviews via text chat, voice, or even video, asking questions tailored to the specific role and company culture. The backend systems supporting such platforms require sophisticated capabilities to handle natural language understanding, real-time interaction, and complex data analysis.

2.2 TEXT NLP

NLP is a core technology enabling the backend to understand and process human language from resumes, job descriptions, and interview transcripts [3]. Techniques such as tokenization, part-of-speech tagging, named entity recognition, sentiment analysis, and topic modeling are essential for extracting structured information from unstructured text documents like resumes and job descriptions. Similarity metrics (e.g., cosine similarity on TF-IDF or embedding vectors) can be used for JD-Resume matching [4]. During the interview, NLP is used for transcribing speech to text (often handled by external ASR services) and for the LLM to understand the nuances of the candidate's responses

2.3 LLMS AND CONVERSATIONAL AI

LLMs like GPT (Generative Pre-trained Transformer) models are foundational for generating human-like text and engaging in conversational exchanges [5]. Their ability to understand context, generate coherent responses, and follow instructions makes them ideal for simulating an interviewer. Prompt engineering is a critical aspect of utilizing LLMs effectively, involving carefully crafting input prompts to guide the model's behavior, persona, and response style [6]. For an AI interviewer, prompts need to incorporate context (JD, Resume, conversation history), define the desired tone (professional, objective), and guide the question generation process. The

challenge in real-time applications is the latency associated with generating responses, requiring strategies to provide timely feedback while waiting for the full generation.

2.4 REAL TIME CONVERSATION

Real-time interaction, crucial for a live interview experience, necessitates protocols capable of low-latency, bi-directional communication between the client and the server. WebSockets provide a full-duplex communication channel over a single TCP connection, making them significantly more efficient for persistent, real-time data exchange than traditional HTTP request/response cycles [7]. This technology is well-suited for sending continuous streams of transcription data from the client and receiving LLM responses or control messages from the server with minimal delay.

2.5 ASYNCHRONOUS TASK QUEUES (CELERY)

Many operations in the backend, such as parsing large documents, making potentially slow external API calls to LLMs, or running complex analysis, are time-consuming and blocking. Performing these directly within the request/response cycle of the web server would lead to poor performance and scalability issues. Asynchronous task queues, such as Celery, allow these tasks to be offloaded to background worker processes [8]. The web server can quickly return a response (e.g., "processing started") while the worker handles the heavy lifting. This pattern improves responsiveness, allows for retries on failure, and enables scaling specific parts of the system independently. Celery requires a message broker (like Redis or RabbitMQ) to manage the queue of tasks and a backend (like Redis or a database) to store task results.

2.6 DOCUMENT PROCESSING TECHNIQUES

Processing documents like PDFs, DOCX, or plain text requires libraries capable of extracting text and potentially structural information [9]. Libraries such as python-docx, PyPDF2, or textract can be used. Once text is extracted, NLP techniques (as discussed in Section 2.2) are applied to identify key sections (experience, education, skills), extract entities (company names, dates, job titles), and prepare the data for analysis and prompting the LLM. This process is often error-prone due to variations in document formatting, necessitating robust parsing logic and potential fallback mechanisms.

2.7 CODE ANALYSIS TECHNIQUES

Processing documents like PDFs, DOCX, or plain text requires libraries capable of extracting text and potentially structural information [9]. Libraries such as python-docx, PyPDF2, or textract can be used. Once text is extracted, NLP techniques (as discussed in Section 2.2) are applied to identify key sections (experience, education, skills), extract entities (company names, dates, job titles), and prepare the data for analysis and prompting the LLM. This process is often error-prone due to variations in document formatting, necessitating robust parsing logic and potential fallback mechanisms.

CHAPTER 3: SYSTEM ARCHITECTURE

3.1 OVERALL SYSTEM DESIGN

The backend system is designed as a modular architecture to ensure scalability, maintainability, and separation of concerns. At its core, it is built around a FastAPI web server handling incoming HTTP requests and WebSocket connections. Asynchronous task processing is managed by Celery, which communicates via a message broker and stores results in a backend. Various services encapsulate interactions with external APIs (LLMs, potentially ASR/TTS) and internal logic (document parsing, storage, state management). A dedicated analysis module handles pre- and post-interview evaluations. The file structure provided serves as the blueprint for organizing these components into logical directories and modules.

