FAB: The Complete Developer's Guide

Full-Stack AI-Powered Interview Readiness System

A Production-Ready Guide from Zero to Deployment

Stack: FastAPI, React/Next.js, Gemini API, RAG, Fine-tuning, Free GPU (Colab/Kaggle)

Table of Contents

Executive Summary

FAB is NOT:

- Another AI chatbot
- A confidence booster
- An "easy interview cracker"
- A memorization tool

FAB IS:

- A **brutally honest** competence verification system
- A **mirror** that exposes shallow understanding
- A **repair engine** that forces skill acquisition
- An **evolution system** that learns from real interview outcomes

Core Philosophy

> "FAB doesn't help you sound smart. It helps you become defensible."

PART 1: SYSTEM ARCHITECTURE OVERVIEW

1.1 Core Modules

Module	Purpose	Output
GitHub Evaluator	Extract real project signal from noise	Structured project understanding, architecture analysis
Resume Evaluator	Detect overclaiming before interviewers do	Claim classification: Earned / Weak / Misleading / Remove
Interview Interrogator (Mode 1)	Break illusions under pressure	Failure points, confidence gaps, collapse detection
Gap Repair Engine (Mode 2)	Convert failure into competence	Defensible framing / Learning drills / Project prescriptions
Role-Specific Optimizer	Context-aware evaluation	Adjusted scoring and question aggression per role
GitHub Improvement Advisor	Stop wasted effort	Max 2 high-signal project suggestions
Evolution System	Never get outdated	RAG memory + failure pattern mining + selective fine-tuning
Live Interview Simulator	Real-time pressure testing	Stream-based questioning with adaptive difficulty

Table 1: FAB Core Modules

1.2 Technical Features (What Makes FAB Unique)

Feature 1: Interview Outcome Feedback Loop

- Users report real interview results (pass/fail, questions asked, where they failed)
- System stores: Question → Outcome patterns
- Feeds RAG, not blind training
- Ground truth data, not opinions

Feature 2: Failure Pattern Miner

- Analyzes: "What kinds of answers fail for backend interns?"
- Correlates: Redis explanations with rejection rates
- Identifies: Which project types stop getting questioned
- Updates evaluation rubrics automatically

Feature 3: Interview Drift Detector

- Tracks: Company X now asks Y questions
- Alerts: "Your prep is outdated for this role"
- Adapts: Question generation to current trends
- Sources: User reports + public interview data

Feature 4: Live Code Review Simulator

- GitHub code → AI reviews like senior engineer
- Asks: "Why this pattern?" "Where does this break?"
- Forces: Verbal defense of code choices
- Simulates: Real technical interview scenarios

Feature 5: Multi-Modal Explanation Evaluator

- Analyzes: Voice tone, filler words, confidence patterns
- Detects: "I think", "basically", "kind of" usage
- Scores: Communication clarity separate from technical depth
- Provides: Speech pattern improvement drills

Feature 6: Adversarial Question Generator

- Not generic questions
- Hostile follow-ups like real interviewers
- Escalates when answers are vague
- Corner-case scenarios that expose gaps

PART 2: PHASE-BY-PHASE IMPLEMENTATION

✅ Phase 0: Prerequisites & Environment Setup

Duration: Days 1-2 (2 days)

 Learning Objectives

- Understand the FAB tech stack (FastAPI, React, Gemini, RAG)
- Set up local development environment
- Configure free GPU services (Google Colab, Kaggle)

 Environment Setup

You must install these tools on your computer before starting:

1. **Python 3.10+:** Download Python
2. **Node.js v18+:** Download Node.js
3. **Git:** Download Git
4. **VS Code:** Download VS Code
5. **Docker:** Download Docker (for local testing)

Install global tools:

Python tools

```
pip install --upgrade pip
pip install virtualenv
```

Node tools

```
npm install -g pnpm
npm install -g vercel # For frontend deployment
```

 API Keys & Accounts Setup

Required accounts (all FREE):

1. **Google AI Studio:** Get Gemini API key
   - Visit: https://ai.google.dev/
   - Create API key (free tier: 60 requests/min)
2. **GitHub Personal Access Token:**
   - Settings → Developer Settings → Personal Access Tokens
   - Scopes: `repo`, `user:email`
3. **Google Colab Account:**
   - Visit: https://colab.research.google.com/
   - Sign in with Google account
4. **Kaggle Account:**
   - Visit: https://www.kaggle.com/
   - Settings → API → Create New Token

 Project Structure

```
fab-system/
├── backend/ # FastAPI backend
├── frontend/ # React/Next.js frontend
├── notebooks/ # Jupyter notebooks for GPU tasks
├── scripts/ # Automation scripts
```

├── docs/ # Documentation
└── tests/ # Integration tests

✅ Phase 0.5: The Developer's Toolkit

Duration: Day 2 (1 day)

Purpose

Set up project tracking and health monitoring before writing code.

Step 1: Create Project Status Log

Using Gemini CLI (Antigravity method):

gemini -p "Create a new file named 'PROJECT_STATUS.md'.
The file must have:

1. Main title: '# FAB Project Status Report'
2. Status legend: '✅ Completed', '🔄 In Progress', '⬜ Not Started', '🚫 Blocked'
3. Project Roadmap section with all 9 phases
4. Completion tracking: Date completed, Key learnings, Blockers faced
5. Testing Results section with expected vs actual outcomes
6. Next Steps section for each phase" > PROJECT_STATUS.md

Step 2: Create Health Check Script

File: scripts/health-check.sh

#!/bin/bash

FAB Health Checker

Run: bash scripts/health-check.sh

RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
NC='\033[0m'

echo -e "$YELLOW---FABSystemHealthCheck---${NC}"
PASSED=true

Check Python

if command -v python3 &> /dev/null; then
echo -e "$GREEN[PASS]${NC} Python3 installed"
else
echo -e "$RED[FAIL]${NC} Python3 not found"
PASSED=false
fi

Check .env files

if [-f "backend/.env"]; then
echo -e "$GREEN[PASS]${NC} Backend .env found"

```
if grep -q "GEMINI_API_KEY=your" backend/.env; then
    echo -e "${RED}[FAIL]${NC} GEMINI_API_KEY not configured"
    PASSED=false
else
    echo -e "${GREEN}[PASS]${NC} GEMINI_API_KEY configured"
fi
```

else
echo -e "$RED[FAIL]${NC} Backend .env not found"
PASSED=false
fi

Check virtual environment

if [-d "backend/venv"]; then
echo -e "$GREEN[PASS]${NC} Virtual environment exists"
else
echo -e "$YELLOW[WARN]${NC} Virtual environment not found. Run: cd backend && python3 -m venv venv"
fi

echo "----------"
if ["$PASSED" = true]; then echo -e "${GREEN}Health check passed!${NC}"
else echo -e "${RED}Health check failed. Fix issues above.${NC}"
exit 1
fi

Step 3: Backend Initialization

Create project structure

mkdir -p fab-system/{backend,frontend,notebooks,scripts,docs,tests}
cd fab-system/backend

Create virtual environment

python3 -m venv venv
source venv/bin/activate # On Windows: venv\Scripts\activate

Create .env.example

cat > .env.example << 'EOF'

API Keys

GEMINI_API_KEY=your_gemini_api_key_here
GITHUB_TOKEN=your_github_token_here

Database (SQLite for development)

DATABASE_URL=sqlite:///./fab.db

JWT Secret

JWT_SECRET=your-super-secret-jwt-key-change-in-production
JWT_ALGORITHM=HS256
JWT_EXPIRY=7d

Server

HOST=0.0.0.0
PORT=8000
ENVIRONMENT=development

CORS

CORS_ORIGINS=http://localhost:3000,http://localhost:5173

Rate Limiting

RATE_LIMIT_PER_MINUTE=60

Colab Integration

COLAB_NOTEBOOK_URL=your_colab_notebook_url
KAGGLE_USERNAME=your_kaggle_username
KAGGLE_KEY=your_kaggle_api_key
EOF

Copy to actual .env

cp .env.example .env
echo "⚠ IMPORTANT: Edit backend/.env and add your real API keys!"

