logo.png

UtopiaHire

AI Career Architect for an Inclusive Future of Work

Complete Developer Guide & Technical Documentation

Development Team

TSYP13 Technical Challenge - 2025

Contents

List of Figures

List of Tables

Executive Summary

Project Overview

UtopiaHire is an AI-powered platform designed to promote fairness and inclusivity in employment across Sub-Saharan Africa and the MENA region. This comprehensive guide serves as the primary reference for developers working on the project.

Challenge Context

- **Competition**: IEEE TSYP13 Technical Challenge
- **Organizers**: IEEE CS Tunisia Section, IEEE CN Tunisia Section, IEEE CyberSecurity Local Group
- **Phase 1 Deadline**: November 16, 2025
- **Phase 2 Deadline**: December 21, 2025

Document Purpose

This guide provides:

- Comprehensive project architecture and design decisions
- Module-by-module technical specifications
- Implementation roadmap and best practices
- Security and privacy guidelines
- Testing and deployment strategies

Target Audience

- Backend developers
- Frontend developers
- AI/ML engineers
- DevOps engineers

- QA testers
- Project managers

Chapter 1

Introduction & Problem Statement

1.1 Challenge Background

1.1.1 The Problem We're Solving

Core Problem Statement

Job seekers in Sub-Saharan Africa and MENA regions face significant barriers to employment:

- **High Rejection Rates**: Outdated resumes and poor formatting lead to automatic filtering
- **Limited Access**: Lack of personalized career guidance and interview preparation
- **Visibility Gap**: Difficulty standing out in competitive job markets
- **Skills Mismatch**: Gap between candidate capabilities and employer requirements

1.1.2 Impact Statistics

- Youth unemployment in MENA: 25-30% (highest globally)
- Sub-Saharan Africa: 60% of unemployed are youth
- 70% of CVs rejected by ATS (Applicant Tracking Systems) due to formatting
- Limited access to career coaching in emerging regions

1.2 Our Solution: UtopiaHire

1.2.1 Vision Statement

"Democratize access to career development tools and create equal opportunities for job seekers in emerging regions through ethical AI technology."

1.2.2 Mission

To build an intelligent, fair, and secure platform that:

1. Empowers job seekers to optimize their applications
2. Improves candidate visibility to recruiters
3. Provides personalized career development insights
4. Ensures data privacy and ethical AI practices

1.2.3 Key Differentiators

Feature	**Competitive Advantage**
Regional Focus	Tailored for MENA & Sub-Saharan Africa job markets
Ethical AI	Bias detection, transparency, fairness-first approach
Privacy-First	End-to-end encryption, GDPR compliance, local data processing
Modular Design	Scalable architecture allowing feature expansion
Multilingual	Support for Arabic, French, English, and local languages
Offline Capability	Progressive Web App with offline features

Table 1.1: UtopiaHire Competitive Advantages

1.3 Core Modules Overview

1.3.1 Module Selection Strategy

We've selected the following modules for our MVP based on:

- Maximum impact for job seekers
- Technical feasibility within timeline
- Differentiation from existing solutions
- Alignment with challenge scoring criteria

1.3.2 Selected Modules

1. Resume Reviewer/Rewriter (Priority: CRITICAL)

Functionality:

- NLP-based resume analysis
- ATS compatibility scoring
- Content optimization suggestions
- Format standardization
- Keyword optimization for job descriptions

AI Techniques:

- Named Entity Recognition (NER) for skill extraction
- Transformer models (BERT/GPT) for content improvement
- Rule-based systems for formatting
- Similarity scoring for job matching

2. AI Interviewer & Profiler (Priority: HIGH)

Functionality:

- Virtual interview simulations
- Real-time speech analysis
- Behavioral assessment
- Confidence scoring
- Personalized improvement feedback

AI Techniques:

- Speech-to-text (Whisper, Google Speech API)
- Sentiment analysis
- Natural language understanding
- Computer vision for body language (optional)

3. Job Matcher (Priority: HIGH)

Functionality:

- Regional job opportunity discovery
- Skill-based matching algorithm
- Personalized job recommendations
- Company culture fit analysis
- Application tracking

AI Techniques:

- Collaborative filtering
- Content-based recommendation
- Graph neural networks for skill relationships
- Embedding-based similarity search

4. Footprint Scanner (Priority: MEDIUM)

Functionality:

- LinkedIn profile analysis
- GitHub contribution scanning
- StackOverflow reputation tracking
- Professional portfolio aggregation
- Online presence scoring

AI Techniques:

- Web scraping with ethical constraints
- API integration (LinkedIn, GitHub, SO)
- Activity pattern analysis
- Contribution quality assessment

1.4 Target User Personas

1.4.1 Persona 1: Fresh Graduate

- **Name**: Amina, 23, Computer Science Graduate
- **Location**: Tunis, Tunisia
- **Challenges**: No work experience, generic resume, nervous about interviews
- **Goals**: Land first job, improve interview skills, understand market requirements
- **UtopiaHire Value**: Resume optimization, interview practice, skill gap identification

1.4.2 Persona 2: Career Switcher

- **Name**: Kwame, 32, Former Teacher → Software Developer
- **Location**: Lagos, Nigeria
- **Challenges**: Highlighting transferable skills, portfolio visibility
- **Goals**: Career transition, demonstrate new skills, get noticed by recruiters
- **UtopiaHire Value**: Skills translation, GitHub showcase, targeted job matching

1.4.3 Persona 3: Experienced Professional

- **Name**: Fatima, 28, 5 years in Marketing
- **Location**: Cairo, Egypt
- **Challenges**: Standing out in competitive market, quantifying achievements
- **Goals**: Senior role, higher compensation, better company culture
- **UtopiaHire Value**: Achievement quantification, company culture matching

1.5 Success Criteria

1.5.1 Technical Metrics

- Resume improvement score: +30% average ATS compatibility
- Interview confidence boost: +40% self-reported confidence
- Job match accuracy: 75%+ relevance rating
- System response time: ¡2 seconds for most operations
- Uptime: 99.5%+

1.5.2 User Metrics

- User satisfaction: 4.5/5 stars
- Task completion rate: 90%+
- User retention: 60% monthly active users
- Interview success rate: +25% compared to baseline

1.5.3 Challenge Evaluation Criteria

Criterion	**Points**	**Our Strategy**
Problem Understanding	10	Deep regional research, persona development
Technical Approach	45	Modular architecture, proven AI techniques
Quality of Deliverables	15	Professional documentation, clean code
Booth Presentation	10	Interactive demo, visual materials
Security Aspect	10	Privacy-by-design, encryption, compliance
Presentation Quality	20	Clear storytelling, impact demonstration
Prototype Functionality	20	Working MVP with all core features
TOTAL	**130**	+ Bonus points (4 max)

Table 1.2: Challenge Scoring Alignment

1.6 Project Scope

1.6.1 In Scope (MVP)

- Resume upload, analysis, and rewriting
- AI-powered interview simulation (text-based)
- Job matching with regional databases
- Basic footprint scanning (LinkedIn, GitHub)
- User authentication and profile management
- Career insights dashboard
- Mobile-responsive web application

1.6.2 Out of Scope (Future Versions)

- Direct job application submission
- Employer/recruiter portal
- Video interview analysis
- Premium subscription features
- Mobile native applications
- Advanced analytics dashboard

1.6.3 Technical Constraints

- Budget: Limited (free-tier cloud services)
- Timeline: 8 weeks to Phase 1, 14 weeks to Phase 2
- Team Size: Maximum 6 members
- Anonymity: All submissions must be anonymous

Chapter 2

System Architecture & Design

2.1 System Architecture Overview

2.1.1 Architecture Principles

1. **Modularity**: Each module operates independently
2. **Scalability**: Horizontal scaling capability
3. **Security-First**: Zero-trust architecture
4. **Privacy-by-Design**: Data minimization and encryption
5. **API-First**: RESTful APIs for all services
6. **Cloud-Native**: Containerized microservices

2.1.2 High-Level Architecture

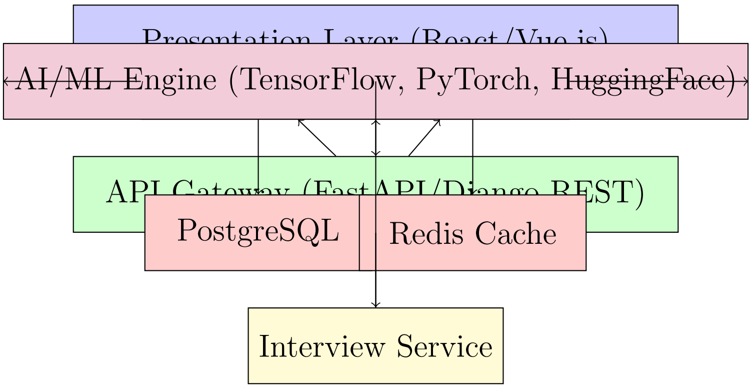
Presentation Layer (React/Vue.js)
AI/ML Engine (TensorFlow, PyTorch, HuggingFace)
API Gateway (FastAPI/Django REST)
PostgreSQL
Redis Cache
Interview Service