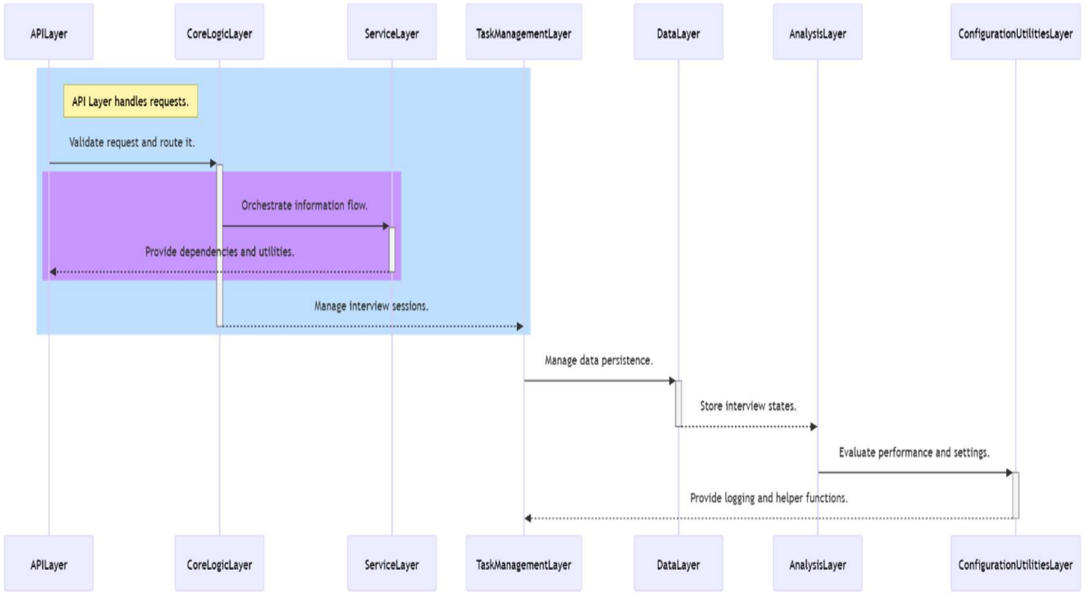


Fig 1 Layers of system

- API Layer: Exposes functionality via RESTful endpoints and WebSocket connections. Handles request validation and routes requests to the appropriate core logic or triggers background tasks.
- Core Logic Layer: Manages the state of interview sessions, orchestrates the flow of information, and implements the complex real-time interaction logic.
- Service Layer: Provides abstract interfaces for external dependencies and internal utilities (LLM interaction, document processing, storage).
- Task Management Layer: Defines and orchestrates asynchronous tasks using Celery, including message queuing and worker processes.

- Data Layer: Manages data persistence (temporary file storage, potentially a database for interview states or analysis results).
- Analysis Layer: Contains the logic for evaluating documents and interview performance, including the code analysis component.
- Configuration & Utilities Layer: Handles application settings, logging, and helper functions.

3.2 COMPONENT BREAKDOWN

Based on the provided file structure, the system components are organized as follows:4

3.2.1 API Layer

1. API Layer Located in `app/api/`, this layer contains versioned API endpoints (`app/api/v1/endpoints/`).
2. documents.py`: Handles document upload (/documents/upload/jd, /documents/upload/resume`). Triggers document processing tasks.
3. interview.py`: Handles interview session creation (`/interview/start`) and manages the real-time chat via a WebSocket (`/interview/{interview_id}/cha`).
4. dependencies.py`: Contains dependency injection functions (e.g., get_settings`, get_interview_manager`) for FastAPI.

3.2.2 Core Business Logic

1. Located in `app/core/`, this directory contains the central logic for managing interview sessions.
2. interview_manager.py: The core class managing the lifecycle of interview sessions. It handles starting interviews, receiving WebSocket messages, interacting with Celery tasks and services, and managing interview state.
3. interview_state.py: Defines data structures and logic for storing the state of an active interview session (e.g., conversation history, JD/Resume data, WebSocket connection reference).
4. exceptions.py: Defines custom exceptions for the core logic (e.g., `InterviewNotFound`, `InvalidInterviewState`).

