

K_1 - SE_2

Thanks for sharing the files. Here's a full **assessment of the current backend progress**, based on the extracted files and your detailed summary:

✅ What's Already Done (And Well Done)

✓ Core Backend Functionalities

- `main.py` : Well-structured FastAPI backend with:
 - Resume upload
 - Groq-powered question generation
 - Question-by-question delivery
 - Video-based stress analysis
 - Session-level stress score averaging

✓ Supabase Integration

- Storing resumes, interview questions, and stress metrics with correct relational design

✓ Emotion-Stress Model

- MobileNetV2 model (`emotion_stress_model.h5`) integrated and functional
- Input handling and frame processing via OpenCV
- Emotions mapped to quantitative stress levels
- Stress scoring per question + per session ✓

✓ Requirements & Environment

- `.env` with proper keys (redacted), and `requirements.txt` for easy setup
 - Resume sample (`resume-teja.pdf`) included for testability
-

🔍 Gaps & What's Missing

1 ❌ No Modularization or Clean Folder Structure

All files are in a flat directory. Ideally:

```
bash
```

```
/app
```

```
|— api/          → FastAPI routes
|— core/         → logic for resume parsing, Groq generation, stress processing
|— models/       → database schemas
|— utils/        → helper functions
|— config.py     → .env loader
```

2 ❌ No Frontend Integration

The system:

- Assumes frontend will capture video and POST it to `/analyze-stress`
- Lacks support for real-time or in-browser camera streaming (e.g., WebRTC base64 or MJPEG)

3 ❌ No Interview Result Report

While stress analysis is done:

- There is **no unified report endpoint** to return:
 - All questions
 - Their stress score
 - A confidence level
 - Technical vs HR performance summary
 - Recommendations

4 ❌ Question Answer Handling is Missing

- No endpoint yet for:
 - Submitting an answer
 - Saving text/audio of answer
 - Performing answer evaluation (LLM/NLP) → for future scoring
-

Roadmap – What More to Do (Clear Steps)

PHASE 1: Backend Finalization

A. Restructure Code into Modules

- Organize by responsibility:
 - `routes/question_routes.py`
 - `services/stress_analysis.py`
 - `utils/resume_parser.py`
 - `config/env.py`

B. Add New Endpoints

- `POST /submit-answer/{session_id}/{question_number}`
→ Accept text or audio and store it (future NLP analysis)
 - `GET /final-report/{session_id}`
→ Combines:
 - All questions
 - Each stress level
 - Overall average
 - Final feedback (use GPT: "Based on the answers + stress level, this candidate needs to...")
-

PHASE 2: Frontend Integration

- React/Next.js frontend to:

- Upload resume
- Show questions one by one
- Record video after each answer (WebRTC + FileReader or Blob)
- Display stress feedback live or after each question
- Show final report graph




PHASE 3: Advanced Features (Optional for Hackathon/Submission)

Feature	Value
✓ Text-Based Answer Evaluation (LLM scoring)	Uses GPT/Groq to rate answer quality
✓ Real-Time Stress Heatmap (per second)	Overlay confidence chart
✓ Personalized Learning Plan	Based on weak areas from answers + stress
✓ AI Panel Interview	3 persona-based GPT interviewers

Summary: Is He on the Right Path?

- ✓ YES – The backend core is strong, API-driven, and modular enough for scaling.
 - ✓ Model integration is working, even if basic — you’ve reached a functioning MVP.
 - ⚠ Just needs clean structure, final reporting, and frontend tie-in for full polish.
-

Would you like me to:

-  Generate a modular folder structure + refactored template?
-  Design the `/final-report` endpoint + JSON output format?
-  Give frontend code to record webcam + send video?

Let’s lock the rest of it down. You’re very close!

Here is a **modular folder structure** and a fully **refactored FastAPI backend template** for your *Mock Interview Project*. This makes your code clean, scalable, and production-ready.

