InterviewPrepBot — Hackathon Documentation

Project Title

InterviewPrepBot — A RAG-powered interview preparation assistant

Abstract

InterviewPrepBot is a local-first web application that helps users prepare for technical and HR interviews by providing personalized answers, mock questions, and tailored suggestions. It uses Retrieval-Augmented Generation (RAG) with a local vector database (Chroma) and a large language model to return concise, context-aware responses generated from curated interview data.

Problem Statement

Many students and job seekers find interview preparation unfocused: generic question lists, no personalization, and lack of targeted answers based on a candidate's profile. The goal is to provide a fast, privacy-preserving assistant that uses curated interview content + a user's profile to generate role-specific, high-quality interview answers and practice questions.

Objectives

- Build a local RAG pipeline that retrieves relevant interview content and augments LLM prompts.
- Personalize answers using user profile (name, education, specialization, skills, target role).
- Use adaptive/semantic chunking to preserve context and improve retrieval quality.
- Support hybrid search (semantic + lexical) with thresholding to reduce irrelevant results.
- Provide a responsive React frontend and a FastAPI Python backend.
- Containerize the system using Docker and apply rate limiting to protect token budget.

Data Pipeline

1. Data collection

- Sources: scraped and curated lists (e.g., top 40 interview Q&A, PDFs with 100 Q&A). Consolidate into a single canonical dataset.
- Store original sources and metadata for traceability (source URL/name, author, timestamp).

2. Data cleaning

- Normalize whitespace, remove unnecessary line breaks, fix encoding issues.
- Remove duplicate questions/answers and normalize question formats.
- Standardize metadata fields: id, source, title, role_tags, skills_tags, text, created_at.

3. Adaptive / Semantic chunking

- Apply adaptive chunking (context-aware splitting by paragraph/semantic boundaries) instead of fixed-length only.
- Use embedding model token limits to compute chunk size; set chunk length to 75–85% of model token capacity and overlap to 10–15%.
- Rationale: ensures each chunk carries meaningful context while preventing token overflow when concatenated with prompts.

4. Embedding

- Choose an embedding model compatible with local or cloud LLMs (ensure same embedding space as retrieval model where possible).
- Convert each chunk to a vector and attach metadata (question id, source, role tags, chunk_index, text snippet).

5. Vector storage

- Use ChromaDB (local) for storing vectors + metadata. Chroma chosen because:
    - It stores metadata alongside vectors easily.
    - Works well for local-first setups.
    - Simple to persist and query.
- Keep a backup export (e.g., serialized vectors + metadata) for reproducibility.

Retrieval & Augmentation

Query handling

1. User provides profile (name, graduation, specialization, skills, target roles) + query (e.g., "How to answer 'Tell me about yourself'").
2. Embed user profile and query (either combined or separately) to form the query vector.

Search strategy

- Use hybrid search: combine semantic similarity + keyword match (lexical) to ensure exact phrase matches (important for technical terms).
- Apply a similarity threshold and a maximum results cap to reduce irrelevant results.
- Retrieve top-K chunks (e.g., K = 8–12) and rank by combined score.

Augmentation & Prompting

- Concatenate selected chunks into an augmentation context block while monitoring token budget.
- Use prompt templates that instruct the LLM to generate answers in a specific format (short answer, bullets, sample follow-up questions, scoring criteria).
- Include user profile tokens in the prompt to personalize answers (role-specific tailoring, skill emphasis).

Sample prompt template (conceptual)

You are an interview coach. Use the following context (extracted interviews & Q/A) and the candidate profile to generate a role-appropriate answer.

Candidate: {name}, Education: {degree}, Specialization: {spec}, Skills: {skills}, Target Role: {role}

Context:

{augmented_chunks}

Task: Provide a concise model answer (max 200 words), 3 follow-up questions, and 2 quick tips for improvement.

