My interview agent

i want to build a platform for engineering students who are persuing or completed their bachlors and struggling in technical interview because lack of interview experience. basically my platform consist of a 3d ai human figure that take interviews of students. basically the interview procedure is - 1.students register in our platform and login. during registration they need to fill up thier all details with education, address, company the're intersted in (select company available in option or write in other section),their skills and finally their resume in pdf format only. 2. after filling all the details profile will be created individually allowing CRUD operation, 3. now 3d ai human figure appear on a big screen and ask them a few questions about them and ask them their resume. 4.start interview button appear after then interviewer gets their introduction analyse it and start asking questions from their resume and point they mentioned in their introduction by going into depths of their resume projects and after that interviewer ask them questions about OOP,DBMS,OS and CN subject most frequently asked questions after that interviewer greets the students and finish the interview and 6. finally interviewr gives the whole analysis (score, communication, technical, skillwise, resume knowledge, tips to improve and score should be real and genuin based on interview)of the interview to the student. every studnt has its seperate dashboard where student can track thier past interviews scores & analysis. making platform fully dynamic. stack selection - Component Recommended Open-Source Option(s) Key Considerations 3D Avatar & Lip-Sync Avatoon (React Three Fiber component) . Provides lip-sync via viseme data and head motions. Requires a compatible 3D model (.glb format). You can find free models or use tools to create one. Speech-to-Text (STT) Handy (desktop app with Whisper) or hns (privacy-focused CLI) . Runs completely offline, ensuring privacy. Requires sufficient local computational resources (GPU recommended). Text-to-Speech (TTS) with Emotion & Visemes HeadTTS or ZeroVOX . Both provide phoneme-level timestamps/visemes for lip-sync. HeadTTS can even run in the browser! HeadTTS: Great for browser-based inference (WebGPU) and provides Oculus visemes. ZeroVOX: Excellent for offline server-side TTS with zero-shot voice cloning. Large Language Model (LLM) Teuken-7B (from OpenGPT-X) or Apertus (from EPFL) . Both are fully open-source and commercially usable. Both are multilingual and designed for chat. You'll need to host them yourself (on your own server or a local machine) to keep costs at zero. Resume Parsing CV_extract (Python-based) or OpenResume parser . CV_extract: Specifically designed to extract structured info from PDF's using LLMs. OpenResume: A popular resume builder with parsing capabilities. Backend & Database FastAPI (Python) & PostgreSQL. These remain excellent, free, and open-source choices for your backend and database. i'm a beginner in tech and i don't familier with any of these technologies your task is to make the whole project for me. in zero cost milestone. guide me how to make this project step by step in vs code starting from very scrap consider i have 0 knowledge of technology and guilde me in super extremly detail for doing this project.

Thought for 7s

Plan & step-by-step guide — build the whole project from scratch (zero cost, beginner friendly)

Amazing idea — I'll lay out a clear, *hands-on*, zero-cost path you can follow in VS Code from absolute scratch. I'll give you exact commands, folder structure, core code snippets, and progressive milestones so you can ship a working prototype locally and then improve it. I'll also point out where true zero-cost choices force tradeoffs (e.g., fully self-hosting big LLMs and advanced TTS/viseme systems requires hardware), and provide simpler local alternatives so you can move forward without GPUs or cloud spend.

Throughout I'll reference the open-source components you listed and suggested alternatives so you can plug them in later:

- React — frontend UI library.
- Avatoon — 3D avatar / lip-sync component (React).
- HeadTTS — browser TTS with visemes.
- ZeroVOX — server-side TTS with viseme timestamps.
- Whisper — speech-to-text (OpenAI Whisper family / forks).
- Teuken-7B — open LLM option.
- Apertus — alternate open LLM.
- CV_extract — resume parsing library.
- FastAPI — backend framework.
- PostgreSQL — recommended DB (we'll start with SQLite for simplicity).

High-level architecture (what we will build)

1. React frontend (Vite) with a 3D avatar canvas (Avatoon or basic GLB model via @react-three/fiber) and a WebSocket connection for real-time audio/viseme updates.
2. FastAPI backend (Python) exposing REST endpoints (auth, profile, resume upload, CRUD) and a WebSocket server for real-time signals. FastAPI
3. Local DB (SQLite initially; optional switch to PostgreSQL later).
4. Resume parser (start simple with PyPDF2 + heuristics; plug in CV_extract later).
5. Interview engine:
   - Phase A (simple, zero-cost): rule/template driven — generate questions from resume fields and common CS topics (OOP, DBMS, OS, CN).
   - Phase B (advanced, optional): use local LLM like Teuken-7B or Apertus to generate deeper follow-ups (requires heavier hardware).
6. TTS + viseme streaming:
   - Browser TTS option: HeadTTS runs in browser (WebGPU) — easiest for lip sync demo.
   - Server TTS option (optional): ZeroVOX for better voices and visemes.
7. STT (speech → text): local Whisper variant for transcription (Whisper) or record audio and upload for batch transcription.