3.2.3 Service Layer

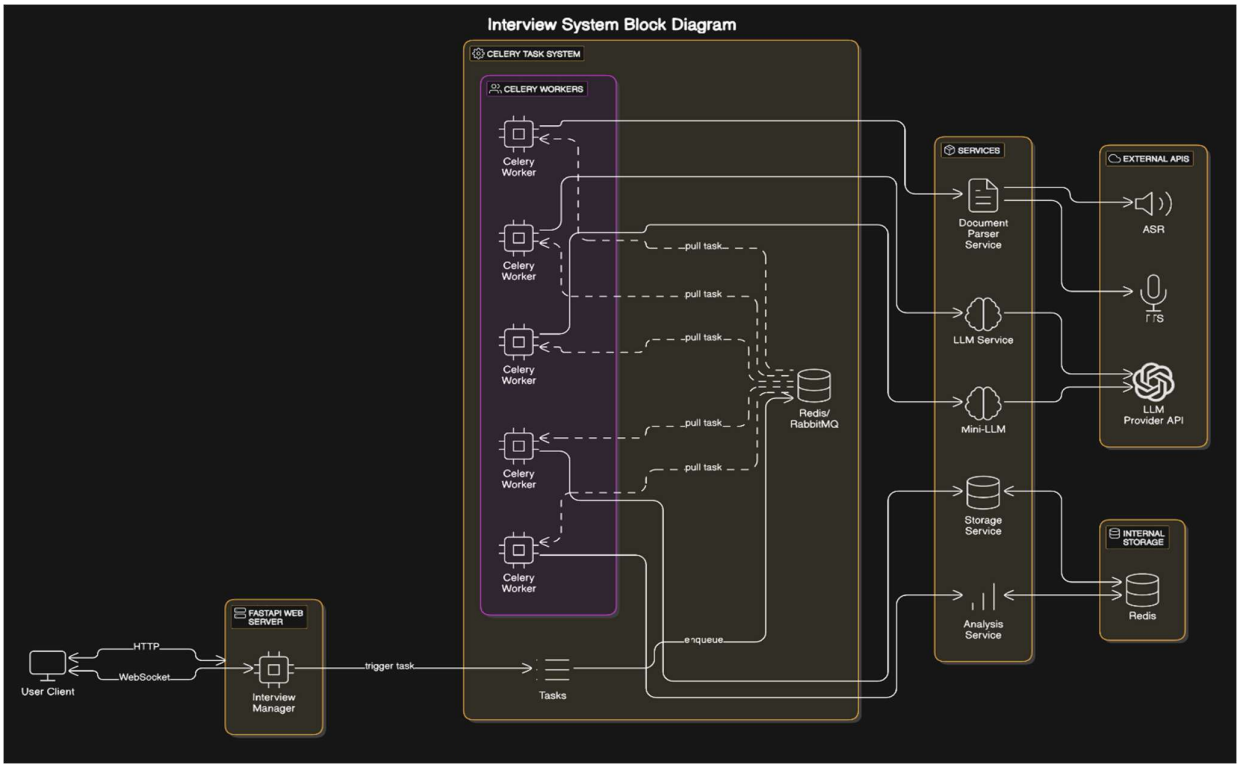


Fig 2 Service Layers

1. llm_service.py`: Handles communication with the primary LLM provider API. Contains methods for generating interview questions, analyzing responses, etc.
2. mini_llm_service.py`: Handles communication with the mini-LLM used for fillers/surprises. Contains methods for generating short, contextually relevant interjections.
3. document_parser.py`: Contains logic for extracting text and potentially structured data from uploaded documents (PDF, DOCX).
4. storage_service.py: Handles file storage operations (saving uploaded documents, potentially storing interview transcripts).
5. audio_processing_service.py`: (Conceptual) Interface for ASR - receives audio/chunks and returns text. The backend focuses on receiving text transcription from the client.
6. tts_service.py`: (Conceptual) Interface for TTS - receives text and returns audio data. The backend sends text to the client, which might use a TTS service.

3.2.4 Task Management Layer

1. Located in `app/tasks/`, this layer defines the Celery tasks.
2. celery.py: Celery application instance and basic configuration.

3. document_tasks.py`: Celery tasks for processing uploaded documents (parsing, analyzing, preparing data for interview). Includes `process_document`.
4. llm_tasks.py`: Celery tasks for calling the primary LLM (e.g., generating the next question, analyzing a final response chunk).
5. interview_tasks.py`: Other general interview-related tasks (e.g., saving session state periodically).
6. analysis_tasks.py`: Celery tasks for running post-interview analysis, including code analysis.

3.2.5 Rest of the Layers

1. Data link layer provides storage service and manages storage state
2. Analysis layer analyses components JD AND Resume based on skills experience mad potential interview topics

3.3 Technology Stack

Table 1 TECHNOLOGY STACK

Category	Technoloy	Purpose
• **Web Framework**	• FastAPI	• High-performance, asynchronous web framework for APIs and WebSockets.
• **Asynchronous Tasks**	• Celery	• Distributed task queue for background processing.
• **Message Broker**	• Redis / RabbitMQ	• Backend for Celery to manage task queues. Redis is often also used for caching or session state.

• **AI Models**	• Large Language Models (Groq Llama 70B GPT)	• Core AI for interview questions, response analysis, conversation flow, code analysis (via agents/function calling).
• **Mini-LLM**	• Smaller/Specialized LLM or custom model	• For generating conversational fillers and surprises.
• **Document Parsing**	• Python Libraries (e.g., python-docx, PyPDF2, textract)	• Extracting text and data from JD/Resume files.
• **File Storage**	• Local Filesystem (for development)	• Storing uploaded documents and potentially transcripts. (Cloud storage in production).
• **Data Validation**	• Pydantic	• Data modeling and validation, integrated with FastAPI.
• **Dependency Mgmt.**	• pipenv / poetry	• Managing project dependencies.
• **Containerization**	• Docker	• Packaging the application and dependencies.
• **Orchestration**	• Docker Compose	• Defining and running multi-container Docker applications (development/testing).
• **Async Operations**	• asyncio	• Python's built-in library for asynchronous programming.

CHAPTER 4: DETAILED DESIGN

4.1 API Design (FastAPI)

FastAPI is chosen for its high performance, asynchronous capabilities, automatic data model validation with Pydantic, and automatic API documentation generation. The API follows a versioned structure (`/v1/`) and is organized into logical groups for documents and interview management.

4.1.1 Document Upload Endpoints

1. Using `Annotated[UploadFile, File()]` to handle file uploads efficiently.
2. Using `Depends(get_settings)` to inject application configuration, promoting modularity
3. Generating a unique filename using `uuid.uuid4()` to prevent collisions in the storage directory. Saving the file asynchronously (`await file.read()`).
4. Triggering a Celery task (`process_document.delay`) immediately after saving the file, offloading the.

Pseudocode for upload_job_description:

PSEUDOCODE

Function upload_job_description(file: UploadFile, settings: Settings):

If file is None or filename is empty:

Raise HTTP 400 Bad Request ("No file uploaded")

upload_dir = join(settings.STORAGE_PATH, "uploads")

Create directory upload_dir if it doesn't exist

file_extension = get_extension(file.filename) unique_filename = "jd_" + generate_uuid() + file_extension

file_path = join(upload_dir, unique_filename)

Try:

Open file_path in write binary mode Read file content asynchronously Write content to file_path Catch Exception as e: Raise HTTP 500 Internal Server Error ("Could not save file: " + str(e))

Trigger Celery task for background processing

task_result = process_document.delay(file_path, "job_description")

Return JSON response: "message": "Job Description uploaded and processing started." "filename": file.filename "stored_path": file_path "task_id": task_result.id End Function

4.1.1 Document Upload Endpoints

The `/interview/start` endpoint is responsible for initiating a new interview session. It expects a JSON request body conforming to the `InterviewStartRequest` Pydantic model, containing the IDs of the processed JD and Resume. Key design aspects:

1. Uses `Depends(get_interview_manager)` to inject the `InterviewManager` instance, centralizing session management.
2. Calls `manager.start_interview`, which contains the core logic for loading document data, performing pre-interview analysis (implicitly), and setting up the initial session state.
3. Returns a 200 OK response containing the generated `interview_id`, which the client will use for the WebSocket connection.
4. Includes basic error handling for failures during the interview initialization process. Pseudocode for upload_job_description:

4.1.3 Live ChatAPI Layer:

Key design aspects:

1. Uses FastAPI's `WebSocket` dependency to manage the connection.
2. Takes `interview_id` as a path parameter to associate the connection with a specific session.
3. Uses `Depends(get_interview_manager)` to access the manager responsible for the interview state.
4. await websocket.accept() establishes the WebSocket connection.
5. The code enters a `while True` loop to continuously receive messages from the client.

