Submission Report: AI-Powered Excel Interviewer

Project Mission: To design and build an automated, AI-powered system to screen candidates for Excel proficiency, solving the business problems of inconsistency, cost, and time bottlenecks in the manual hiring process.

This document details the product strategy, system architecture, and final implementation of the deployed application

Phase 1: Product Strategy & Design (The "Why")

This phase addressed the core strategic challenges from the assignment: creating an intelligent agent with no prior data, justifying the technology stack, and designing a robust interview flow.

1. Solving the "Cold Start" & "Intelligent Evaluation" Problem

- **The Problem:** The assignment assumed that there was no pre-existing dataset of interview transcripts. Therefore, a traditional machine learning (fine-tuning) approach was impossible.
- **The Solution:** This "Cold Start" problem was solved by creating a **knowledge-driven system**, not a data-driven one. The core "brain" of the application is the `adaptive_question_bank.json` file. This expert-curated file acts as the "ground truth" and contains two key components:
    1. **A Question Bank:** A pool of questions categorized by topic (e.g., "Lookup Functions") and tagged by **difficulty** ("Easy", "Medium", "Hard").
    2. **An Evaluation Rubric:** *Every single question* is paired with a detailed, expert-defined rubric. This rubric lists the exact concepts a candidate's answer must contain to be considered correct.
- **Justification:** This approach directly solves the core business problem of "inconsistent evaluations." The AI is not "guessing" what a good answer is; it is programmatically instructed to act as a strict grader, comparing the candidate's natural language answer directly against this expert rubric. This makes the evaluation intelligent, consistent, and verifiable.

2. Designing an "Agentic" Interview Flow

- **The Goal:** A simple 5-question list does not constitute an "agent." The goal was to design a truly adaptive agent that mimics the behavior of a real interviewer, fulfilling the "Agentic Behavior and State Management" requirement.
- **The Solution:** The agent manages a "state" for each candidate, tracking `current_difficulty`, `consecutive_failures`, `hard_questions_passed`, and `questions_asked`. This state is governed by a **Decision Engine**:
    - A candidate who answers a question **correctly** (Score 4-5) is promoted to the next difficulty level (e.g., "Easy" → "Medium").
    - A candidate who answers **partially** (Score 3) is tested again at the same level.
    - A candidate who answers **poorly** (Score 1-2) is demoted (e.g., "Hard" → "Medium").
- **Justification:** This adaptive logic is far more efficient and provides a richer signal. It quickly finds a candidate's "skill ceiling" : experts are not bored by easy questions, and novices are not demoralized by overly complex ones. The interview concludes when a clear signal is found (e.g. 3 hard questions passed, 2 consecutive failures, or a 15-question maximum is reached).

3. Justifying the Technology Stack

- **Language: Python**
    - **Why:** The non-negotiable standard for AI and data science. It provides the entire ecosystem of required libraries (`streamlit`, `google-generativeai`).
- **AI Model: Google Gemini (e.g., gemini-2.5-pro)**
    - **Why:** This model was chosen for its advanced reasoning capabilities and, most critically, its **native JSON output mode**. This feature is the backbone of the "Grader," as it allows the AI to be forced to return a structured `{"score": 5, "feedback": "..."}` object, which the agent can then read and act upon reliably.
- **Framework & Hosting: Streamlit & Streamlit Community Cloud**
    - **Why:** For a Proof-of-Concept, speed-to-deployment is paramount. Streamlit allows a fully interactive, stateful chat application to be built and deployed from a single Python script. Streamlit Community Cloud provides free, instant hosting directly from a GitHub repository, fulfilling the "Deployed Link" deliverable.

Phase 2: System Architecture & Development (The "How")

This phase involved building the core components of the application. The system is designed with a modular, "multi-persona" AI architecture.