✓ Folder Structure

bash

```
mock_interview_backend/
|
├─ app/
|   └─ api/
|       ├── routes_resume.py           # Resume upload & parsing
|       ├── routes_questions.py        # Question generation & retrieval
|       ├── routes_stress.py           # Stress analysis endpoints
|       └─ routes_misc.py              # Health checks, Supabase test
|
|   └─ core/
|       ├── resume_parser.py           # Extract text and info from PDF
|       ├── question_generator.py      # Uses Groq to generate Qs
|       ├── stress_analyzer.py         # Video → emotion model → stress
|       └─ report_generator.py         # Compile session report
|
|   └─ models/
|       └─ supabase_schema.py          # Table/column references
|
|   └─ utils/
|       └─ supabase_client.py          # Reusable Supabase methods
|
|   └─ config/
|       └─ env.py                      # Loads environment variables
|
|   └─ main.py                         # FastAPI app entry point
|
├─ models/
|   └─ emotion_stress_model.h5         # Pre-trained emotion model
|
├─ .env                               # API keys (excluded in prod)
└─ requirements.txt                   # Dependencies
```

└─ README.md
└─ resume-teja.pdf

Project setup guide

Test resume

Refactored main.py

python

```
from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from app.api import routes_resume, routes_questions, routes_stress, routes_misc
from app.config.env import load_env

app = FastAPI()

# CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Load ENV
load_env()

# Register Routes
app.include_router(routes_misc.router)
app.include_router(routes_resume.router)
app.include_router(routes_questions.router)
app.include_router(routes_stress.router)
```

Sample: routes_questions.py

python

```

from fastapi import APIRouter
from app.core.question_generator import generate_questions_from_resume

router = APIRouter()

@router.post("/generate-questions/{mock_user_id}/{resume_id}")
async def generate_questions(mock_user_id: str, resume_id: str):
    return await generate_questions_from_resume(mock_user_id, resume_id)

```

Sample: core/question_generator.py

python

```

from app.utils.supabase_client import supabase
from app.core.resume_parser import extract_text_from_pdf
from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate
import os

groq_api_key = os.getenv("GROQ_API_KEY")
client = ChatOpenAI(
    openai_api_key=groq_api_key,
    model="mixtral-8x7b-32768",
    base_url="https://api.groq.com/openai/v1"
)

async def generate_questions_from_resume(user_id, resume_id):
    # 1. Fetch resume PDF from Supabase
    resume_url = supabase.get_resume_url(resume_id)
    resume_text = extract_text_from_pdf(resume_url)

    # 2. Format prompt and call Groq
    prompt_template = PromptTemplate(
        input_variables=["resume_text", "job_role"],
        template="""
        Resume Content: {resume_text}
        Job Role: Software Engineer

```

```

        Generate:
        - 3 Technical Questions
        - 3 HR Questions
        - 2 Situational Questions
        - 1 Surprise Question
        """

    )
    prompt = prompt_template.format(resume_text=resume_text, job_role="Software Engineer")
    result = client.predict(prompt)

    # 3. Save result to DB
    supabase.save_questions(user_id, resume_id, result)
    return {"status": "success", "questions": result}

```

utils/supabase_client.py

```

python

from supabase import create_client
import os

supabase = create_client(
    os.getenv("SUPABASE_URL"),
    os.getenv("SUPABASE_KEY")
)

# Example helper
def get_resume_url(resume_id):
    result = supabase.table("mock_interview_resumes").select("*").eq("id", resume_id).execute()
    return result.data[0]["file_path"]

```

config/env.py

```
python
```



```
from dotenv import load_dotenv
import os

def load_env():
    load_dotenv()
    print("🔑 Environment variables loaded.")

required_vars = ["SUPABASE_URL", "SUPABASE_KEY", "GROQ_API_KEY"]
for var in required_vars:
    if not os.getenv(var):
        raise EnvironmentError(f"Missing env var: {var}")
```

✅ Benefits of This Structure

- Easier to **test, scale, and debug**
- Each file has **one responsibility**
- Future devs (or your frontend teammate) will **understand it instantly**
- Allows easy **CI/CD deployment** on Vercel, AWS, or GCP

Would you like me to:

- 🎯 Zip and give this boilerplate template?
- ✉️ Add frontend API calls for each route?

You're now building like a **pro startup backend**. Let's finish strong! 💪