Figure 2.1: UtopiaHire System Architecture

Layer	Technology	Alternative	Rationale
Frontend	React.js	Vue.js	Large ecosystem, team expertise
Backend	Django + DRF	FastAPI	Mature, batteries included, admin panel
Database	PostgreSQL	MongoDB	ACID compliance, complex queries
Cache	Redis	Memcached	Pub/sub, persistence, data structures
AI/ML	Python + HuggingFace	Custom models	Pre-trained models, faster development
Deployment	Docker + K8s	Docker Swarm	Industry standard, scalability
Cloud	AWS/GCP	Azure	Free tier, documentation
CI/CD	GitHub Actions	GitLab CI	Integrated with repository

Table 2.1: Technology Stack Decisions

2.1.3 Technology Stack Decision Matrix

2.2 Detailed Architecture Components

2.2.1 Frontend Architecture

Component Structure

Listing 2.1: Frontend Directory Structure

```
src/
          components/
                common/
                      Header.jsx
                      Footer.jsx
                      Sidebar.jsx
                      Button.jsx
                resume/
                      ResumeUploader.jsx
                      ResumeAnalyzer.jsx
                      ResumeEditor.jsx
                      ATSScoreCard.jsx
                interview/
                      InterviewSimulator.jsx
                      QuestionPanel.jsx
                      ResponseRecorder.jsx
                      FeedbackDashboard.jsx
                jobs/
                      JobList.jsx
                      JobCard.jsx
                      JobFilters.jsx
                      MatchScoreBadge.jsx
                profile/
                    UserProfile.jsx
                    FootprintScanner.jsx
                    CareerInsights.jsx
          services/
                api.js
                auth.js
                resume.js
                interview.js
                jobs.js
          store/
                index.js
                slices/
                      authSlice.js
                      resumeSlice.js
                      jobsSlice.js
          utils/
                validators.js
                formatters.js
                constants.js
          App.jsx
```

State Management

- **Tool**: Redux Toolkit
- **Approach**: Feature-based slices
- **Middleware**: Redux Thunk for async operations
- **Persistence**: Redux Persist for offline capability

UI/UX Framework

- **Component Library**: Material-UI (MUI) or Tailwind CSS
- **Icons**: React Icons or Material Icons
- **Charts**: Recharts or Chart.js
- **Forms**: React Hook Form + Yup validation

2.2.2 Backend Architecture

Django Project Structure

Listing 2.2: Backend Directory Structure

```
utopiahire/
          manage.py
          config/
                settings/
```

```
                      base.py
                      development.py
                      production.py
                      test.py
                urls.py
                wsgi.py
                asgi.py
          apps/
                users/
                      models.py
                      serializers.py
                      views.py
                      urls.py
                      tests.py
                resume/
                      models.py
                      serializers.py
                      views.py
                      services/
                            parser.py
                            analyzer.py
                            rewriter.py
                      tasks.py (Celery)
                interview/
                      models.py
                      serializers.py
                      views.py
                      services/
                            question_generator.py
                            response_analyzer.py
                            feedback_generator.py
                      tasks.py
                jobs/
                      models.py
                      serializers.py
                      views.py
                      services/
                            scraper.py
                            matcher.py
                            recommender.py
                      tasks.py
                footprint/
                    models.py
                    services/
                          linkedin_scanner.py
                          github_scanner.py
                          stackoverflow_scanner.py
                    tasks.py
          core/
                utils/
                middleware/
```

```
              permissions.py
              exceptions.py
        ai_engine/
              models/
                    resume_nlp.py
                    interview_nlp.py
                    job_matching.py
              pipelines/
                    resume_pipeline.py
                    interview_pipeline.py
              utils/
                  preprocessing.py
                  embeddings.py
        tests/
        docs/
        requirements/
            base.txt
            development.txt
            production.txt
```

2.2.3 Database Schema Overview

Core Entities

Entity	**Key Attributes**
User	id, email, password_hash, profile_id, created_at, is_verified
Profile	id, user_id, full_name, location, languages, skills, experience_years
Resume	id, user_id, file_path, parsed_content, ats_score, version, created_at
ResumeAnalysis	id, resume_id, strengths, weaknesses, suggestions, keyword_density
Interview	id, user_id, job_role, difficulty, status, score, created_at
InterviewQuestion	id, interview_id, question_text, expected_answer, category
InterviewResponse	id, question_id, response_text, score, feedback, duration
Job	id, title, company, location, description, requirements, salary_range
JobApplication	id, user_id, job_id, status, match_score, applied_at
Footprint	id, user_id, platform, profile_url, score, last_scanned
CareerInsight	id, user_id, insight_type, content, priority, created_at