Pre-Phase Checklist

- [ ] All tools installed (Python, Node, Git, VS Code, Docker)
- [ ] API keys obtained (Gemini, GitHub, Kaggle)
- [ ] Project structure created
- [ ] Health check script runs successfully
- [ ] .env file configured with real keys
- [ ] Virtual environment activated

Phase 1: Foundation Layer (GitHub Evaluator)

Duration: Days 3-5 (3 days)

Learning Objectives

- Fetch and parse GitHub repositories intelligently
- Extract architectural decisions and tech stack
- Detect boilerplate vs original code
- Build structured project understanding (not just scraping)

Key Dependencies

Inside backend venv

pip install fastapi0.104.1
pip install uvicorn[standard]0.24.0
pip install pydantic2.5.0
pip install pydantic-settings2.1.0
pip install sqlalchemy2.0.23
pip install alembic1.12.1
pip install httpx0.25.1
pip install python-jose[cryptography]3.3.0
pip install passlib[bcrypt]1.7.4
pip install python-multipart0.0.6
pip install google-generativeai0.3.1
pip install gitpython3.1.40
pip install pygments2.17.2 # Code analysis
pip install radon6.0.1 # Complexity analysis

 Step-by-Step Guide

Step 1.1: Create Project Workspace

cd backend

Create directory structure

mkdir -p app/{api,core,db,models,services,schemas,utils}
touch app/**init**.py
touch app/main.py

Step 1.2: Core Configuration

File: app/core/config.py

from pydantic_settings import BaseSettings
from typing import List

class Settings(BaseSettings):
# API Keys
GEMINI_API_KEY: str
GITHUB_TOKEN: str

```
# Database
DATABASE_URL: str = "sqlite:///./fab.db"

# JWT
JWT_SECRET: str
JWT_ALGORITHM: str = "HS256"
JWT_EXPIRY: str = "7d"

# Server
HOST: str = "0.0.0.0"
PORT: int = 8000
ENVIRONMENT: str = "development"

# CORS
CORS_ORIGINS: List[str] = ["http://localhost:3000"]

class Config:
    env_file = ".env"
    case_sensitive = True
```

settings = Settings()

Step 1.3: Database Models

File: app/models/user.py

from sqlalchemy import Column, String, DateTime, JSON, Boolean
from sqlalchemy.sql import func
from app.db.base import Base
import uuid

class User(Base):
tablename = "users"

```
id = Column(String, primary_key=True, default=lambda: str(uuid.uuid4()))
email = Column(String, unique=True, nullable=False, index=True)
hashed_password = Column(String, nullable=False)
full_name = Column(String)
github_username = Column(String, index=True)

# User metadata
is_active = Column(Boolean, default=True)
created_at = Column(DateTime(timezone=True), server_default=func.now())
updated_at = Column(DateTime(timezone=True), onupdate=func.now())
last_login = Column(DateTime(timezone=True))

# Profile data
target_roles = Column(JSON)  # ["Backend Engineer", "Full-Stack"]
experience_level = Column(String)  # "Student", "Intern", "Junior"
```

File: app/models/github_analysis.py

from sqlalchemy import Column, String, DateTime, JSON, Float, ForeignKey, Text
from sqlalchemy.sql import func
from app.db.base import Base
import uuid

class GitHubAnalysis(Base):
tablename = "github_analyses"

```
id = Column(String, primary_key=True, default=lambda: str(uuid.uuid4()))
user_id = Column(String, ForeignKey("users.id"), nullable=False)
repo_url = Column(String, nullable=False)
repo_name = Column(String, nullable=False)
```

```
    # Analysis results
    tech_stack = Column(JSON)  # {"languages": [], "frameworks": [], "tools": []}
    architecture_summary = Column(Text)
    project_type = Column(String)  # "Web App", "CLI Tool", "Library", etc.

    # Signal quality scores (0-100)
    originality_score = Column(Float)  # High if not boilerplate
    depth_score = Column(Float)        # Code complexity and depth
    commit_consistency_score = Column(Float)
    readme_quality_score = Column(Float)

    # Detected patterns
    boilerplate_percentage = Column(Float)
    tutorial_fingerprints = Column(JSON)  # List of detected tutorial sources

    # Weaknesses and risks
    weaknesses = Column(JSON)  # ["No error handling", "No tests"]
    architectural_risks = Column(JSON)
    overclaimed_features = Column(JSON)

    # Metadata
    analyzed_at = Column(DateTime(timezone=True), server_default=func.now())
    analysis_version = Column(String, default="1.0")
```

Step 1.4: GitHub Service (Core Logic)

File: app/services/github_evaluator.py

```
import httpx
import google.generativeai as genai
from typing import Dict, List, Optional
from app.core.config import settings
import base64
import re
from radon.complexity import cc_visit
from radon.metrics import mi_visit

class GitHubEvaluator:
def __init__(self):
self.github_token = settings.GITHUB_TOKEN
self.headers = {
"Authorization": f"token {self.github_token}",
```

"Accept": "application/vnd.github.v3+json"
}
genai.configure(api_key=settings.GEMINI_API_KEY)
self.model = genai.GenerativeModel('gemini-pro')

```
async def analyze_repository(self, repo_url: str) -> Dict:
    """
    Main entry point: Analyze GitHub repository
    """
    # Extract owner/repo from URL
    owner, repo = self._parse_repo_url(repo_url)

    # Fetch repository metadata
    repo_data = await self._fetch_repo_metadata(owner, repo)

    # Fetch important files
    files_data = await self._fetch_important_files(owner, repo)

    # Analyze tech stack
    tech_stack = self._analyze_tech_stack(repo_data, files_data)

    # Analyze code quality
    quality_scores = await self._analyze_code_quality(files_data)

    # Detect boilerplate/tutorial code
    boilerplate_analysis = await self._detect_boilerplate(files_data)

    # Get AI architectural analysis
    architecture_analysis = await self._ai_architecture_analysis(
        repo_data, files_data, tech_stack
    )

    return {
        "repo_name": repo_data["name"],
        "repo_url": repo_url,
        "tech_stack": tech_stack,
        "architecture_summary": architecture_analysis["summary"],
        "project_type": architecture_analysis["project_type"],
        "originality_score": boilerplate_analysis["originality_score"],
```

```
        "depth_score": quality_scores["depth_score"],
        "commit_consistency_score": quality_scores["commit_score"],
        "readme_quality_score": quality_scores["readme_score"],
        "boilerplate_percentage": boilerplate_analysis["boilerplate_percentage"],
        "tutorial_fingerprints": boilerplate_analysis["detected_tutorials"],
        "weaknesses": architecture_analysis["weaknesses"],
        "architectural_risks": architecture_analysis["risks"],
        "overclaimed_features": architecture_analysis["overclaimed_features"]
    }

def _parse_repo_url(self, url: str) -> tuple:
    """Extract owner/repo from GitHub URL"""
    pattern = r"github\.com/([^/]+)/([^/]+)"
    match = re.search(pattern, url)
    if not match:
        raise ValueError("Invalid GitHub URL")
    return match.group(1), match.group(2).replace(".git", "")

async def _fetch_repo_metadata(self, owner: str, repo: str) -> Dict:
    """Fetch repo metadata from GitHub API"""
    async with httpx.AsyncClient() as client:
        response = await client.get(
            f"https://api.github.com/repos/{owner}/{repo}",
            headers=self.headers
        )
        response.raise_for_status()
        return response.json()

async def _fetch_important_files(self, owner: str, repo: str) -> Dict:
    """
    Fetch strategically important files:
    - README.md
    - package.json / requirements.txt / Cargo.toml
    - Main source files (top 10 by LOC)
    """
    files = {}

    # Fetch README
```

```
    try:
        files["README"] = await self._fetch_file_content(owner, repo, "README.md
    except:
        files["README"] = None

    # Fetch dependency files
    dependency_files = [
        "package.json", "requirements.txt", "Cargo.toml",
        "go.mod", "pom.xml", "build.gradle"
    ]
    for dep_file in dependency_files:
        try:
            files[dep_file] = await self._fetch_file_content(owner, repo, dep_file)
            break  # Only need one
        except:
            continue

    # Fetch main source files (implementation needed)
    # TODO: Implement tree traversal and LOC ranking

    return files

async def _fetch_file_content(self, owner: str, repo: str, path: str) -> str:
    """Fetch file content from GitHub"""
    async with httpx.AsyncClient() as client:
        response = await client.get(
            f"https://api.github.com/repos/{owner}/{repo}/contents/{path}",
            headers=self.headers
        )
        response.raise_for_status()
        content = response.json()["content"]
        return base64.b64decode(content).decode('utf-8')

def _analyze_tech_stack(self, repo_data: Dict, files_data: Dict) -> Dict:
    """Extract tech stack from repo data and files"""
    tech_stack = {
        "languages": [],
        "frameworks": [],
```

```
        "tools": []
    }

    # From GitHub API languages
    # (Implementation depends on GitHub API response)

    # From dependency files
    if "package.json" in files_data and files_data["package.json"]:
        # Parse package.json for frameworks
        pass

    return tech_stack

async def _analyze_code_quality(self, files_data: Dict) -> Dict:
    """Analyze code complexity and quality metrics"""
    scores = {
        "depth_score": 0.0,
        "commit_score": 0.0,
        "readme_score": 0.0
    }

    # README quality (length, sections, examples)
    if files_data.get("README"):
        readme = files_data["README"]
        scores["readme_score"] = self._score_readme_quality(readme)

    # Code depth (complexity analysis with radon)
    # TODO: Implement complexity scoring

    return scores

def _score_readme_quality(self, readme: str) -> float:
    """Score README quality (0-100)"""
    score = 0.0

    # Length check (too short = bad)
    if len(readme) > 500:
        score += 20
```

```
        # Has sections
        sections = ["install", "usage", "example", "api", "contributing"]
        for section in sections:
            if section.lower() in readme.lower():
                score += 10

        # Has code examples
        if "```" in readme:
            score += 20

        return min(score, 100.0)

    async def _detect_boilerplate(self, files_data: Dict) -> Dict:
        """Detect if code is mostly boilerplate or copied from tutorials"""
        # Common boilerplate patterns
        boilerplate_patterns = [
            "create-react-app",
            "cookiecutter",
            "vue-cli",
            "express-generator",
            "create-next-app"
        ]

        detected_tutorials = []
        boilerplate_percentage = 0.0

        # Check README for tutorial mentions
        if files_data.get("README"):
            readme = files_data["README"].lower()
            for pattern in boilerplate_patterns:
                if pattern in readme:
                    detected_tutorials.append(pattern)

        # Heuristic: if detected patterns, assume 60-80% boilerplate
        if detected_tutorials:
            boilerplate_percentage = 70.0
        else:
```

```
            boilerplate_percentage = 20.0  # Some boilerplate always exists

        originality_score = 100.0 - boilerplate_percentage

        return {
            "boilerplate_percentage": boilerplate_percentage,
            "originality_score": originality_score,
            "detected_tutorials": detected_tutorials
        }

    async def _ai_architecture_analysis(
        self, repo_data: Dict, files_data: Dict, tech_stack: Dict
    ) -> Dict:
        """Use Gemini to analyze architecture and detect weaknesses"""

        # Build context for AI
        context = f"""
```

You are a strict senior engineer reviewing this GitHub project.