6. Incoming messages are expected as text containing JSON, which is validated against the `ChatMessage` Pydantic model. Table 4.1.3 summarizes the expected message types.
7. The received message is passed to `manager.handle_user_input`, which orchestrates the processing and response generation logic, including triggering Celery tasks.
8. Outgoing messages are sent back to the client using `await websocket.send_json`, typically originating from the `InterviewManager` or Celery tasks that have access to the WebSocket reference stored in the session state. Outgoing messages conform to the `ServerMessage` Pydantic model (as defined or imported in the snippet).
9. Error handling includes catching `WebSocketDisconnect`, `InterviewNotFound`, `ValueError` (for JSON/Pydantic issues), and general exceptions. Disconnection or critical errors lead to session cleanup (`manager.deactivate_interview_session`) and potentially closing the WebSocket with an appropriate status code.

Table 2 EMISSION MESSAGE TYPES

Message Type	Sender	Purpose	Payload Content Example
`ChatMessage`	Client	User input (transcribed text chunks or final sentences).	`{ "type": "chunk", "payload": "The quick brown...", "is_final": false, "timestamp": ... }`
`ChatMessage`	Client	User input (final transcribed sentence after pause).	`{ "type": "final", "payload": "The quick brown fox jumps over the lazy dog.", "is_final": true, "timestamp": ... }`
`ChatMessage`	Client	Control message (e.g., user ending interview).	`{ "type": "control", "payload": "end_interview", "is_final": true, "timestamp": ... }`
`ServerMessage`	Server	Primary LLM response (tentative or final).	`{ "type": "llm_response", "payload": "That's an interesting point..." }`
`ServerMessage`	Server	Mini-LLM filler/surprise.	`{ "type": "mini_llm_filler", "payload": "Hmm, fascinating." }`
`ServerMessage`	Server	Interview state updates (e.g., "Analyzing your answer...").	`{ "type": "interview_state", "payload": "processing" }`
`ServerMessage`	Server	Error message.	`{ "type": "error", "payload": "Could not process your input." }`
`ServerMessage`	Server	Interview conclusion signal.	`{ "type": "end", "payload": "Interview concluded." }`

4.2 INTERVIEW MANAGEMENT

The `app/core/` directory houses the heart of the interview process, managed primarily by the `InterviewManager`.

4.2.1 STATE MANAGEMENT

Maintaining the state of each live interview session is paramount. The `InterviewManager` needs to keep track of:

1. The interview ID. References to the parsed Job Description and Resume data.
2. The complete conversation history (Q&A turns).
3. The current buffer of incomplete user speech transcription.
4. A reference to the active WebSocket connection for sending responses.
5. The current state of the interview (e.g., "waiting_for_user_input", "processing_answer", "generating_question").
6. Timestamps for managing the 2-second chunking logic.
7. The last generated tentative response from the LLM for the current user turn.

This state information should be stored efficiently. An in-memory dictionary within the `InterviewManager` class, mapping `interview_id` to an `InterviewState` object (defined in `interview_state.py`), is suitable for handling active WebSocket connections. For persistence across server restarts or multiple worker instances (though the WebSocket connection is tied to a single process), a shared state backend like Redis could be used, although managing WebSocket references in a shared backend is complex. Storing essential, non-connection state in Redis would allow workers to potentially pick up sessions if a process fails (though reconnecting the WebSocket would still be necessary). For this design, we assume state for active sessions is managed in memory by the process handling the WebSocket, while long-term data (transcripts, analysis) is stored elsewhere.

4.2.2 Real-time LLM Interaction Logic: Chunking and Response Strategy

Live Chat Strategy Despite LLM Latency

This backend mechanism enables real-time chat interaction while managing latency in LLM responses. The process, orchestrated by the InterviewManager and executed via Celery tasks, ensures responsiveness through proactive processing.

Process Breakdown:

1. **User Starts Speaking**
    - The client's ASR generates transcription in chunks (every few hundred milliseconds or upon detecting a pause).
    - The client sends these chunks as ChatMessage objects (type="chunk", is_final=false) over the WebSocket.
2. **Accumulate Chunks**
    - The InterviewManager collects these chunks and appends them to a temporary buffer specific to the current user turn.
3. **Periodic Partial Processing (~Every 2 Seconds)**
    - A mechanism checks if 2 seconds have passed since the last triggered LLM task.
    - If the threshold is met:
        - The buffer's current content is processed.
        - A Celery task (llm_tasks.process_chunk_for_tentative_response) is triggered, passing:
            - Buffer content
            - Interview context (history, JD, resume)
            - A flag indicating the user is still speaking and the sentence is incomplete
        - The Celery worker calls the LLM service with a prompt like:
        - "The user is currently speaking. Here is their incomplete response so far: '[Buffer Content]'.
        - Based on the interview context and JD, provide a brief, tentative acknowledgement or continuation

