Design Document for AI Mock Interviewer PoC

Project Overview

This document outlines the design and rationale for the AI Mock Interviewer Proof of Concept (PoC), focused on simulating Excel skill interviews. The system enables users to engage in a chat-based mock interview with an AI agent ("Alex"), receive real-time evaluations, and obtain a performance report. Developed within a 10-hour deadline, it prioritizes an MVP that meets core requirements: structured conversation flow, evaluation, database storage, and deployment.

Key Goals:

- Address the "cold start" problem using advanced prompt engineering with Gemini AI.
- Ensure a simple, scalable architecture for rapid development.
- Provide a functional end-to-end experience with testing and documentation.

Architecture

The application follows a three-tier architecture:

1. **Frontend (Client Tier):** React-based single-page application (SPA) for the chat interface. Handles user input, displays messages, and calls backend APIs.
2. **Backend (Application Tier):** Spring Boot REST API managing business logic, AI integration, and database interactions.
3. **Data Tier:** PostgreSQL database for persisting interview sessions and chat history.

Data Flow:

- User starts interview via frontend → Backend creates session, generates initial AI message via Gemini → Stores in DB.
- User sends message → Backend evaluates via Gemini, generates response, updates DB.
- User ends interview → Backend generates report via Gemini, marks session complete.

This separation ensures modularity: Frontend focuses on UI, backend on logic/AI/DB.

Tech Stack Justification

- **Backend - Spring Boot (Java 21):** Chosen for rapid API development with built-in features like JPA for DB access and easy integration. Familiarity with Java/Spring allows quick implementation of controllers, services, and entities.

- **Frontend - React (with Vite):** Enables dynamic, responsive chat UI. Vite provides fast builds; hooks manage state efficiently. Tailwind CSS for quick, professional styling.
- **Database - PostgreSQL:** Reliable relational DB for structured data (sessions/messages). Supports JSON if needed for evaluations; free tier on Render.
- **AI - Google Gemini API:** Powerful LLM for conversational AI and evaluations. Handles prompt engineering well for "cold start" (no dataset needed). Free tier is sufficient for PoC, with easy Java SDK integration.
- **Other Tools:** Lombok for boilerplate reduction, Axios for frontend API calls, Docker for local DB consistency.

This stack aligns with expertise in Java/React, enabling fast iteration while being production-ready.

"Cold Start" Solution: Advanced Prompt Engineering

Without pre-existing datasets, we use Gemini's instruction-following capabilities via carefully crafted prompts:

- **System Prompt for Interview Flow:** Instructs AI to act as "Alex" (Excel expert), start with intro, ask 3-4 escalating questions (basic formulas to advanced like VBA/pivots), evaluate prior answer (Correct/Partial/Incorrect with brief reason), and adapt based on history.
- **Evaluation Logic:** Embedded in prompt: Analyze user answer against expected, score implicitly, provide one-line feedback before next question.
- **Report Prompt:** Separate prompt takes full chat history, summarizes scores, strengths/weaknesses, and overall rating (e.g., 8/10).

This ensures coherent, stateful conversations without training data.

Database Schema

Two entities for simplicity:

- **InterviewSession:**
    - ID (UUID, PK)
    - User ID (String, optional for future auth)
    - Start Time (Timestamp)
    - End Time (Timestamp)
    - Status (Enum: ACTIVE, COMPLETED)
- **ChatMessage:**
    - ID (UUID, PK)
    - Session ID (FK to InterviewSession)
    - Sender (Enum: USER, AI)

    - Message (Text)
    - Evaluation (String, for AI responses)
    - Timestamp (Timestamp)

Relationships: One-to-Many (Session to Messages). JPA/Hibernate auto-manages schema via `ddl-auto=update`.

API Endpoints

- `POST /api/v1/interviews`: Starts session, returns ID and initial AI message.
- `POST /api/v1/interviews/{id}/chat`: Sends user message, returns AI response with evaluation.
- `GET /api/v1/interviews/{id}/report`: Ends session, returns text report.

Error Handling: 404 for invalid ID, 400 for inactive sessions.

Frontend Design

- **Components:** Chat container with message list (bubbles for USER/AI), input form.
- **State Management:** `useState` for messages/session ID; `useEffect` for polling/updates.
- **Styling:** Tailwind for dark-themed, responsive UI (mobile-friendly chat).

Deployment Rationale

- **Backend & Database on Render:** Chosen for the free tier supporting Dockerized Java apps and managed PostgreSQL. Easy Git integration for auto-deploys, env var management for secrets (DB creds, Gemini key). Rationale: Quick setup (no AWS config), built-in proxy/SSL, and logs for debugging. Drawback: Free tier has cold starts (addressed below).
- **Frontend on Vercel:** Ideal for static React apps—fast deploys, global CDN, auto-SSL. Rationale: Seamless Git integration, preview branches for testing, free for PoC. Alternative Firebase for similar static hosting if needed.

Note: The deployed link may be slow or experience startup delays due to limitations of the Render free tier account (e.g., cold starts after inactivity, taking 30-60 seconds). For better performance, consider upgrading to a paid tier.

Development Phases

The project was executed in structured phases to meet the deadline:

Phase 1: Foundation & Design (2 Hours)

- Created `DESIGN.md` with architecture, stack justification, prompt strategy, and schema.
- Scaffolded projects: Spring Initializr for backend, Vite for frontend, Docker for DB.

Phase 2: Backend API Development (3 Hours)

- Implemented entities/repositories, controllers, and GeminiService with prompts.
- Tested APIs via Postman (see `apitest.txt` for guide).

Phase 3: Frontend Development (3 Hours)

- Built UI components, state logic, and API integration with Axios.
- Styled with Tailwind; tested local flow.

Phase 4: Deployment & Finalization (2 Hours)

- Containerized backend, deployed to Render with env vars.
- Deployed frontend to Vercel.
- E2E tested 2-3 interviews, updated docs (`README.md` with setup/deploy details).

Risks & Limitations

- Gemini quota: Free tier limits—monitor usage.
- Scalability: MVP; add auth/scaling in the future.
- Cold Starts: As noted, Render free tier issue—paid tier resolves.

This design ensures a functional, deployable PoC. For full code, see repo files.