Repository: {repo_data.get('name')}
Description: {repo_data.get('description')}
Tech Stack: {tech_stack}

README Content:
{files_data.get('README', 'No README found')}

TASK:

1. Summarize the architecture in 2-3 sentences
2. Identify project type (Web App, CLI Tool, Library, etc.)
3. List weaknesses (missing error handling, no tests, security issues)
4. List architectural risks (scalability, coupling, data flow issues)
5. Identify overclaimed features (README says X but code shows Y)

Be BRUTALLY HONEST. Do NOT praise. Assume interviewer hostility.
Output JSON format:
{{
"summary": "...",
"project_type": "...",
"weaknesses": ["..."],
"risks": ["..."],
"overclaimed_features": ["..."]
}}
"""

```
    try:
        response = self.model.generate_content(context)
        # Parse JSON from response
        # TODO: Implement JSON extraction from Gemini response

        return {
            "summary": "Architecture analysis pending",
            "project_type": "Unknown",
            "weaknesses": [],
            "risks": [],
            "overclaimed_features": []
        }
    except Exception as e:
        print(f"AI analysis failed: {e}")
        return {
            "summary": "Analysis failed",
            "project_type": "Unknown",
            "weaknesses": ["Analysis error"],
            "risks": [],
            "overclaimed_features": []
        }
```

Step 1.5: API Endpoint

File: `app/api/github.py`

```
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from app.services.github_evaluator import GitHubEvaluator
from app.schemas.github import GitHubAnalysisRequest, GitHubAnalysisResponse
from app.db.session import get_db
from app.models.github_analysis import GitHubAnalysis

router = APIRouter(prefix="/github", tags=["GitHub Analysis"])

@router.post("/analyze", response_model=GitHubAnalysisResponse)
async def analyze_github_repo(
request: GitHubAnalysisRequest,
db: Session = Depends(get_db)
):
"""
Analyze a GitHub repository and return structured insights
```

"""
evaluator = GitHubEvaluator()

```
try:
    # Perform analysis
    analysis_result = await evaluator.analyze_repository(request.repo_url)

    # Save to database
    db_analysis = GitHubAnalysis(
        user_id=request.user_id,
        **analysis_result
    )
    db.add(db_analysis)
    db.commit()
    db.refresh(db_analysis)

    return analysis_result

except Exception as e:
    raise HTTPException(status_code=400, detail=str(e))
```

 Manual Test Guide (Phase 1)

1. **Start the backend:**

```
cd backend
source venv/bin/activate
uvicorn app.main:app --reload --port 8000
```

2. **Test GitHub Analysis (using cURL):**

```
curl -X POST http://localhost:8000/api/v1/github/analyze
-H "Content-Type: application/json"
-d '{
"repo_url": "https://github.com/your-username/your-repo",
"user_id": "test-user-123"
}'
```

3. **Expected Response:**

```
{
"repo_name": "your-repo",
"repo_url": "https://github.com/your-username/your-repo",
"tech_stack": {
"languages": ["Python", "JavaScript"],
```

"frameworks": ["FastAPI", "React"],
"tools": ["Docker"]
},
"architecture_summary": "Backend API with React frontend...",
"project_type": "Full-Stack Web Application",
"originality_score": 65.0,
"depth_score": 72.0,
"weaknesses": ["No authentication", "Missing error handling in API routes"],
"architectural_risks": ["No rate limiting", "Single database connection"]
}

4. **Validation Checklist:**
    - [ ] GitHub API rate limit not exceeded
    - [ ] Analysis completes within 10-15 seconds
    - [ ] Originality score is between 0-100
    - [ ] Weaknesses array is not empty (brutally honest)
    - [ ] Architecture summary is accurate

Phase 2: Resume Evaluator & Claim Verifier

Duration: Days 6-7 (2 days)

Learning Objectives

- Parse resumes (PDF, DOCX, plain text)
- Extract claims and skills
- Match resume claims against GitHub evidence
- Classify claims: Earned / Weak / Misleading / Remove

Additional Dependencies

```
pip install pypdf23.0.1
pip install python-docx1.1.0
pip install pdfplumber0.10.3
pip install spacy3.7.2
python -m spacy download en_core_web_sm
```

Step-by-Step Guide

Step 2.1: Resume Parser Service

File: `app/services/resume_parser.py`

```
import PyPDF2
import docx
import spacy
from typing import Dict, List
import re

class ResumeParser:
def __init__(self):
self.nlp = spacy.load("en_core_web_sm")
```

```
def parse_resume(self, file_path: str, file_type: str) -> Dict:
    """
    Parse resume and extract structured data
    """
    if file_type == "pdf":
        text = self._extract_pdf_text(file_path)
    elif file_type == "docx":
        text = self._extract_docx_text(file_path)
    else:
        with open(file_path, 'r') as f:
            text = f.read()

    # Extract sections
    sections = self._extract_sections(text)

    # Extract skills and claims
    skills = self._extract_skills(sections.get("skills", ""))
    projects = self._extract_project_claims(sections.get("projects", ""))
    experience = self._extract_experience_claims(sections.get("experience", ""))

    return {
        "raw_text": text,
        "skills": skills,
        "projects": projects,
        "experience": experience,
        "education": sections.get("education", "")
    }

def _extract_pdf_text(self, file_path: str) -> str:
    """Extract text from PDF"""
    with open(file_path, 'rb') as file:
        reader = PyPDF2.PdfReader(file)
        text = ""
        for page in reader.pages:
            text += page.extract_text()
    return text
```

```
def _extract_docx_text(self, file_path: str) -> str:
    """Extract text from DOCX"""
    doc = docx.Document(file_path)
    return "\n".join([para.text for para in doc.paragraphs])

def _extract_sections(self, text: str) -> Dict[str, str]:
    """
    Extract resume sections (Skills, Experience, Projects, Education)
    """
    sections = {}

    # Common section headers
    section_patterns = {
        "skills": r"(SKILLS|TECHNICAL SKILLS|TECHNOLOGIES)",
        "experience": r"(EXPERIENCE|WORK EXPERIENCE|EMPLOYMENT)",
        "projects": r"(PROJECTS|PERSONAL PROJECTS)",
        "education": r"(EDUCATION|ACADEMIC BACKGROUND)"
    }

    for section_name, pattern in section_patterns.items():
        match = re.search(f"{pattern}.*?(?=\\n[A-Z]{{3,}}|$)", text, re.DOTALL | re.IC
        if match:
            sections[section_name] = match.group(0)

    return sections

def _extract_skills(self, skills_text: str) -> List[str]:
    """Extract individual skills from skills section"""
    # Common delimiters: commas, bullets, pipes
    skills = re.split(r'[,•|]', skills_text)
    return [skill.strip() for skill in skills if len(skill.strip()) > 2]

def _extract_project_claims(self, projects_text: str) -> List[Dict]:
    """
    Extract project-specific claims
    Example: "Built scalable REST API using FastAPI"
    """
    claims = []
```

```
    # Use spaCy to extract sentences
    doc = self.nlp(projects_text)
    for sent in doc.sents:
        # Look for action verbs: Built, Developed, Implemented, Designed
        if re.search(r'\b(built|developed|implemented|designed|created|architec
                sent.text, re.IGNORECASE):
            claims.append({
                "claim": sent.text.strip(),
                "type": "project"
            })

    return claims

def _extract_experience_claims(self, experience_text: str) -> List[Dict]:
    """Extract experience-specific claims"""
    claims = []

    doc = self.nlp(experience_text)
    for sent in doc.sents:
        # Look for responsibility indicators
        if re.search(r'\b(responsible for|led|managed|coordinated|oversaw)\b',
                sent.text, re.IGNORECASE):
            claims.append({
                "claim": sent.text.strip(),
                "type": "experience"
            })

    return claims
```