1. **The "Grader" (`evaluate_answer` function):** This is a specialized AI "brain" with a strict persona. Its only job is to receive a question, a rubric, and a user's answer. It is constrained by a prompt to return *only* a JSON object with the score and feedback. It has no memory and no other role.
2. **The "Reporter" (`generate_final_report` function):** This is a separate AI "brain" with a "Hiring Manager" persona. Its job is to synthesize the *entire* interview history (a JSON list of all answers and scores) and write the high-level, human-readable **Constructive Feedback Report**.
3. **The "Agent" (The `app.py` script):** This is the main application logic. It manages the **User Interface (UI)** and the **Session State**. It is responsible for the overall "Structured Interview Flow" :
    - It introduces the app.
    - It selects a question.
    - It calls the "Grader" to get a score.
    - It runs the "Decision Engine" to update its state.
    - It decides when to end the interview.
    - It calls the "Reporter" to generate the final summary.

This modular design makes the system robust, testable, and easy to maintain.

Phase 3: Enterprise Refinement & Deployment (The "V2")

The final phase was to evolve the PoC into a secure, enterprise-grade application that truly solves the business problem. This involved adding three critical refinements.

1. Security: The Password Wall

- **What Was Done:** A public-facing tool is not a screening product. A global password wall was implemented at the very start of the app.
- **Justification:** This ensures that only authorized personnel (and candidates who have been given the password) can access the app, preventing public use and securing API keys. This is handled via `st.secrets`.

2. Test Integrity: The "Silent" Interview

- **What Was Done:** A critical product decision was made to **remove all real-time feedback**. A candidate should not be told their score or *any* feedback during the interview.
- **Justification:** Showing scores turns the "test" into a "tutoring session," which invalidates the results. In the final version, the candidate answers a question, and the app simply moves to the next one. The scoring and adaptive logic all happen **silently in the background**. This dramatically increases the integrity and professional-grade quality of the screening.

3. Data Routing: Private & Automated Reporting

- **What Was Done:** Since the candidate no longer sees their report, a private data pipeline was implemented to run when the interview is over.
- **How it Works:** The app generates the final report in the background, then (using Python's built-in `smtplib`) **automatically emails the full, private report** to a secure, pre-defined company email address (e.g., `hiring-manager@company.com`), which is also stored in `st.secrets`. The candidate only sees a generic "Thank you for your time" message.
- **Justification:** This is the final piece that solves the entire business problem. It creates a seamless, automated workflow:
    1. A candidate takes a secure, objective test.
    2. The company *instantly* receives a detailed, consistent performance report in their inbox.
    3. A robust `try...except` block ensures that if the email send fails, the full report is printed to the **app's private logs** as a guaranteed fallback, ensuring 100% data capture.

This final architecture successfully fulfills the entire mission: an automated, consistent, and secure system that replaces the manual screening bottleneck.

Phase 4: Future Improvements

This Proof-of-Concept serves as a strong foundation, but several clear paths exist for a V3/enterprise-grade product.

1. **Dedicated Database & Admin Dashboard:**
    - **Improvement:** Replace the current email/log solution with a robust database (e.g., Google Sheets, Firebase, or Supabase).
    - **Benefit:** This would allow for the creation of a *second* Streamlit app, a private "Admin Dashboard." Hiring managers could log in to this dashboard to see a sortable, filterable table of *all* candidate submissions, review performance over time, and analyze metrics.
2. **Advanced "Data Flywheel" Strategy:**
    - **Improvement:** The new database would collect hundreds of interview transcripts. This curated data (question, answer, score, and AI feedback) is the exact dataset the project was initially missing.
    - **Benefit:** This data can be used to **fine-tune** a smaller, open-source model. Over time, this fine-tuned model could replace the expensive `gemini-2.5-pro` model for the "Grader" persona, dramatically reducing API costs while potentially increasing accuracy on this specific task.
3. **Enhanced Question Bank Management:**
    - **Improvement:** Create a simple "Question Bank" admin tool (it could be another page in the Admin Dashboard).
    - **Benefit:** This would allow non-technical hiring managers to **add, edit, and retire questions** from the `adaptive_question_bank.json` file through a simple web interface, without ever needing to touch the application's source code.