System Architecture

- **Frontend:** React + Tailwind CSS (single-page UI for input, review, mock interview flows).
- **Backend:** FastAPI (serves endpoints for embedding, vector search, prompt generation, rate limiting).
- **Embeddings & LLM:** Local or hosted model depending on hackathon constraints. Embedding model must be compatible with retrieval.
- **Vector DB:** ChromaDB (local persistence).
- **Containerization:** Docker for reproducible development & deployment.
- **Rate limiting & Token Management:** Track incoming augmented prompt size + expected model output size to avoid exceeding context window; limit requests per minute per user.

(ASCII flow)

[User UI] -> [FastAPI] -> [Embed user+query] -> [ChromaDB search (hybrid)] -> [Augment + Prompt] -> [LLM] -> [Result]

Tech Stack

- Frontend: React, Tailwind CSS, Vite (or Create React App)
- Backend: Python, FastAPI, Uvicorn
- Embeddings & LLM: chosen local or hosted models (example: sentence-transformers / OpenAI / local LLaMA family)
- Vector DB: ChromaDB (local)
- Containerization: Docker
- Others: Redis (optional for caching, rate limit counters)

Implementation Steps (Detailed)

1. **Prepare data**: Consolidate PDFs and raw Q/A into a folder data/raw/.
2. **Cleaning script**: scripts/clean_data.py — normalize text and produce data/cleaned.jsonl.
3. **Chunking**: scripts/chunk_data.py — implement adaptive chunking that splits on semantic boundaries and respects token-based sizes.
4. **Embedding**: scripts/embed_chunks.py — load chunked data, call embedding model, store resulting vectors and metadata.
5. **Vector DB ingestion**: scripts/ingest_chroma.py — create a Chroma collection and persist vectors.
6. **Backend endpoints** (FastAPI):
    - POST /embed-query — returns query embedding
    - POST /search — takes embedding, returns top-K chunks
    - POST /generate — builds prompt, calls LLM, returns result
7. **Frontend**: build inputs for user profile, question box, display area for generated answer, feedback (thumbs up/down), and practice mode (timed answers).
8. **Rate limiting**: integrate slowapi or custom token-bucket per IP/user to limit requests.
9. **Dockerize**: Write Dockerfile for backend + frontend, docker-compose.yml to run Chroma locally.
10. **Testing & Evaluation**: create unit tests for chunking, embedding, and end-to-end integration tests.

API Contract (Example)

- POST /api/v1/query
    - Body: { "profile": {..}, "question": "..." }
    - Flow: embed -> search -> augment -> generate -> return
    - Response: { "answer": "...", "followups": [...], "tips": [...] }

UI/User Flow

1. User fills profile (name, degree, specialization, skills, target role).
2. User types/pastes a question or chooses from predefined sections (HR, CS fundamentals, system design, behavioral).
3. System shows a model answer, 3 follow-ups, and two tips. User can ask for simplification or for a sample answer with real-life examples.
4. Provide practice mode: timed mock questions, record user's answer (optional), and give feedback.

Deployment & Ops

- Use Docker Compose to run backend + Chroma + optional Redis.
- For local hackathon demo, run everything on a single machine with limited concurrency.
- Monitor memory and disk usage (vectors can grow quickly); add a retention policy for rarely used vectors.

Security, Privacy & Ethics

- Keep data local and private by default; show users where their data is stored.
- Avoid exposing raw scraped sources directly to users without attribution.
- Add hygiene checks on user-provided personal data.
- Provide a feedback mechanism to mark hallucinations or incorrect answers.

Future Enhancements

- Add multimodal data (video/mock interview recordings) and transcriptions.
- Fine-tune a local LLM on curated high-quality answers for better consistency.
- Add candidate progress tracking and personalized study plans.
- Add interview scheduling & calendar integration.

Conclusion

InterviewPrepBot is a compact, local-first RAG system optimized for personalized interview preparation. The design balances context preservation (adaptive chunking), retrieval quality (hybrid search), and privacy (local Chroma + containerization). This documentation outlines a runnable implementation plan suitable for a hackathon MVP and future expansion.