Step 2.2: Claim Verifier Service

File: app/services/claim_verifier.py

```
from typing import Dict, List
import google.generativeai as genai
from app.core.config import settings

class ClaimVerifier:
def __init__(self):
```

```
genai.configure(api_key=settings.GEMINI_API_KEY)
self.model = genai.GenerativeModel('gemini-pro')
```

```
async def verify_claims(
    self,
    resume_data: Dict,
    github_analysis: Dict
) -> Dict:
    """
    Verify resume claims against GitHub evidence
    """
    verification_results = {
        "skills": await self._verify_skills(resume_data["skills"], github_analysis),
        "projects": await self._verify_project_claims(resume_data["projects"], githu
        "overall_credibility": 0.0,
        "dangerous_claims": [],
        "removal_recommendations": []
    }

    # Calculate overall credibility
    verification_results["overall_credibility"] = self._calculate_credibility(verifica

    return verification_results

async def _verify_skills(self, claimed_skills: List[str], github_data: Dict) -> List[D
    """
    Verify each skill claim
    Classification: EARNED / WEAK / MISLEADING / REMOVE
    """
    verified_skills = []

    github_tech_stack = github_data.get("tech_stack", {})
    github_languages = github_tech_stack.get("languages", [])
    github_frameworks = github_tech_stack.get("frameworks", [])

    for skill in claimed_skills:
        verification = {
            "skill": skill,
```

```
            "evidence_level": "NONE",
            "classification": "REMOVE",
            "reason": ""
        }

        # Check if skill appears in GitHub
        if skill.lower() in [lang.lower() for lang in github_languages]:
            verification["evidence_level"] = "STRONG"
            verification["classification"] = "EARNED"
            verification["reason"] = f"Found in {len(github_languages)} repositories"
        elif skill.lower() in [fw.lower() for fw in github_frameworks]:
            verification["evidence_level"] = "MODERATE"
            verification["classification"] = "WEAK"
            verification["reason"] = "Used but limited depth in codebase"
        else:
            verification["evidence_level"] = "NONE"
            verification["classification"] = "REMOVE"
            verification["reason"] = "No evidence in GitHub repositories"

        verified_skills.append(verification)

    return verified_skills

async def _verify_project_claims(self, project_claims: List[Dict], github_data: Di
    """
    Verify project-specific claims (e.g., "Built scalable backend")
    """
    verified_claims = []

    for claim in project_claims:
        # Use AI to verify claim against GitHub evidence
        verification = await self._ai_verify_claim(claim["claim"], github_data)
        verified_claims.append(verification)

    return verified_claims

async def _ai_verify_claim(self, claim: str, github_data: Dict) -> Dict:
    """
```

```
    Use Gemini to verify a specific claim against GitHub evidence
    """
    prompt = f"""
```

You are a hostile technical interviewer reviewing a resume claim.

CLAIM: "{claim}"

GITHUB EVIDENCE:

- Architecture: {github_data.get('architecture_summary', 'Unknown')}
- Tech Stack: {github_data.get('tech_stack', {})}
- Weaknesses: {github_data.get('weaknesses', [])}
- Originality Score: {github_data.get('originality_score', 0)}

TASK:
Classify this claim as one of:

1. EARNED - Strong evidence supports this claim
2. WEAK - Some evidence but overstated
3. MISLEADING - Evidence contradicts claim
4. REMOVE - No evidence at all

Provide harsh reasoning. Assume interviewer skepticism.

Output JSON:
{{
"claim": "...",
"classification": "EARNED|WEAK|MISLEADING|REMOVE",
"reasoning": "...",
"risk_level": "LOW|MEDIUM|HIGH"
}}
"""

```
    try:
        response = self.model.generate_content(prompt)
        # TODO: Parse JSON response
        return {
            "claim": claim,
            "classification": "WEAK",
            "reasoning": "Pending AI analysis",
            "risk_level": "MEDIUM"
        }
    except Exception as e:
        print(f"AI verification failed: {e}")
        return {
```

```
            "claim": claim,
            "classification": "UNKNOWN",
            "reasoning": "Verification failed",
            "risk_level": "HIGH"
        }

    def _calculate_credibility(self, verification_results: Dict) -> float:
        """
        Calculate overall resume credibility score (0-100)
        """
        total_claims = 0
        earned_count = 0

        # Count skill verifications
        for skill in verification_results["skills"]:
            total_claims += 1
            if skill["classification"] == "EARNED":
                earned_count += 1

        # Count project claim verifications
        for claim in verification_results["projects"]:
            total_claims += 1
            if claim["classification"] == "EARNED":
                earned_count += 1

        if total_claims == 0:
            return 0.0

        return (earned_count / total_claims) * 100.0
```

Step 2.3: API Endpoint

File: `app/api/resume.py`

```
from fastapi import APIRouter, Depends, File, UploadFile, HTTPException
from sqlalchemy.orm import Session
from app.services.resume_parser import ResumeParser
from app.services.claim_verifier import ClaimVerifier
from app.db.session import get_db
from app.models.resume_analysis import ResumeAnalysis
```

```
import os
import uuid

router = APIRouter(prefix="/resume", tags=["Resume Analysis"])

@router.post("/analyze")
async def analyze_resume(
file: UploadFile = File(...),
user_id: str = None,
github_analysis_id: str = None,
db: Session = Depends(get_db)
):
"""
Upload and analyze resume, verify claims against GitHub
"""
# Save uploaded file temporarily
file_id = str(uuid.uuid4())
file_extension = file.filename.split(".")[-1]
temp_path = f"/tmp/resume_{file_id}.{file_extension}"

    with open(temp_path, "wb") as f:
        f.write(await file.read())

    try:
        # Parse resume
        parser = ResumeParser()
        resume_data = parser.parse_resume(temp_path, file_extension)

        # Get GitHub analysis for verification
        github_analysis = db.query(GitHubAnalysis).filter(
            GitHubAnalysis.id == github_analysis_id
        ).first()

        if not github_analysis:
            raise HTTPException(status_code=404, detail="GitHub analysis not found")

        # Verify claims
        verifier = ClaimVerifier()
        verification_results = await verifier.verify_claims(
            resume_data,
            {
                "tech_stack": github_analysis.tech_stack,
                "architecture_summary": github_analysis.architecture_summary,
```

```
            "weaknesses": github_analysis.weaknesses,
            "originality_score": github_analysis.originality_score
        }
    )

    # Save analysis to database
    db_resume_analysis = ResumeAnalysis(
        user_id=user_id,
        github_analysis_id=github_analysis_id,
        resume_text=resume_data["raw_text"],
        extracted_skills=resume_data["skills"],
        verified_skills=verification_results["skills"],
        project_claims=verification_results["projects"],
        overall_credibility=verification_results["overall_credibility"]
    )
    db.add(db_resume_analysis)
    db.commit()

    return {
        "status": "success",
        "credibility_score": verification_results["overall_credibility"],
        "skills_verification": verification_results["skills"],
        "project_claims_verification": verification_results["projects"],
        "removal_recommendations": [
            skill["skill"] for skill in verification_results["skills"]
            if skill["classification"] == "REMOVE"
        ]
    }

finally:
    # Clean up temp file
    if os.path.exists(temp_path):
        os.remove(temp_path)
```

 Manual Test Guide (Phase 2)

1. **Prepare test resume** (save as `test_resume.pdf`):

JOHN DOE
Backend Engineer

SKILLS:
Python, FastAPI, Docker, Kubernetes, Redis, PostgreSQL, React, TypeScript

PROJECTS:

- Built scalable REST API using FastAPI and Docker
- Implemented microservices architecture with Redis caching
- Designed real-time chat application with WebSockets

EXPERIENCE:

- Led backend development team of 5 engineers
- Architected distributed system handling 1M+ requests/day

2. **Upload and analyze:**

curl -X POST http://localhost:8000/api/v1/resume/analyze
-F "file=@test_resume.pdf"
-F "user_id=test-user-123"
-F "github_analysis_id=<your-github-analysis-id>"

3. **Expected Response:**

{
"status": "success",
"credibility_score": 45.0,
"skills_verification": [
{
"skill": "Python",
"evidence_level": "STRONG",
"classification": "EARNED",
"reason": "Found in 3 repositories with significant usage"
},
{
"skill": "Kubernetes",
"evidence_level": "NONE",
"classification": "REMOVE",
"reason": "No evidence in GitHub repositories"
}
],
"removal_recommendations": ["Kubernetes", "Redis"]
}

4. **Validation Checklist:**
    - [ ] Credibility score is realistic (should be harsh)
    - [ ] Some skills marked as "REMOVE" (system is brutally honest)
    - [ ] Claims matched against actual GitHub code

- [ ] Recommendations are actionable

🎯 Phase 3: Interview Interrogator (Mode 1 - Evaluator)

Duration: Days 8-11 (4 days)

📚 Learning Objectives

- Generate role-specific interview questions from GitHub projects
- Implement escalating follow-up logic
- Detect vague answers and drill down
- Score answers for technical depth and clarity
- Identify failure points and confidence gaps

🔧 Additional Dependencies

```
pip install websockets12.0
pip install redis5.0.1 # For session state
pip install sentence-transformers==2.2.2 # For semantic similarity
```

📝 Step-by-Step Guide

Step 3.1: Question Generator Service

File: `app/services/question_generator.py`

```
import google.generativeai as genai
from typing import Dict, List
from app.core.config import settings
import json

class QuestionGenerator:
def __init__(self):
genai.configure(api_key=settings.GEMINI_API_KEY)
self.model = genai.GenerativeModel('gemini-pro')
```

```
    async def generate_questions(
        self,
        github_analysis: Dict,
        resume_analysis: Dict,
        target_role: str,
        difficulty: str = "medium"
    ) -> List[Dict]:
        """
        Generate interview questions based on GitHub and resume
        """
        questions = []
```

```
        # Base questions from projects
        project_questions = await self._generate_project_questions(github_analysis, t
        questions.extend(project_questions)

        # Adversarial questions from weaknesses
        weakness_questions = await self._generate_weakness_questions(
            github_analysis["weaknesses"],
            target_role
        )
        questions.extend(weakness_questions)

        # Claim verification questions
        claim_questions = await self._generate_claim_questions(
            resume_analysis,
            github_analysis
        )
        questions.extend(claim_questions)

        # Sort by difficulty and priority
        questions = self._prioritize_questions(questions, difficulty)

        return questions[:15]  # Max 15 questions per session

    async def _generate_project_questions(self, github_analysis: Dict, role: str) -> Li
        """Generate questions from user's actual projects"""
        prompt = f"""
```

You are a hostile {role} interviewer.

