AI-Driven Automated Interviewer for Project Presentations

System Design and Technical Documentation

Swaib Ilias Mazumder

January 17, 2026

1 Introduction

Modern technical interviews and project evaluations require assessing not only final results but also a candidate's understanding of system design, architectural decisions, and implementation trade-offs. Traditional evaluations are manual, subjective, and difficult to scale. This project introduces an **AI-driven automated interviewer system** that simulates a live technical interview during project presentations. The system observes the presenter's screen in real time, extracts contextual information, and dynamically generates technically relevant interview questions using Large Language Models (LLMs). The system is designed as a modular, extensible, and industry-aligned application suitable for research, interview preparation, and automated evaluation pipelines.

2 Design Goals and Principles

The system is built following established industry software and ML system design principles:

- **Modularity**: Each component is independently replaceable.
- **Single Source of Truth**: Centralized application state management.
- **Event-Driven Control Loop**: Periodic orchestration instead of blocking execution.
- **Fail-Safe Design**: Graceful degradation when external APIs fail.
- **Observability**: Extensive logging and runtime metrics.
- **Scalability**: Designed to support future multimodal inputs (audio, video).

3 High-Level System Architecture

At a high level, the system follows a **control-loop architecture** commonly used in production ML systems.

$$\textit{Streamlit UI} \rightarrow \textit{Orchestrator} \rightarrow \{\textit{Screen Capture}, \textit{OCR}, \textit{LLM}\} \rightarrow \textit{State Update}$$

The orchestrator acts as the central coordinator, periodically evaluating system state and triggering downstream components.

4 Component-Level Architecture

4.1 Streamlit User Interface (UI)

File: `app/main.py`

The Streamlit UI serves as the primary interaction layer between the user and the system. It is responsible for:

- Session lifecycle control (Start, Pause, Stop, Clear)
- Displaying live screen snapshots
- Showing generated interview questions
- Visualizing logs and system metrics

Design Rationale

- Streamlit enables rapid prototyping and real-time UI updates.
- UI logic is kept stateless; all persistent data is stored in the central state.

4.2 Central Application State

File: `app/state.py`

The application state is implemented as a centralized data structure and acts as the **single source of truth**. It stores:

- Session status (`IDLE`, `RUNNING`, `PAUSED`)
- Timing information (timestamps, intervals)
- Current question index and difficulty
- Logs, counters, and runtime metrics

Industry Alignment

This design mirrors state containers used in:

- Distributed systems (control planes)
- Reinforcement learning environments
- Workflow orchestration engines

4.3 Orchestrator (Control Loop)

File: `app/logic/orchestrator.py`

The orchestrator is the core execution engine of the system. It runs as a periodic loop triggered by the UI.

Responsibilities

- Managing session lifecycle transitions
- Enforcing timing constraints (e.g., screenshot interval)
- Routing data between components
- Handling API failures and fallback logic

Control Loop Logic

At each tick:

1. Check session state
2. Capture screen if interval elapsed
3. Perform OCR on captured frame
4. Generate or queue interview questions
5. Update state and logs

This loop-based orchestration is similar to:

- Kubernetes controllers
- Training loops in ML systems
- Streaming data pipelines

4.4 Screen Capture Module

File: `app/capture/screen.py`

This module is responsible for capturing live screenshots of the presenter's screen. Captured frames are saved as temporary artifacts for downstream processing.

Key Characteristics

- Platform-agnostic screen capture
- Configurable resolution and interval
- Minimal blocking to maintain UI responsiveness

4.5 OCR Pipeline

The OCR step extracts textual content from captured screen frames. This content acts as the contextual grounding for question generation.

Design Considerations

- OCR is treated as a best-effort signal
- No hard dependency—system continues even if OCR fails

4.6 LLM Interviewer

File: `app/logic/llm_interviewer.py`

The LLM interviewer is responsible for generating interview-style questions using a Large Language Model.

Prompt Engineering Strategy

Prompts are constructed using:

- Extracted OCR context
- Question history
- Difficulty ramp parameters

Failure Handling

- API rate limits are detected
- Predefined fallback questions are used
- System logs all failures for observability

4.7 Logging and Observability

The system maintains structured logs visible directly in the UI.

Metrics Tracked

- OCR calls and latency
- LLM calls and failures
- Session timestamps

This design aligns with industry best practices in ML system observability.

5 Execution Workflow

1. User launches the Streamlit application
2. Session is initialized in `IDLE` state
3. User starts session → state transitions to `RUNNING`
4. Orchestrator loop begins execution
5. Screens are captured and processed
6. Interview questions are generated dynamically

6 Limitations

- Dependency on external LLM APIs
- OCR accuracy varies with screen content
- Speech-to-Text integration is experimental

7 Future Extensions

Planned improvements include:

- Fully offline LLM inference
- Robust Speech-to-Text pipeline
- Automated rubric-based scoring
- Interview summary and report generation

8 Conclusion

This project demonstrates a production-aligned, modular AI system capable of real-time contextual evaluation. Its architecture follows established industry patterns, making it suitable for extension into scalable automated interview and assessment platforms.