Milestones (progressive, with exact actions)

Milestone 0 — prerequisites (do this first)

Commands below assume Windows/Mac/Linux. Install each tool:

1. Install VS Code: https://code.visualstudio.com/
2. Install Git: https://git-scm.com/
3. Install Node.js (LTS >= 18): https://nodejs.org/
4. Install Python 3.10+ and pip. On Windows use installer; on Linux use your package manager.
5. (Optional but recommended) Install poetry or pipenv — we'll use plain venv & pip to keep things simple.

Open a terminal and verify:

```
node -v

npm -v

python -V

git --version
```

Milestone 1 — create repo and basic project skeleton

1. Make a folder and init git:

```
mkdir deepseek-interview-bot

cd deepseek-interview-bot

git init
```

2. Create two folders: backend and frontend.

```
mkdir backend frontend
```

Milestone 2 — backend: FastAPI + SQLite (auth, user profile, resume upload)

We'll scaffold a small FastAPI app with SQLAlchemy + Alembic later (start without migrations).

1. Create a virtual env and install basics:

```
cd backend

python -m venv .venv

# activate .venv: Windows: .venv\Scripts\activate  |  mac/linux: source .venv/bin/activate

pip install fastapi uvicorn sqlalchemy pydantic python-multipart aiofiles passlib bcrypt PyPDF2
```

2. Minimal backend/app/main.py

```
# backend/app/main.py
```

```
from fastapi import FastAPI, UploadFile, File, HTTPException, Depends

from fastapi.staticfiles import StaticFiles

from fastapi.middleware.cors import CORSMiddleware

from pydantic import BaseModel

import shutil

import os

from pathlib import Path

from sqlalchemy import create_engine, Column, Integer, String, Text, DateTime

from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy.orm import sessionmaker

import datetime


BASE_DIR = Path(__file__).resolve().parent

UPLOAD_DIR = BASE_DIR / "uploads"

UPLOAD_DIR.mkdir(parents=True, exist_ok=True)


app = FastAPI()

app.add_middleware(CORSMiddleware, allow_origins=["*"], allow_methods=["*"],
allow_headers=["*"])


DATABASE_URL = "sqlite:///./db.sqlite3"

engine = create_engine(DATABASE_URL, connect_args={"check_same_thread": False})

SessionLocal = sessionmaker(bind=engine)

Base = declarative_base()


class User(Base):

    __tablename__ = "users"

    id = Column(Integer, primary_key=True)

    name = Column(String, index=True)

    email = Column(String, unique=True, index=True)

    resume_path = Column(String, nullable=True)
    created_at = Column(DateTime, default=datetime.datetime.utcnow)

Base.metadata.create_all(bind=engine)

class UserCreate(BaseModel):
    name: str
    email: str

@app.post("/users/")
def create_user(user: UserCreate):
    db = SessionLocal()
    existing = db.query(User).filter(User.email == user.email).first()
    if existing:
        raise HTTPException(400, "User exists")
    new = User(name=user.name, email=user.email)
    db.add(new)
    db.commit()
    db.refresh(new)
    return {"id": new.id, "name": new.name, "email": new.email}

@app.post("/users/{user_id}/upload_resume")
async def upload_resume(user_id: int, file: UploadFile = File(...)):
    if not file.filename.lower().endswith(".pdf"):
        raise HTTPException(400, "Only PDF resumes allowed")
    db = SessionLocal()
    u = db.query(User).filter(User.id == user_id).first()
    if not u:
        raise HTTPException(404, "User not found")
    out_path = UPLOAD_DIR / f"{user_id}_{file.filename}"
    with open(out_path, "wb") as f:
        shutil.copyfileobj(file.file, f)
    u.resume_path = str(out_path)
    db.add(u)
    db.commit()
    return {"ok": True, "resume_path": u.resume_path}


@app.get("/users/{user_id}")
def get_user(user_id: int):
    db = SessionLocal()
    u = db.query(User).filter(User.id == user_id).first()
    if not u: raise HTTPException(404)
    return {"id": u.id, "name": u.name, "email": u.email, "resume_path": u.resume_path}
```

3. Run server:

```
uvicorn app.main:app --reload --port 8000
```

Open http://localhost:8000/docs — you'll see the interactive API docs.