CANDIDATE'S PROJECT:
Architecture: {github_analysis.get('architecture_summary')}
Tech Stack: {github_analysis.get('tech_stack')}

TASK:
Generate 5 brutal interview questions that:

1. Expose shallow understanding
2. Ask "why X instead of Y?"
3. Ask about failure modes
4. Ask about tradeoffs
5. Ask about scale/edge cases

Questions should be specific to their actual code, not generic.

Output JSON array:
[
{{
"question": "...",
"type": "technical_depth",
"difficulty": "hard",
"attack_vector": "architecture|tradeoffs|scale|error_handling"
}}
]
"""

```
    try:
        response = self.model.generate_content(prompt)
        # TODO: Parse JSON array from response
        return [
            {
                "question": "Why did you choose FastAPI instead of Flask for this proj
                "type": "technical_depth",
                "difficulty": "medium",
                "attack_vector": "tradeoffs",
                "follow_up_triggers": ["performance", "async", "documentation"]
            }
        ]
    except Exception as e:
        print(f"Question generation failed: {e}")
        return []

async def _generate_weakness_questions(self, weaknesses: List[str], role: str) ->
    """Generate questions targeting detected weaknesses"""
    questions = []

    weakness_templates = {
        "No tests": "Your project has no automated tests. How would you test the au
        "No error handling": "I see minimal error handling in your API routes. Wh
        "Security issues": "Your API doesn't implement rate limiting. How would yo
        "No documentation": "The codebase lacks API documentation. How would
    }

    for weakness in weaknesses:
```

```
        for pattern, template_question in weakness_templates.items():
            if pattern.lower() in weakness.lower():
                questions.append({
                    "question": template_question,
                    "type": "weakness_exploitation",
                    "difficulty": "hard",
                    "attack_vector": "gaps",
                    "requires_rebuild": True
                })

    return questions

async def _generate_claim_questions(self, resume_analysis: Dict, github_analys
    """Generate questions targeting weak/misleading claims"""
    questions = []

    # Find weak or misleading claims
    weak_claims = [
        claim for claim in resume_analysis.get("project_claims", [])
        if claim.get("classification") in ["WEAK", "MISLEADING"]
    ]

    for claim_data in weak_claims:
        claim = claim_data["claim"]
        questions.append({
            "question": f"Your resume says: '{claim}'. Walk me through the impleme
            "type": "claim_verification",
            "difficulty": "hard",
            "attack_vector": "overclaiming",
            "expected_failure": True
        })

    return questions

def _prioritize_questions(self, questions: List[Dict], difficulty: str) -> List[Dict]:
    """Sort questions by priority and difficulty"""
    priority_order = {
        "weakness_exploitation": 1,
```

```
        "claim_verification": 2,
        "technical_depth": 3
    }

    difficulty_scores = {
        "easy": 1,
        "medium": 2,
        "hard": 3
    }

    target_difficulty = difficulty_scores.get(difficulty, 2)

    # Sort by priority and difficulty match
    sorted_questions = sorted(
        questions,
        key=lambda q: (
            priority_order.get(q["type"], 99),
            abs(difficulty_scores.get(q.get("difficulty", "medium"), 2) - target_difficulty
        )
    )

    return sorted_questions
```

Step 3.2: Answer Evaluator Service

File: `app/services/answer_evaluator.py`

```
import google.generativeai as genai
from typing import Dict, List
import re
from sentence_transformers import SentenceTransformer
import numpy as np
from app.core.config import settings

class AnswerEvaluator:
def __init__(self):
genai.configure(api_key=settings.GEMINI_API_KEY)
self.model = genai.GenerativeModel('gemini-pro')
self.embedding_model = SentenceTransformer('all-MiniLM-L6-v2')
```

```
async def evaluate_answer(
    self,
```

```
    question: str,
    answer: str,
    context: Dict
) -> Dict:
    """
    Evaluate interview answer for technical depth and clarity
    """
    # Quick heuristic checks
    surface_signals = self._check_surface_signals(answer)

    # Semantic depth analysis
    depth_score = await self._analyze_depth(question, answer, context)

    # Confidence analysis
    confidence_analysis = self._analyze_confidence(answer)

    # Generate follow-up
    needs_followup = depth_score < 70 or surface_signals["filler_word_count"] >
    followup_question = None

    if needs_followup:
        followup_question = await self._generate_followup(question, answer, surf

    return {
        "depth_score": depth_score,
        "confidence_score": confidence_analysis["score"],
        "filler_words": surface_signals["filler_words"],
        "filler_word_count": surface_signals["filler_word_count"],
        "vague_phrases": surface_signals["vague_phrases"],
        "answer_length": len(answer.split()),
        "needs_followup": needs_followup,
        "followup_question": followup_question,
        "classification": self._classify_answer(depth_score, confidence_analysis["sc
    }

def _check_surface_signals(self, answer: str) -> Dict:
    """Detect filler words and vague phrases"""
    filler_words = ["um", "uh", "like", "basically", "kind of", "sort of", "i think", "m
```

```
        vague_phrases = ["and stuff", "things like that", "you know", "et cetera"]

        answer_lower = answer.lower()

        detected_fillers = [word for word in filler_words if word in answer_lower]
        detected_vague = [phrase for phrase in vague_phrases if phrase in answer_lo

        return {
            "filler_words": detected_fillers,
            "filler_word_count": sum(answer_lower.count(word) for word in filler_wo
            "vague_phrases": detected_vague
        }

    async def _analyze_depth(self, question: str, answer: str, context: Dict) -> float:
        """
        Analyze technical depth of answer using AI
        Returns score 0-100
        """
        prompt = f"""
```

You are evaluating a technical interview answer.

QUESTION: {question}

CANDIDATE'S ANSWER: {answer}

CONTEXT (from their GitHub):

- Tech Stack: {context.get('tech_stack', {})}
- Project Type: {context.get('project_type', 'Unknown')}

TASK:
Score this answer 0-100 based on:

1. Technical accuracy (40 points)
2. Depth of understanding (30 points)
3. Specific examples (20 points)
4. Tradeoff awareness (10 points)

Be HARSH. Most candidates score 40-60.

Output JSON:
{{
"score": 0-100,
"reasoning": "...",
"missing_elements": ["..."]

}}
"""

```
        try:
            response = self.model.generate_content(prompt)
            # TODO: Parse JSON response
            return 55.0  # Placeholder
        except:
            return 50.0

    def _analyze_confidence(self, answer: str) -> Dict:
        """Analyze confidence level from linguistic patterns"""
        # Confidence detractors
        uncertainty_phrases = [
            "i'm not sure", "i think", "probably", "maybe",
            "i guess", "not entirely sure", "might be"
        ]

        answer_lower = answer.lower()
        uncertainty_count = sum(1 for phrase in uncertainty_phrases if phrase in an

        # Length check (too short = uncertain)
        word_count = len(answer.split())
        length_score = min(word_count / 50, 1.0) * 50  # Max 50 points for length

        # Confidence score (0-100)
        confidence_score = max(0, 100 - (uncertainty_count * 15) - (50 - length_score))

        return {
            "score": confidence_score,
            "uncertainty_count": uncertainty_count,
            "word_count": word_count
        }

    async def _generate_followup(self, question: str, answer: str, signals: Dict) -> str
        """Generate drilling follow-up question"""
        prompt = f"""
```

You are a hostile interviewer. The candidate gave a weak answer.

ORIGINAL QUESTION: {question}
CANDIDATE'S ANSWER: {answer}
DETECTED ISSUES: {signals}

Generate a follow-up question that:

1. Drills down on the vague parts
2. Asks for specific examples
3. Tests depth of knowledge

Be aggressive. Expose the gap.

Output just the question text.
"""

```
        try:
            response = self.model.generate_content(prompt)
            return response.text.strip()
        except:
            return "Can you be more specific? Give me a concrete example."

    def _classify_answer(self, depth_score: float, confidence_score: float) -> str:
        """Classify answer quality"""
        avg_score = (depth_score + confidence_score) / 2

        if avg_score >= 80:
            return "STRONG"
        elif avg_score >= 60:
            return "ACCEPTABLE"
        elif avg_score >= 40:
            return "WEAK"
        else:
            return "FAILED"
```