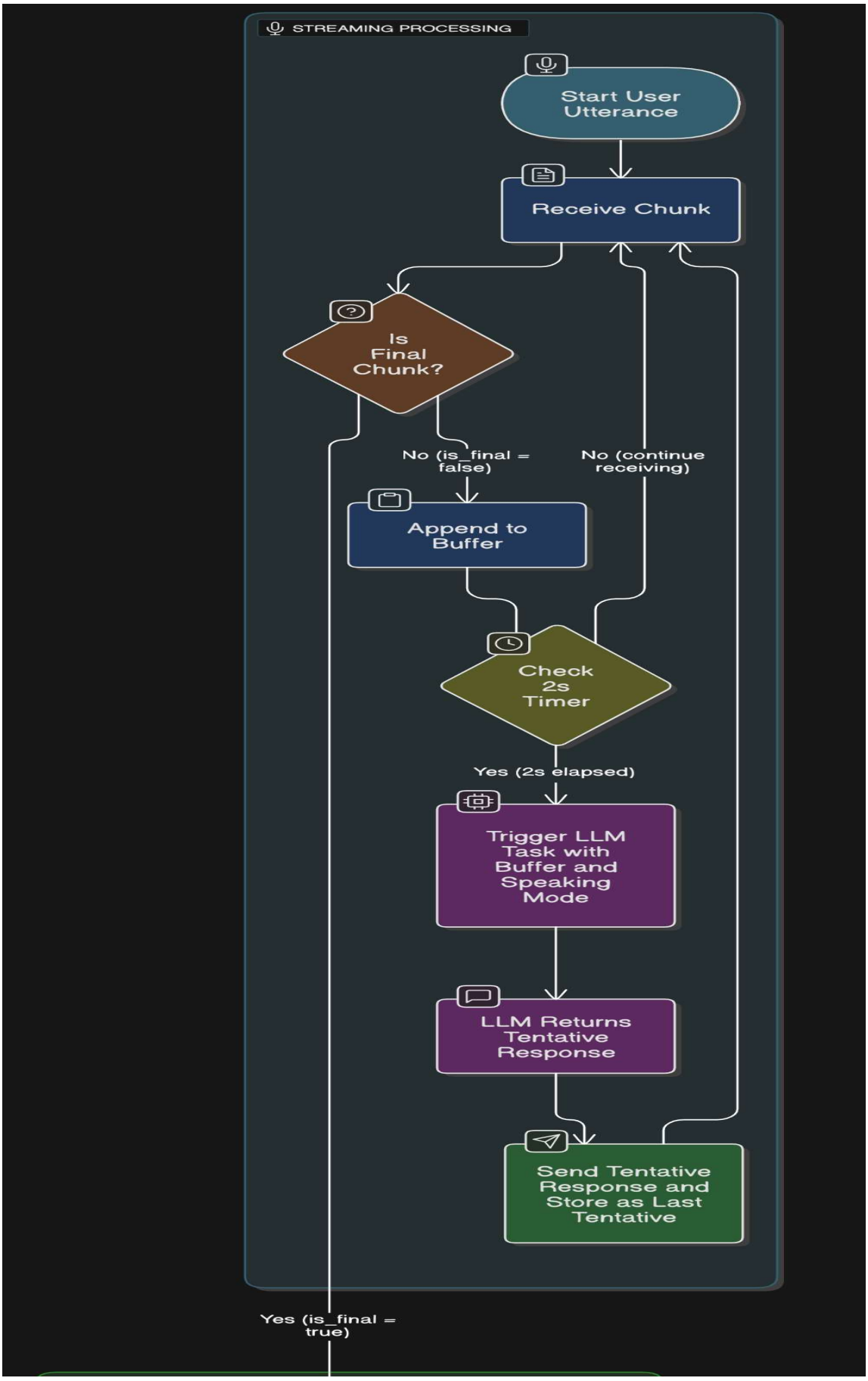
        - phrase, without assuming the sentence is complete."
        - The LLM generates a short, contextually relevant phrase (e.g., "Okay,", "Interesting...", "Right,").
        - The Celery worker sends this tentative response to the InterviewManager, which:
            - Sends the tentative response to the client via WebSocket
            - Stores it as the last_tentative_response for this turn

4. **User Stops Speaking (Final Sentence)**
    - ASR detects a significant pause and sends the final transcribed sentence (type="final", is_final=true).
5. **Trigger Final Processing**
    - The InterviewManager receives the final message, appends it (or replaces the buffer), marks the turn as complete, and triggers:
        - A new Celery task (llm_tasks.process_final_response), passing:
            - The full user utterance
            - Interview context
            - A flag indicating this is the final input for the turn
6. **Handle LLM Latency**
    - llm_tasks.process_final_response begins processing while the InterviewManager waits.
7. **Instantaneous Output Strategy**
    - If the final LLM response arrives **quickly** (within 500ms - 1 second), it is sent directly via WebSocket.
    - If the response is **delayed**, the InterviewManager:

        - Immediately sends the last_tentative_response generated in the chunking phase.
        - Ensures the user receives feedback promptly, even if not the final answer.
        - Discards or logs the delayed final response.

8. **Generate Next Question/Response**
    - Once a response (quick final or fallback tentative) is sent, the InterviewManager:
        - Updates the conversation history.
        - Triggers the process to generate the next question or response (potentially via another Celery task).