What you have now: user creation endpoint, resume upload saved locally. No auth yet (later we add JWT).

Milestone 3 — resume parsing (simple) + project extraction

For zero cost, start with PyPDF2 to get text and simple heuristics to find education, projects, skills. Later you can plug in CV_extract.

Add parser backend/app/parse_resume.py:

```
# backend/app/parse_resume.py
from PyPDF2 import PdfReader
import re


def extract_text_from_pdf(path):
    reader = PdfReader(path)
    pages = [p.extract_text() for p in reader.pages]
    return "\n".join(p for p in pages if p)


def simple_resume_parser(text):
    # skills: look for a Skills: section or capitalized words separated by commas
    skills = []
    skills_match = re.search(r"(skills|technical skills|skillset)[:\n](.*?)(?:\n\n|\n[A-Z])", text, re.I|re.S)
    if skills_match:
        skills = [s.strip() for s in re.split(r",|\n|·|-", skills_match.group(2)) if s.strip()]
    # projects: find lines containing 'project' or 'project:' lines
    projects = re.findall(r"(?:project[:\s-]*)(.+)", text, re.I)
    # fallback: lines with 'Project' or 'Projects'
    return {"skills": skills[:30], "projects": projects[:10], "raw": text[:5000]}
```

Integrate into /users/{id}/parse_resume endpoint to return parsed fields. This gives you structured data fast.

Milestone 4 — question generator (template driven)

We'll generate questions from resume fields and also include core domain questions (OOP, DBMS, OS, CN). This is zero-cost and deterministic.

Create backend/app/question_gen.py:

```
# backend/app/question_gen.py
COMMON_TOPICS = {
    "OOP": [
        "Explain the four pillars of OOP with examples.",
        "What is polymorphism? Give a code example in any language.",
        "How does inheritance differ from composition?"
    ],
    "DBMS": [
        "Explain normalization and why it's important.",
        "What is an index and when should you use it?",
        "Explain ACID properties."
    ],
    "OS": [
        "What is a process vs a thread?",
        "Explain context switching and why it matters.",
        "Describe deadlock and ways to prevent it."
    ],
    "CN": [
        "Explain the TCP three-way handshake.",
        "What is latency vs bandwidth?",
        "Explain how DNS resolution works."
    ]
}


def questions_from_resume(parsed):
    qs = []
    for proj in parsed.get("projects", []):
        qs.append(f"Tell me about this project: {proj[:120]}. What was your role and key technical challenges?")
        qs.append(f"Which module in this project was the hardest and why? How did you test it?")
    for s in parsed.get("skills", [])[:10]:
        qs.append(f"You listed '{s}'. Explain one important concept related to {s}.")
    # add common topics
    for t in ["OOP","DBMS","OS","CN"]:
        qs.extend(COMMON_TOPICS[t][:2])
    return qs
```

Endpoint /users/{id}/start_interview will call parser -> question_gen -> create Interview record in DB and return first question.

Milestone 5 — frontend: React app with avatar + audio play + WebSocket

Create frontend with Vite:

1. In frontend:

```
cd ../frontend
npm create vite@latest . -- --template react
npm install
npm install three @react-three/fiber @react-three/drei
# optional avatoon package if available:
# npm i avatoon-react (example; replace with actual package if exist)
```

2. Simple avatar component (frontend/src/AvatarScene.jsx) that loads a GLB model and exposes a setViseme(name, strength) API.

```
// frontend/src/AvatarScene.jsx
import React, { useRef, useEffect } from "react";
import { Canvas } from "@react-three/fiber";
import { OrbitControls, useGLTF } from "@react-three/drei";

function AvatarInner({ modelUrl, viseme }) {
  const gltf = useGLTF(modelUrl);
  // NOTE: real lip-sync requires model morph targets named as visemes
  useEffect(() => {
    if (!gltf) return;
    // Example: set morphTargetInfluences for viseme blendshapes
    // gltf.scene... depending on model specifics
  }, [viseme, gltf]);
  return <primitive object={gltf.scene} />;
}

export default function AvatarScene({ modelUrl, viseme }) {
  return (
    <Canvas style={{ height: 500 }}>
      <ambientLight />
      <pointLight position={[10, 10, 10]} />
      <AvatarInner modelUrl={modelUrl} viseme={viseme} />
      <OrbitControls />
    </Canvas>
  );
}
```