Step 3.3: Interview Session Manager

File: `app/services/interview_session.py`

```
from typing import Dict, List, Optional
import json
import redis
import uuid
from datetime import datetime, timedelta
```

class InterviewSession:
def **init**(self, redis_client: redis.Redis):
self.redis = redis_client

```
def create_session(
    self,
    user_id: str,
    questions: List[Dict],
    target_role: str
) -> str:
    """Create new interview session"""
    session_id = str(uuid.uuid4())

    session_data = {
        "session_id": session_id,
        "user_id": user_id,
        "target_role": target_role,
        "questions": questions,
        "current_question_index": 0,
        "answers": [],
        "started_at": datetime.utcnow().isoformat(),
        "status": "in_progress"
    }

    # Store in Redis with 1 hour expiry
    self.redis.setex(
        f"interview_session:{session_id}",
        timedelta(hours=1),
        json.dumps(session_data)
    )

    return session_id

def get_session(self, session_id: str) -> Optional[Dict]:
    """Retrieve session data"""
    data = self.redis.get(f"interview_session:{session_id}")
    if data:
        return json.loads(data)
```

```
        return None

    def save_answer(
        self,
        session_id: str,
        question_index: int,
        answer: str,
        evaluation: Dict
    ):
        """Save answer and evaluation to session"""
        session = self.get_session(session_id)
        if not session:
            raise ValueError("Session not found")

        session["answers"].append({
            "question_index": question_index,
            "answer": answer,
            "evaluation": evaluation,
            "answered_at": datetime.utcnow().isoformat()
        })

        session["current_question_index"] = question_index + 1

        # Check if session complete
        if session["current_question_index"] >= len(session["questions"]):
            session["status"] = "completed"
            session["completed_at"] = datetime.utcnow().isoformat()

        # Update Redis
        self.redis.setex(
            f"interview_session:{session_id}",
            timedelta(hours=1),
            json.dumps(session)
        )

    def get_current_question(self, session_id: str) -> Optional[Dict]:
        """Get next question in session"""
        session = self.get_session(session_id)
```

```
        if not session or session["status"] != "in_progress":
            return None

        idx = session["current_question_index"]
        if idx < len(session["questions"]):
            return session["questions"][idx]

        return None

    def get_session_summary(self, session_id: str) -> Dict:
        """Generate session performance summary"""
        session = self.get_session(session_id)
        if not session:
            raise ValueError("Session not found")

        answers = session["answers"]

        # Calculate metrics
        total_questions = len(answers)
        avg_depth_score = sum(a["evaluation"]["depth_score"] for a in answers) / tota
        avg_confidence_score = sum(a["evaluation"]["confidence_score"] for a in ansv

        failed_questions = [
            a for a in answers
            if a["evaluation"]["classification"] in ["WEAK", "FAILED"]
        ]

        strong_questions = [
            a for a in answers
            if a["evaluation"]["classification"] == "STRONG"
        ]

        return {
            "session_id": session_id,
            "total_questions": total_questions,
            "avg_depth_score": avg_depth_score,
            "avg_confidence_score": avg_confidence_score,
            "strong_answers": len(strong_questions),
```

```
        "weak_answers": len([a for a in answers if a["evaluation"]["classification"] =
        "failed_answers": len([a for a in answers if a["evaluation"]["classification"]
        "failure_points": [
            {
                "question": session["questions"][a["question_index"]]["question"],
                "answer": a["answer"],
                "issues": a["evaluation"].get("missing_elements", [])
            }
            for a in failed_questions
        ]
    }
```

Step 3.4: WebSocket Interview API

File: app/api/interview.py

```
from fastapi import APIRouter, WebSocket, WebSocketDisconnect, Depends
from sqlalchemy.orm import Session
from app.services.question_generator import QuestionGenerator
from app.services.answer_evaluator import AnswerEvaluator
from app.services.interview_session import InterviewSession
from app.db.session import get_db
import json

router = APIRouter(prefix="/interview", tags=["Interview System"])

@router.websocket("/ws/{session_id}")
async def interview_websocket(
websocket: WebSocket,
session_id: str,
db: Session = Depends(get_db)
):
"""
Real-time interview session via WebSocket
"""
await websocket.accept()
```

```
# Initialize services
session_manager = InterviewSession(redis_client)  # TODO: Inject Redis
evaluator = AnswerEvaluator()

try:
    # Get session
    session = session_manager.get_session(session_id)
```

```
    if not session:
        await websocket.send_json({"error": "Session not found"})
        await websocket.close()
        return

    # Send first question
    current_question = session_manager.get_current_question(session_id)
    if current_question:
        await websocket.send_json({
            "type": "question",
            "data": current_question
        })

    # Listen for answers
    while True:
        data = await websocket.receive_json()

        if data["type"] == "answer":
            answer = data["answer"]
            question_index = data["question_index"]

            # Evaluate answer
            question_data = session["questions"][question_index]
            evaluation = await evaluator.evaluate_answer(
                question=question_data["question"],
                answer=answer,
                context=session.get("context", {})
            )

            # Save to session
            session_manager.save_answer(session_id, question_index, answer, evalu

            # Send evaluation
            await websocket.send_json({
                "type": "evaluation",
                "data": evaluation
            })
```

```
            # Send follow-up or next question
            if evaluation["needs_followup"]:
                await websocket.send_json({
                    "type": "followup",
                    "data": {
                        "question": evaluation["followup_question"]
                    }
                })
            else:
                next_question = session_manager.get_current_question(session_id)
                if next_question:
                    await websocket.send_json({
                        "type": "question",
                        "data": next_question
                    })
                else:
                    # Session complete
                    summary = session_manager.get_session_summary(session_id)
                    await websocket.send_json({
                        "type": "session_complete",
                        "data": summary
                    })
                    break

        elif data["type"] == "skip":
            # Allow skipping questions
            next_question = session_manager.get_current_question(session_id)
            if next_question:
                await websocket.send_json({
                    "type": "question",
                    "data": next_question
                })

except WebSocketDisconnect:
    print(f"Client disconnected from session {session_id}")
except Exception as e:
```

```
    await websocket.send_json({"error": str(e)})
    await websocket.close()
```

```
@router.post("/start")
async def start_interview_session(
user_id: str,
github_analysis_id: str,
resume_analysis_id: str,
target_role: str,
difficulty: str = "medium",
db: Session = Depends(get_db)
):
"""
Start new interview session
"""
# Get analysis data
github_analysis = db.query(GitHubAnalysis).filter(
GitHubAnalysis.id == github_analysis_id
).first()
```

```
resume_analysis = db.query(ResumeAnalysis).filter(
    ResumeAnalysis.id == resume_analysis_id
).first()

if not github_analysis or not resume_analysis:
    raise HTTPException(status_code=404, detail="Analysis data not found")

# Generate questions
generator = QuestionGenerator()
questions = await generator.generate_questions(
    github_analysis={
        "architecture_summary": github_analysis.architecture_summary,
        "tech_stack": github_analysis.tech_stack,
        "weaknesses": github_analysis.weaknesses,
        "project_type": github_analysis.project_type
    },
    resume_analysis={
        "project_claims": resume_analysis.project_claims
    },
    target_role=target_role,
    difficulty=difficulty
```

```
    )

    # Create session
    session_manager = InterviewSession(redis_client)
    session_id = session_manager.create_session(
        user_id=user_id,
        questions=questions,
        target_role=target_role
    )

    return {
        "session_id": session_id,
        "total_questions": len(questions),
        "websocket_url": f"ws://localhost:8000/api/v1/interview/ws/{session_id}"
    }
```

 Manual Test Guide (Phase 3)

Test 1: Start Interview Session

curl -X POST http://localhost:8000/api/v1/interview/start
-H "Content-Type: application/json"
-d '{
"user_id": "test-user-123",
"github_analysis_id": "<your-analysis-id>",
"resume_analysis_id": "<your-resume-id>",
"target_role": "Backend Engineer",
"difficulty": "medium"
}'

Expected Response:

{
"session_id": "abc-123-def-456",
"total_questions": 12,
"websocket_url": "ws://localhost:8000/api/v1/interview/ws/abc-123-def-456"
}

Test 2: Connect to WebSocket (using browser console or wscat)

const ws = new WebSocket("ws://localhost:8000/api/v1/interview/ws/abc-123-def-456");

ws.onmessage = (event) => {
const message = JSON.parse(event.data);
console.log("Received:", message);

```
if (message.type === "question") {
// Send answer
setTimeout(() => {
ws.send(JSON.stringify({
type: "answer",
question_index: 0,
answer: "I chose FastAPI because of its async support and automatic API documentation with Swagger. It's faster than Flask for I/O-bound operations."
}));
}, 2000);
}

if (message.type === "evaluation") {
console.log("Score:", message.data.depth_score);
console.log("Needs follow-up:", message.data.needs_followup);
}
};

ws.onopen = () => console.log("Connected to interview session");
```

Test 3: Weak Answer (should trigger follow-up)

```
ws.send(JSON.stringify({
type: "answer",
question_index: 0,
answer: "Um, I think FastAPI is good. It's, like, faster and stuff."
}));
```

Expected Evaluation:

```
{
"type": "evaluation",
"data": {
"depth_score": 25,
"confidence_score": 35,
"filler_word_count": 3,
"filler_words": ["um", "i think", "like"],
"needs_followup": true,
"followup_question": "You mentioned FastAPI is faster. Faster than what? Can you quantify the performance difference?",
"classification": "FAILED"
}
}
```

Validation Checklist Phase 3:

- [ ] Questions are specific to user's GitHub projects
- [ ] Weak answers trigger drilling follow-ups
- [ ] Filler word detection works accurately
- [ ] Depth scores are harsh (most answers score 40-70)
- [ ] Session completes with performance summary
- [ ] Failure points are clearly identified

Phase 4: Gap Repair Engine (Mode 2 - Fixer)

Duration: Days 12-14 (3 days)

Learning Objectives

- Classify failure gaps: Conceptual / Experiential / Structural
- Decide repair path: Defend / Learn / Rebuild
- Generate defensible framing for weak areas
- Create targeted learning drills (2-4 hours max)
- Prescribe small, brutal projects that force competence

Step-by-Step Guide

Step 4.1: Gap Classifier Service

File: app/services/gap_classifier.py

from typing import Dict, List
import google.generativeai as genai
from app.core.config import settings

class GapClassifier:
"""
Classify knowledge gaps from interview failures
"""

```
def __init__(self):
    genai.configure(api_key=settings.GEMINI_API_KEY)
    self.model = genai.GenerativeModel('gemini-pro')

async def classify_gaps(self, failure_points: List[Dict]) -> List[Dict]:
    """
    Classify each failure point into gap type
    """
    classified_gaps = []

    for failure in failure_points:
        gap_type = await self._determine_gap_type(
            question=failure["question"],
            answer=failure["answer"],
            issues=failure["issues"]
        )

        classified_gaps.append({
            "question": failure["question"],
```

```
            "answer": failure["answer"],
            "gap_type": gap_type["type"],
            "severity": gap_type["severity"],
            "reasoning": gap_type["reasoning"]
        })

    return classified_gaps

async def _determine_gap_type(self, question: str, answer: str, issues: List[str]) -
    """
    Determine if gap is Conceptual, Experiential, or Structural
    """
    prompt = f"""
```

You are analyzing why a candidate failed an interview question.