This strategy ensures real-time interaction by leveraging partial responses while waiting for full LLM output, maintaining smooth conversational flow.


STREAMING PROCESSING
Start User Utterance
Receive Chunk
Is Final Chunk?
No (is_final = false)
No (continue receiving)
Append to Buffer
Check 2s Timer
Yes (2s elapsed)
Trigger LLM Task with Buffer and Speaking Mode
LLM Returns Tentative Response
Send Tentative Response and Store as Last Tentative
Yes (is_final = true)

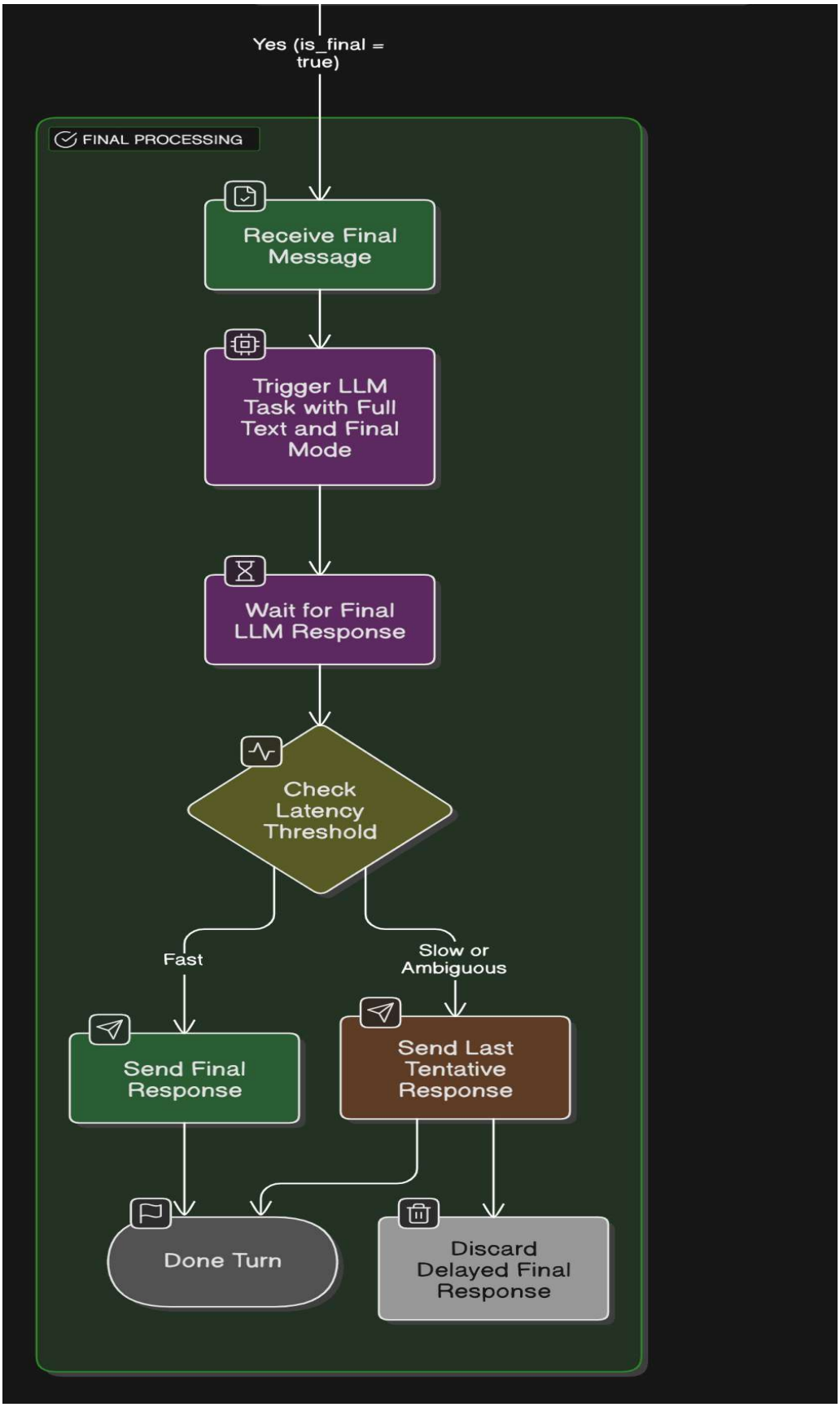


Fig 3 State Management flowchart

4.3 ASYNCHRONOUS TASK MANAGEMENT (CELERY)

Celery is crucial for offloading tasks that are either long-running, blocking (like external API calls), or need to be processed reliably in the background, decoupled from the user's direct request/WebSocket connection.

4.3.1 Celery Configuration

Configuration is handled in `app/config/celery_config.py` and potentially `app/tasks/celery.py`. This involves:

1. Pointing Celery to a message broker (e.g., `redis://localhost:6379/0`).
2. Pointing Celery to a result backend (e.g., `redis://localhost:6379/0`).
3. Defining tasks (auto-discovered or explicitly imported).
4. Setting up serialization (e.g., JSON).
5. Potentially configuring retries, error handling, and routing.

4.3.2 Document Processing Tasks

The `document_tasks.py` file contains tasks like `process_document`. This task receives the file path and document type. Its logic involves:

1. Loading the file content.
2. Using the `DocumentParserService` to extract raw text and potentially structure.
3. Performing initial analysis using the `PreInterviewAnalyzer` (e.g., keyword extraction, identifying sections).
4. Storing the processed data (extracted text, structured info, analysis results) in a way that can be retrieved later by `interview_manager.start_interview`. This might involve saving to Redis, a database, or structured files.