3. WebSocket connection to backend to receive audio chunks & viseme events:

```
// frontend/src/InterviewPage.jsx (simplified)
import React, { useState, useEffect, useRef } from "react";
import AvatarScene from "./AvatarScene";

export default function InterviewPage({ userId }) {
 const wsRef = useRef();
 const audioRef = useRef(new Audio());
 const [viseme, setViseme] = useState({}); // {visemeName: strength}

 useEffect(() => {
  wsRef.current = new WebSocket("ws://localhost:8000/ws/interview");
  wsRef.current.onmessage = (ev) => {
   // assume backend sends JSON events or binary audio
   // event = {type: "viseme", name: "aa", strength: 0.8} or {type:"audio", data: base64}
   const data = JSON.parse(ev.data);
   if (data.type === "viseme") {
    setViseme({ [data.name]: data.strength });
   } else if (data.type === "audio") {
    // create blob and play
    const audioBlob = base64ToBlob(data.data, "audio/wav");
    const url = URL.createObjectURL(audioBlob);
    audioRef.current.src = url;
    audioRef.current.play();
   }
  };
  return () => wsRef.current.close();
 }, []);
 return (
  <div>
   <h2>Interview</h2>
   <AvatarScene modelUrl="/models/avatar.glb" viseme={viseme} />
   <button onClick={() => wsRef.current.send(JSON.stringify({ type: "start", userId }))}>
    Start Interview
```

```
      </button>
    </div>
  );
}


function base64ToBlob(base64, type='application/octet-stream'){
  const b = atob(base64);
  let n = b.length;
  const u8 = new Uint8Array(n);
  while(n--) u8[n] = b.charCodeAt(n);
  return new Blob([u8], { type });
}
```

Note: This uses simple base64 audio. For real streaming, use WebRTC or binary WebSocket frames — but base64 is easiest to prototype.

Milestone 6 — TTS + viseme generation (browser first)

For zero cost and fastest path, start with in-browser TTS that provides visemes:

- Use HeadTTS if you can run it in browser; it can generate audio + viseme timestamps without server processing. That avoids server audio streaming complexity.

Flow:

1. Backend sends interview text (question) via WebSocket to frontend.
2. Frontend runs HeadTTS to synthesize audio + viseme timeline locally.
3. Frontend plays audio and drives avatar visemes synchronously using the returned viseme timeline.

If HeadTTS isn't available on your machine, fallback: use SpeechSynthesis WebAPI for audio (no viseme), and generate simple mouth open/close animations on the avatar timed to audio duration.

Example pseudo code on frontend when receiving a question:

```
// receive question text
const result = await headtts.synthesize(questionText); // returns { audioBlob, visemes: [{time, name, strength}] }
audioRef.current.src = URL.createObjectURL(result.audioBlob);
audioRef.current.play();
// play visemes in sync:
```

```
result.visemes.forEach(v => {
  setTimeout(() => setViseme({[v.name]: v.strength}), v.time*1000);
});
```

Milestone 7 — speech → text (student answers)

Zero-cost alternative:

- Record audio in browser (MediaRecorder) and upload to backend for transcription. For offline local transcription use a Whisper variant you can run locally later. For a beginner, you can transcribe via a simple SpeechRecognition WebAPI (Chrome only), but accuracy is limited.

Example (browser):

```
// record audio, then upload to /transcribe
const stream = await navigator.mediaDevices.getUserMedia({ audio: true });
const recorder = new MediaRecorder(stream);
recorder.start();
// stop after user stops
recorder.ondataavailable = async e => {
  const fd = new FormData();
  fd.append('file', e.data, 'answer.wav');
  await fetch('http://localhost:8000/transcribe', { method: 'POST', body: fd });
}
```

Backend /transcribe endpoint (simple placeholder):

```
@app.post("/transcribe")
async def transcribe(file: UploadFile = File(...)):
    # save file -> pass to whisper or simple placeholder that returns a dummy transcription
    path = UPLOAD_DIR / file.filename
    with open(path, "wb") as f:
        shutil.copyfileobj(file.file, f)
    # TODO: call whisper/faster-whisper/fork here
    return {"text": "transcribed text placeholder"}
```

Later: replace placeholder with faster-whisper or whisper.cpp wrapper to run locally.

Milestone 8 — evaluation & scoring algorithm (basic, explainable)

You need credible, reproducible scoring without an LLM initially. Use rule-based scoring:

1. **Communication (0–20)**: measure average words per answer, filler words count (um/uh), speech rate variability. (Basic proxy: longer coherent answers → higher score.)
2. **Technical correctness (0–40)**: ask known answerable questions; compare student answers with expected keywords; give partial points for keyword matches.
3. **Resume knowledge (0–20)**: Did the student explain project details? Check presence of key keywords from the project description.
4. **Soft skills / Tips (qualitative)**: provide textual advice using templates.