QUESTION: {question}
CANDIDATE'S ANSWER: {answer}
DETECTED ISSUES: {issues}

TASK:
Classify the knowledge gap as ONE of:

1. CONCEPTUAL - Lacks theoretical understanding (can be learned in 2-4 hours)
   Example: Doesn't understand Redis eviction policies
2. EXPERIENTIAL - Understands theory but never implemented (needs practice)
   Example: Knows what async/await is but never debugged race conditions
3. STRUCTURAL - Never built this type of system (needs full project)
   Example: Claims microservices experience but has no distributed system in GitHub

Also rate severity: LOW, MEDIUM, HIGH

Output JSON:
{{
"type": "CONCEPTUAL|EXPERIENTIAL|STRUCTURAL",
"severity": "LOW|MEDIUM|HIGH",
"reasoning": "..."
}}
"""

```
    try:
        response = self.model.generate_content(prompt)
        # TODO: Parse JSON response
        return {
            "type": "EXPERIENTIAL",
```

```
            "severity": "MEDIUM",
            "reasoning": "Candidate knows the concept but lacks hands-on experien
        }
    except:
        return {
            "type": "UNKNOWN",
            "severity": "MEDIUM",
            "reasoning": "Classification failed"
        }
```

Step 4.2: Repair Strategy Generator

File: app/services/repair_generator.py

```
from typing import Dict, List, Optional
import google.generativeai as genai
from app.core.config import settings

class RepairGenerator:
"""
Generate repair strategies: Defensible Framing, Learning Drills, or Project Prescriptions
"""

def __init__(self):
    genai.configure(api_key=settings.GEMINI_API_KEY)
    self.model = genai.GenerativeModel('gemini-pro')

async def generate_repair_strategy(self, gap: Dict) -> Dict:
    """
    Decide: Defend, Learn, or Rebuild
    Generate appropriate repair strategy
    """
    gap_type = gap["gap_type"]
    severity = gap["severity"]

    if gap_type == "CONCEPTUAL":
        # Learning drill
        strategy = await self._generate_learning_drill(gap)
    elif gap_type == "EXPERIENTIAL":
        # Might be defensible OR need practice
        if severity == "LOW":
```

```
            strategy = await self._generate_defensible_framing(gap)
        else:
            strategy = await self._generate_practice_exercise(gap)
    else:  # STRUCTURAL
        # Must rebuild
        strategy = await self._generate_project_prescription(gap)

    return strategy

async def _generate_defensible_framing(self, gap: Dict) -> Dict:
    """
    Teach how to honestly frame limited experience
    """
    prompt = f"""
```

You are a career advisor teaching honest self-representation.

CANDIDATE'S WEAK AREA:
Question: {gap['question']}
Their Answer: {gap['answer']}
Gap Type: {gap['gap_type']}

TASK:
The candidate has operational familiarity but not deep expertise.
Generate a defensible framing statement that:

1. Acknowledges scope of experience
2. Avoids overclaiming
3. Shows honesty and self-awareness
4. Prepares for likely follow-up questions

Output JSON:
{{
"repair_type": "DEFENSIBLE_FRAMING",
"honest_statement": "...",
"avoid_saying": ["...", "..."],
"expected_followups": ["...", "..."],
"when_to_say_no": "..."
}}
"""

```
    try:
        response = self.model.generate_content(prompt)
        # TODO: Parse JSON
```

```
            return {
                "repair_type": "DEFENSIBLE_FRAMING",
                "honest_statement": "I've used Redis for basic caching in development, b
                "avoid_saying": [
                    "I'm a Redis expert",
                    "I've architected Redis clusters"
                ],
                "expected_followups": [
                    "What eviction policies are you familiar with?",
                    "How would you monitor Redis in production?"
                ],
                "when_to_say_no": "If asked about Redis clustering or replication, admit
            }
        except:
            return {"repair_type": "DEFENSIBLE_FRAMING", "error": "Generation failed

    async def _generate_learning_drill(self, gap: Dict) -> Dict:
        """
        Generate 2-4 hour focused learning task
        """
        prompt = f"""
```

You are designing a surgical learning intervention.

CANDIDATE'S KNOWLEDGE GAP:
Question Failed: {gap['question']}
Gap Type: CONCEPTUAL
Severity: {gap['severity']}

TASK:
Design a 2-4 hour learning drill that:

1. Targets ONLY this specific gap
2. Is hands-on, not passive reading
3. Forces understanding through practice
4. Includes a forced re-explanation afterward

Output JSON:
{{
"repair_type": "LEARNING_DRILL",
"time_required": "2-4 hours",
"learning_objectives": ["...", "..."],
"steps": [
{{"action": "...", "duration": "30 min"}},

{{"action": "...", "duration": "1 hour"}}
],
"verification_task": "..."
}}
"""

```
        try:
            response = self.model.generate_content(prompt)
            return {
                "repair_type": "LEARNING_DRILL",
                "time_required": "3 hours",
                "learning_objectives": [
                    "Understand LRU vs LFU eviction policies",
                    "Implement custom eviction logic"
                ],
                "steps": [
                    {"action": "Read Redis eviction policy docs (official)", "duration": "30 mi
                    {"action": "Implement LRU cache in Python from scratch", "duration": "
                    {"action": "Compare with Redis implementation", "duration": "45 min"}
                    {"action": "Write blog post explaining tradeoffs", "duration": "45 min"}
                ],
                "verification_task": "Re-answer the original interview question out loud
            }
        except:
            return {"repair_type": "LEARNING_DRILL", "error": "Generation failed"}

    async def _generate_practice_exercise(self, gap: Dict) -> Dict:
        """
        Generate hands-on practice exercise (4-8 hours)
        """
        prompt = f"""
```

You are designing a focused practice exercise.

CANDIDATE'S GAP:
Question: {gap['question']}
Gap Type: EXPERIENTIAL (knows theory, lacks practice)

TASK:
Design a small exercise that:

1. Forces them to implement the concept

2. Includes a failure condition they must debug
3. Takes 4-8 hours max
4. Generates interview stories

Output JSON:
{{
"repair_type": "PRACTICE_EXERCISE",
"time_required": "4-8 hours",
"exercise_title": "...",
"objective": "...",
"requirements": ["...", "..."],
"forced_failure": "...",
"success_criteria": "...",
"interview_story": "..."
}}
"""

```
        try:
            response = self.model.generate_content(prompt)
            return {
                "repair_type": "PRACTICE_EXERCISE",
                "time_required": "6 hours",
                "exercise_title": "Build Rate-Limited API with Concurrency Bug",
                "objective": "Experience race conditions in rate limiting",
                "requirements": [
                    "Create FastAPI endpoint with rate limiting (10 req/min)",
                    "Use in-memory counter (intentionally flawed)",
                    "Write concurrent test script that breaks it"
                ],
                "forced_failure": "Under concurrent load, rate limiting fails. Some reque
                "success_criteria": "Fix race condition using proper locking or atomic op
                "interview_story": "Now you can say: 'I built rate limiting and discovered
            }
        except:
            return {"repair_type": "PRACTICE_EXERCISE", "error": "Generation failed"}

    async def _generate_project_prescription(self, gap: Dict) -> Dict:
        """
        Generate complete project prescription for structural gaps
        """
        prompt = f"""
```

You are prescribing a portfolio project to fill a major gap.

CANDIDATE'S STRUCTURAL GAP:
Failed Question: {gap['question']}
Reasoning: {gap['reasoning']}

TASK:
Prescribe a SMALL but PAINFUL project that:

1. Targets this specific gap
2. Forces failure and recovery
3. Creates defensible interview stories
4. Takes 1-2 weeks max (evenings/weekends)

Include:

- Clear scope (what to build)
- Constraint that forces the skill (e.g., "must handle 1000 concurrent connections")
- Expected failure mode (what will break first)
- Success criteria (how you know it works)
- Interview stories this enables

Output JSON:
{{
"repair_type": "PROJECT_PRESCRIPTION",
"time_required": "1-2 weeks",
"project_title": "...",
"gap_targeted": "...",
"scope": {{"must_include": ["...", "..."], "must_exclude": ["..."]}},
"constraint": "...",
"expected_failure": "...",
"success_criteria": "...",
"interview_stories": ["...", "..."],
"technologies": ["...", "..."]
}}

```
    try:
        response = self.model.generate_content(prompt)
        return {
            "repair_type": "PROJECT_PRESCRIPTION",
            "time_required": "10-14 days",
            "project_title": "Build Distributed Task Queue",
            "gap_targeted": "Microservices architecture and async processing",
            "scope": {
                "must_include": [
                    "Producer API (submit tasks)",
                    "Worker service (process tasks)",
```

```
                "Redis as message broker",
                "Task retry logic",
                "Status tracking endpoint"
            ],
            "must_exclude": [
                "Complex UI (CLI is fine)",
                "User authentication",
                "Kubernetes deployment"
            ]
        },
        "constraint": "Must handle 100 concurrent task submissions without dat
        "expected_failure": "Workers crash under load. Tasks lost. No retry mecl
        "success_criteria": [
            "1000 tasks processed successfully",
            "Worker crashes don't lose tasks",
            "Failed tasks retry automatically"
        ],
        "interview_stories": [
            "I built a task queue and discovered workers crashing lost tasks. Impl
            "Under load testing, found race conditions in task assignment. Fixed v
        ],
        "technologies": ["FastAPI", "Redis", "Docker", "pytest"]
    }
except:
    return {"repair_type": "PROJECT_PRESCRIPTION", "error": "Generation faile
```