.

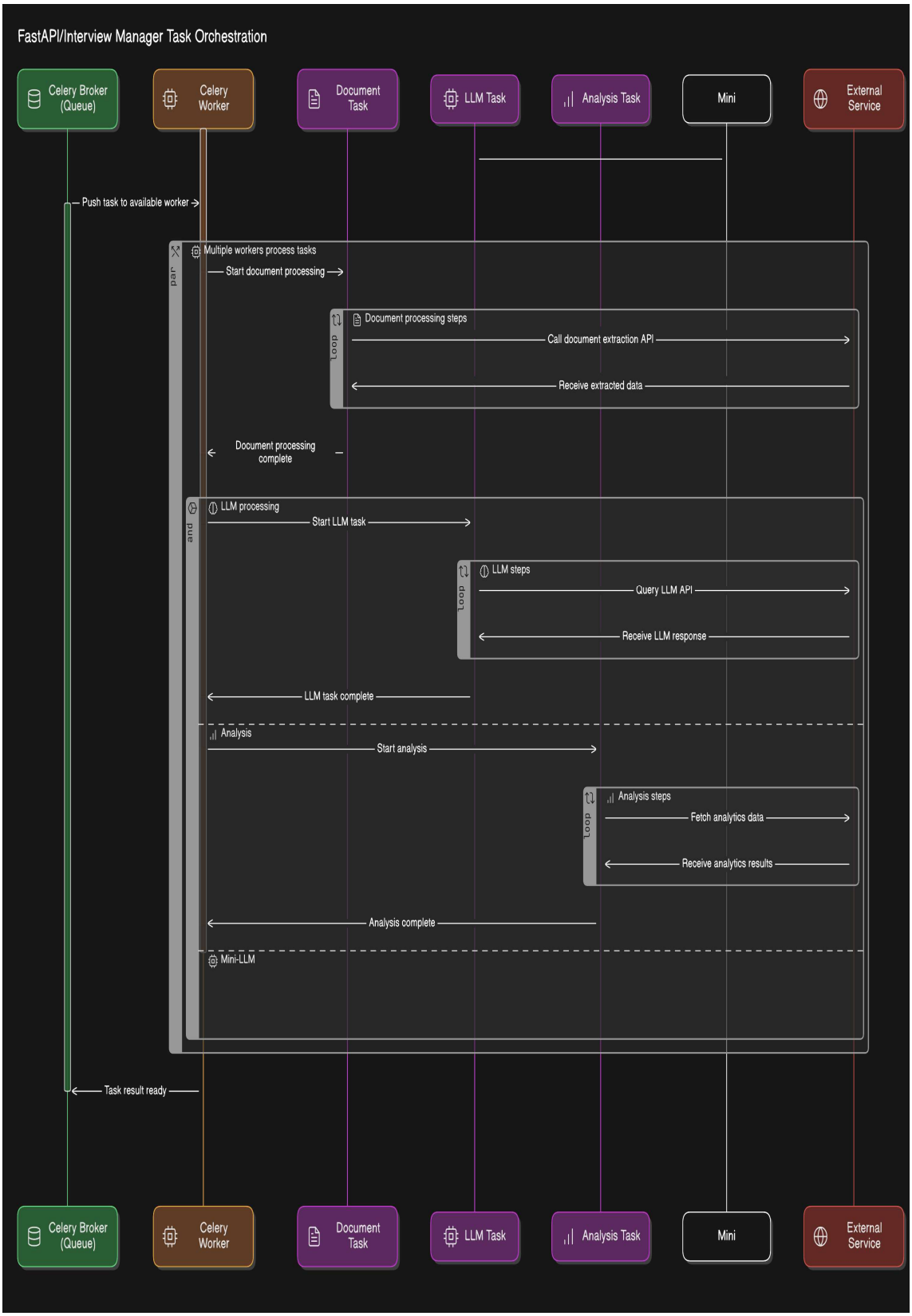


Figure 4 Document Processing Tasks

4.3.3 LLM Interaction Tasks

1. `llm_tasks.py` contains tasks responsible for interacting with the primary LLM. `process_chunk_for_tentative_response`: Called frequently during user speaking. Takes a text chunk (buffer), context, and calls
2. `LLMService.generate_response` with a 'user is speaking' prompt. Returns a brief, tentative response. As discussed, the Manager handles sending this response.
3. `process_final_response`: Called when the user stops speaking. Takes the full transcribed text, context, and calls
4. `LLMService.generate_response` with a 'user finished speaking' prompt. Returns the definitive LLM response. The Manager handles sending this based on timing. Other potential tasks:
5. `generate_initial_question`, `generate_next_question`, `analyze_answer_deeply`.

CHAPTER 5: CONCLUSION

5.1 CONCLUSION

This report presents the detailed design for a robust backend system capable of powering an AI interview application with a focus on a live, responsive conversational experience. The project is a blend of modular architecture based on FastAPI and Celery, leveraging asynchronous processing for performance and scalability. Designing API endpoints for core functionalities like document upload and interview initiation. It includes Implementing a WebSocket layer for real-time communication during the interview, and Developing a specific, innovative strategy for real-time LLM interaction that handles streaming transcription chunks and prioritizes instantaneous feedback through tentative responses and a fallback mechanism for final responses. Integration of asynchronous tasks for document processing, LLM calls, mini-LLM fillers, and comprehensive analysis. Designing modules for pre-interview analysis (JD-Resume matching), post-interview performance evaluation, and specialized code analysis using LLM agents and function calling.