Sample scoring snippet:

```
def score_answer(answer_text, expected_keywords):
    score = 0
    words = answer_text.lower()
    for kw in expected_keywords:
        if kw.lower() in words:
            score += 1
    # normalize
    return min(100, int((score / max(1, len(expected_keywords))) * 100))
```

Store per question scores and average them to produce overall metrics.

Milestone 9 — dashboard & storing results

Add Interview table in DB (user_id, started_at, ended_at, transcript, score_json). Provide endpoints to list a user's past interviews and show detailed scoring.

Frontend: student dashboard lists interviews and opens detailed view with timeline, transcript, score, and playback of recorded audio.

Milestone 10 — polishing & swap to real components later

After prototype works locally, you can upgrade components one by one:

- Replace simple parser with CV_extract for structured data.
- Replace template question generator with a hosted LLM (self-hosted Teuken-7B / Apertus) if you have GPU or a cloud budget.
- Replace browser TTS with ZeroVOX for better voices + visemes. Alternatively, use HeadTTS in browser for free experiments.

- Replace SQLite with PostgreSQL when moving to production.

Helpful implementation tips and exact commands you'll use a lot

- Start backend dev server:

```
cd backend
. .venv/bin/activate   # or equivalent on Windows
uvicorn app.main:app --reload --port 8000
```

- Start frontend dev server:

```
cd frontend
npm run dev
```

- Add CORS in FastAPI so React (port 5173) can call API:

```
from fastapi.middleware.cors import CORSMiddleware
app.add_middleware(CORSMiddleware, allow_origins=["http://localhost:5173"],
allow_methods=["*"], allow_headers=["*"])
```

- Capture audio in browser: navigator.mediaDevices.getUserMedia({audio:true}) and MediaRecorder.
- Basic JWT auth (later): use python-jose and passlib for password hashing.

Minimal file/folder layout (recommended)

```
deepseek-interview-bot/
 ├─ backend/
 │   ├─ app/
 │  │   ├─ main.py
 │  │   ├─ parse_resume.py
 │  │   ├─ question_gen.py
 │  │   ├─ models.py      # SQLAlchemy models (User, Interview)
 │  │   └─ utils.py
 │  └─ .venv/
 ├─ frontend/
 │   ├─ src/
 │  │   ├─ main.jsx
```

```
| |  ├– App.jsx
| |  ├– InterviewPage.jsx
| | └─ AvatarScene.jsx
| └─ package.json
└─ README.md
```

Roadmap: what to learn and in what order (specific learning targets)

1. Basic Git + VS Code navigation. (1 day)
2. JavaScript + React basics (components, state, props) (2–3 days).
3. Python basics & FastAPI quickstart (2–3 days).
4. SQLAlchemy + simple database CRUD (2 days).
5. Browser APIs: MediaRecorder, WebSockets (2 days).
6. 3D in React: @react-three/fiber basics (2–4 days).
7. Resume parsing basics (text extraction, regex) then CV_extract later.
8. Optional: Whisper / whisper.cpp for offline STT; running small LLMs (advanced).

Limitations & realistic expectations (honest)

- **LLMs & high-quality TTS with visemes**: hosting large models (7B+) or advanced TTS that produce phoneme-accurate visemes requires GPUs or paid cloud. For a zero-cost version, use rule-based question generation and in-browser TTS (HeadTTS) or simple animations.
- **Accuracy of automated scoring**: initial rule-based scoring is blunt. Consider showing qualitative feedback and allow human override.
- **Browser lip sync models**: morph targets and viseme mapping require 3D models made with compatible blendshapes. You may need to source or create a .glb with named morph targets.

Next immediate steps (do these now — exact commands)

1. Create repo and start backend:

```
mkdir deepseek-interview-bot && cd deepseek-interview-bot
git init
mkdir backend frontend
# setup backend
cd backend
```

```
python -m venv .venv
source .venv/bin/activate    # or Windows equivalent
pip install fastapi uvicorn sqlalchemy pydantic python-multipart aiofiles PyPDF2
mkdir app
# create the main.py from earlier
uvicorn app.main:app --reload --port 8000
```

2. Create frontend:

```
cd ../frontend
npm create vite@latest . -- --template react
npm install
npm install three @react-three/fiber @react-three/drei
npm run dev
```

3. Test creating a user & uploading a PDF using http://localhost:8000/docs (the FastAPI UI).