Step 4.3: Repair Strategy API

File: `app/api/repair.py`

```
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from app.services.gap_classifier import GapClassifier
from app.services.repair_generator import RepairGenerator
from app.db.session import get_db

router = APIRouter(prefix="/repair", tags=["Gap Repair"])

@router.post("/analyze-gaps")
async def analyze_gaps(
session_id: str,
db: Session = Depends(get_db)
):
```

"""
Analyze interview failures and classify gaps
"""
# Get interview session
session_manager = InterviewSession(redis_client)
summary = session_manager.get_session_summary(session_id)

```
if not summary:
    raise HTTPException(status_code=404, detail="Session not found")

# Classify gaps
classifier = GapClassifier()
classified_gaps = await classifier.classify_gaps(summary["failure_points"])

return {
    "session_id": session_id,
    "total_gaps": len(classified_gaps),
    "gap_breakdown": {
        "conceptual": sum(1 for g in classified_gaps if g["gap_type"] == "CONCEPTUA
        "experiential": sum(1 for g in classified_gaps if g["gap_type"] == "EXPERIEN
        "structural": sum(1 for g in classified_gaps if g["gap_type"] == "STRUCTURA
    },
    "gaps": classified_gaps
}
```

@router.post("/generate-repair-plan")
async def generate_repair_plan(
session_id: str,
db: Session = Depends(get_db)
):
"""
Generate complete repair plan for all gaps
"""
# Get classified gaps
classifier = GapClassifier()
session_manager = InterviewSession(redis_client)
summary = session_manager.get_session_summary(session_id)
classified_gaps = await classifier.classify_gaps(summary["failure_points"])

```
# Generate repair strategies
generator = RepairGenerator()
repair_plan = []
```

```
    for gap in classified_gaps:
        strategy = await generator.generate_repair_strategy(gap)
        repair_plan.append({
            "gap": gap,
            "strategy": strategy
        })

    return {
        "session_id": session_id,
        "repair_plan": repair_plan,
        "estimated_total_time": _calculate_total_time(repair_plan),
        "priority_order": _prioritize_repairs(repair_plan)
    }
```

```
def _calculate_total_time(repair_plan: List[Dict]) -> str:
    """Calculate total time required for all repairs"""
    # Implementation depends on parsing time_required strings
    return "2-4 weeks"

def _prioritize_repairs(repair_plan: List[Dict]) -> List[str]:
    """Prioritize repairs by severity and type"""
    # Structural gaps first, then experiential, then conceptual
    priority = []
```

```
    for item in sorted(repair_plan, key=lambda x: (
        0 if x["gap"]["gap_type"] == "STRUCTURAL" else 1 if x["gap"]["gap_type"] == "EX
        0 if x["gap"]["severity"] == "HIGH" else 1 if x["gap"]["severity"] == "MEDIUM" e
    )):
        priority.append(item["gap"]["question"])

    return priority
```

Manual Test Guide (Phase 4)

Test 1: Analyze Gaps

curl -X POST "http://localhost:8000/api/v1/repair/analyze-gaps?session_id=abc-123"

Expected Response:

```
{
"session_id": "abc-123",
```

"total_gaps": 5,
"gap_breakdown": {
"conceptual": 2,
"experiential": 2,
"structural": 1
},
"gaps": [
{
"question": "How would you implement distributed caching?",
"gap_type": "STRUCTURAL",
"severity": "HIGH",
"reasoning": "Candidate has never built distributed system"
}
]
}

Test 2: Generate Repair Plan

curl -X POST "http://localhost:8000/api/v1/repair/generate-repair-plan?session_id=abc-123"

Expected Response:

{
"session_id": "abc-123",
"repair_plan": [
{
"gap": {
"question": "Explain Redis eviction policies",
"gap_type": "CONCEPTUAL",
"severity": "MEDIUM"
},
"strategy": {
"repair_type": "LEARNING_DRILL",
"time_required": "3 hours",
"steps": [...]
}
},
{
"gap": {
"question": "How would you handle race conditions?",
"gap_type": "EXPERIENTIAL",
"severity": "HIGH"
},
"strategy": {
"repair_type": "PRACTICE_EXERCISE",
"exercise_title": "Build Rate-Limited API with Concurrency Bug",
"time_required": "6 hours"
}
},
{
"gap": {
"question": "Design a microservices architecture",

"gap_type": "STRUCTURAL",
"severity": "HIGH"
},
"strategy": {
"repair_type": "PROJECT_PRESCRIPTION",
"project_title": "Build Distributed Task Queue",
"time_required": "10-14 days"
}
}
],
"estimated_total_time": "2-3 weeks",
"priority_order": [
"Design a microservices architecture",
"How would you handle race conditions?",
"Explain Redis eviction policies"
]
}

Validation Checklist Phase 4:

- [ ] Gaps correctly classified (Conceptual vs Experiential vs Structural)
- [ ] Repair strategies match gap types
- [ ] Project prescriptions are scoped (1-2 weeks max)
- [ ] Learning drills are time-bound (2-4 hours)
- [ ] Defensible framing avoids teaching lies
- [ ] System sometimes says "Do NOT claim this skill"

(Continuing with remaining phases in follow-up...)

PART 3: PRODUCTION DEPLOYMENT GUIDE

Phase 9: Production Deployment & Handover

Duration: Days 24-25 (2 days)

 Deployment Architecture

Figure 1: FAB Production Architecture

Component Deployment:

1. **Backend API:** Vercel Serverless Functions (Free tier)
2. **Frontend:** Vercel Static Hosting (Free tier)
3. **Database:** PlanetScale MySQL (Free tier, 5GB)
4. **Redis:** Upstash Redis (Free tier, 10K requests/day)
5. **GPU Tasks:** Google Colab (Free tier, manual trigger)
6. **File Storage:** Vercel Blob (Free tier, 1GB)

Deployment Checklist

- [ ] Backend deployed to Vercel
- [ ] Frontend deployed to Vercel
- [ ] Database migrated to PlanetScale
- [ ] Redis connected to Upstash
- [ ] Environment variables configured
- [ ] CORS configured for production domain
- [ ] Rate limiting enabled
- [ ] Error tracking setup (Sentry free tier)
- [ ] Analytics setup (PostHog free tier)
- [ ] Documentation updated with production URLs

PART 4: TESTING & VALIDATION

Integration Testing Requirements

Phase 1 Tests:

- [ ] GitHub API rate limiting respected
- [ ] Repository analysis completes in <15 seconds
- [ ] Originality score calculation accurate
- [ ] Boilerplate detection functional

Phase 2 Tests:

- [ ] Resume parsing works for PDF and DOCX
- [ ] Claim verification matches GitHub evidence
- [ ] Credibility score is harsh but fair
- [ ] Removal recommendations are actionable

Phase 3 Tests:

- [ ] Interview sessions persist in Redis
- [ ] WebSocket connections stable for 30+ minutes
- [ ] Answer evaluation detects filler words
- [ ] Follow-up questions drill deeper
- [ ] Session summary calculates correctly

Phase 4 Tests:

- [ ] Gap classification is accurate
- [ ] Repair strategies match gap types
- [ ] Project prescriptions are scoped appropriately
- [ ] Learning drills are time-bound

PART 5: SUCCESS METRICS

What Success Looks Like

User Feedback Indicators:

- Users say "this is uncomfortably accurate"
- Users report reduced overclaiming in interviews
- Users complete prescribed projects
- Users pass interviews after repair cycle

System Health Indicators:

- 95%+ uptime
- <2 second API response time
- <10% error rate
- Positive GitHub analysis accuracy

Evolution Indicators:

- Interview outcome feedback collected
- Failure patterns detected and updated
- RAG memory growing with real data
- Question quality improving over time

References

[1] OpenAI. (2024). GPT-4 Technical Report. https://openai.com/research/gpt-4

[2] Google AI. (2024). Gemini API Documentation. https://ai.google.dev/

[3] Hugging Face. (2024). Sentence Transformers Library. https://www.sbert.net/

[4] FastAPI. (2024). FastAPI Framework Documentation. https://fastapi.tiangolo.com/

[5] Redis Labs. (2024). Redis Eviction Policies. https://redis.io/docs/manual/eviction/

[6] GitHub. (2024). GitHub REST API v3 Documentation. https://docs.github.com/en/rest

[7] LangChain. (2024). LangChain RAG Implementation Guide. https://python.langchain.com/docs/use_cases/question_answering/

[8] Vercel. (2024). Serverless Functions Documentation. https://vercel.com/docs/functions

[9] PlanetScale. (2024). MySQL at Scale. https://planetscale.com/docs

[10] Upstash. (2024). Redis for Serverless. https://upstash.com/docs/redis