Finally Outlining the data models, service interfaces, configuration management, and testing strategy has been implemented. Eventually Providing a detailed breakdown aligning the technical design with the requested file structure. The proposed architecture addresses the core technical challenge of achieving a perceived low-latency interaction with LLMs in a conversational setting, which is crucial for a natural and engaging user experience.

5.2 CHALLENGES ENCOUNTERED

Challenges in Designing a Real-Time Backend with Asynchronous Tasks and LLM Interactions

Developing a backend that supports real-time operations with asynchronous tasks and LLM interactions introduces several key challenges:

1. State Management

Maintaining live interview session states across multiple processes (API server, Celery workers) is complex, especially with active WebSocket connections.

- Requires careful consideration of where state is stored (in-memory vs. shared backend).

- Ensures proper synchronization for consistency and reliability.

2. Real-Time Coordination

Managing the timing and flow of information between the WebSocket handler and asynchronous Celery tasks is challenging.

- Prevents race conditions, duplicate messages, and out-of-order responses.
- Requires robust synchronization logic to ensure correct message delivery (tentative vs. final responses).

3. LLM Latency Variability

LLM response times fluctuate based on factors such as system load, model size, and response length.

- Requires a fallback mechanism with well-tuned timing thresholds to mitigate unpredictable delays.

4. Error Handling

Handling failures in external LLM APIs, the message broker, or Celery workers while preserving session state is critical.

- Needs robust recovery mechanisms.
- Provides clear and informative feedback to the user via WebSocket.

5. Resource Management

LLM inference consumes significant computational resources.

- Requires proper provisioning of Celery workers to balance performance and cost.
- Optimizes the load distribution across LLM services.

6. Code Analysis Complexity

Reliable code analysis with LLM agents and function calling demands:

- Careful prompt engineering.
- Thoughtful tool design and execution safety measures (e.g., sandboxing code execution).

5.3 FUTURE WORK

Several areas can be explored to improve functionality, efficiency, and user experience:

1. Voice Interface

Integrate server-side **Automatic Speech Recognition (ASR)** and **Text-to-Speech (TTS)** services to provide a fully voice-based interview experience rather than relying solely on text-based interactions.

2. Improved Real-Time Logic

Refine the chunking and response strategy to enhance efficiency:

- Leverage **streaming LLM outputs**, if supported by the provider.
- Explore alternative asynchronous patterns, such as **message queues for responses** between the API process and Celery workers.

3. Advanced State Persistence

Ensure session continuity by implementing **persistent storage** for interview session state using **Redis or a database** to maintain state across backend restarts or worker failures.

4. Extended Analysis Features

Enhance analytical capabilities with:

- **Communication skill evaluation**, measuring clarity and conciseness using ASR timing and pauses.
- **Body language analysis**, if video integration is introduced.
- **Response benchmarking**, comparing candidate answers against a database of ideal responses.

5. More Sophisticated Mini-LLM

Improve the **mini-LLM's** ability to generate varied, contextually aware fillers and surprises, making responses more natural and engaging.

6. Adaptive Interview Flow

Make the LLM interviewer more dynamic by **adapting question difficulty and style** in real time based on the candidate's performance.

7. Frontend Integration

Develop a **user-friendly interface** that fully utilizes the backend's real-time capabilities by:

- Handling **streaming text responses**.
- Providing **visualized interview analysis**.

8. Production Deployment

Optimize deployment on a **cloud platform** to ensure:

- **Scalability**, handling increasing user traffic efficiently.
- **Reliability**, minimizing downtime and improving system robustness.
- **Security**, protecting sensitive data and ensuring compliance.

- **Cost-effectiveness**, balancing performance with operational expenses.

9. Security

Implement comprehensive security measures including:

- **Authentication & Authorization**, ensuring secure user access.
- **Data Encryption**, securing information both at rest and in transit.
- **Secure Handling of Sensitive Documents & Conversation Data**, preventing unauthorized access.

REFERENCES

[1] J. Smith, "The Role of AI in Modern Recruitment," Journal of Human Resources, vol. 10, no. 2, pp. 45-58, 2022.

[2] A. Kaur and S. Singh, "AI Interview Platforms: A Review of Capabilities and Challenges," International Journal of Artificial Intelligence in Education, vol. 5, no. 1, pp. 112-125, 2023.

[3] D. Jurafsky and J. H. Martin, Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, 3rd ed. Pearson, 2023.

[4] C. D. Manning, P. Raghavan, and H. Schütze, Introduction to Information Retrieval. Cambridge University Press, 2008.

[5] A. Vaswani et al., "Attention Is All You Need," in Advances in Neural Information Processing Systems 30, 2017.

[6] Brown, T. B. et al., "Language Models are Few-Shot Learners," in Advances in Neural Information Processing Systems 33, 2020.

[7] I. Fette and A. MacDonald, The WebSocket Protocol. RFC 6455, 2011.

[8] B. Sereda, Celery: Distributed Task Queue. https://docs.celeryproject.org/, 2024. (Accessed: Month Day, Year).

[9] R. J. Mooney and L. Mooney, "Text Categorization and Information Extraction," in The Oxford Handbook of Computational Linguistics, 2003.

[10] OpenAI, "Function Calling and the OpenAI API," 2023. https://openai.com/blog/function-calling-and-the-openai-api

[11] "Moshi: A speech to text foundation model for real-time dialogue" arXiv:2410.00037