Table 2.2: Main Database Entities

2.2.4 API Architecture

RESTful API Design Principles

- **Versioning**: /api/v1/
- **Authentication**: JWT (JSON Web Tokens)

- **Authorization**: Role-based access control (RBAC)
- **Rate Limiting**: 100 requests/minute per user
- **Pagination**: Cursor-based for large datasets
- **Error Handling**: Standardized error responses

API Endpoint Structure

Listing 2.3: Core API Endpoints

```
# Authentication
POST   /api/v1/auth/register
POST   /api/v1/auth/login
POST   /api/v1/auth/logout
POST   /api/v1/auth/refresh
POST   /api/v1/auth/password-reset

# Resume
POST   /api/v1/resume/upload
GET    /api/v1/resume/:id
PUT    /api/v1/resume/:id
DELETE /api/v1/resume/:id
POST   /api/v1/resume/:id/analyze
POST   /api/v1/resume/:id/rewrite
GET    /api/v1/resume/:id/versions

# Interview
POST   /api/v1/interview/start
GET    /api/v1/interview/:id
POST   /api/v1/interview/:id/answer
POST   /api/v1/interview/:id/complete
GET    /api/v1/interview/:id/feedback

# Jobs
GET    /api/v1/jobs
GET    /api/v1/jobs/:id
POST   /api/v1/jobs/match
GET    /api/v1/jobs/recommendations
POST   /api/v1/jobs/:id/apply

# Footprint
POST   /api/v1/footprint/scan
GET    /api/v1/footprint/:platform
PUT    /api/v1/footprint/:id

# Career Insights
GET    /api/v1/insights
GET    /api/v1/insights/report
```

2.2.5 Microservices Communication

Service Mesh

- **Pattern**: API Gateway + Backend Services
- **Communication**: REST for synchronous, RabbitMQ/Kafka for async
- **Service Discovery**: Kubernetes DNS or Consul
- **Load Balancing**: Nginx or Kubernetes Ingress

Async Task Processing

- **Queue**: Celery + Redis/RabbitMQ
- **Use Cases**:
  - Resume parsing and analysis (CPU intensive)
  - AI model inference (long-running)
  - Email notifications
  - Job scraping and matching
  - Footprint scanning
- **Monitoring**: Flower (Celery monitoring tool)

2.3 Infrastructure & DevOps

2.3.1 Containerization

Listing 2.4: Docker Compose Structure

```
version: '3.8'
services:
  frontend:
    build: ./frontend
    ports:
      - "3000:3000"

  backend:
    build: ./backend
    ports:
      - "8000:8000"
    depends_on:
      - db
      - redis

  db:
    image: postgres:15
    volumes:
      - postgres_data:/var/lib/postgresql/data

  redis:
    image: redis:7-alpine

  celery:
    build: ./backend
    command: celery -A config worker
    depends_on:
      - redis

  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
```

2.3.2 CI/CD Pipeline

1. **Code Commit**: Push to GitHub
2. **Automated Tests**: Unit, integration, E2E tests
3. **Code Quality**: SonarQube, ESLint, Black
4. **Security Scan**: OWASP ZAP, Bandit, npm audit
5. **Build**: Docker images
6. **Deploy**: Staging environment
7. **Manual Approval**: For production
8. **Production Deploy**: Zero-downtime deployment
9. **Monitoring**: Post-deployment health checks

2.3.3 Monitoring & Logging

- **Application Monitoring**: Prometheus + Grafana
- **Logging**: ELK Stack (Elasticsearch, Logstash, Kibana)
- **Error Tracking**: Sentry
- **APM**: New Relic or Datadog (if budget allows)
- **Uptime Monitoring**: UptimeRobot or Pingdom

Chapter 3

Core Modules Specification

3.1 Module 1: Resume Reviewer/Rewriter

3.1.1 Functional Requirements

Upload & Parsing

- Accept multiple formats: PDF, DOCX, TXT
- Extract structured data: personal info, education, experience, skills
- Handle multiple languages: English, French, Arabic
- OCR support for scanned documents

Analysis Features

Resume Analysis Components

1. **ATS Compatibility Score** (0-100)
   - Format compliance check
   - Keyword density analysis
   - Section structure validation
   - File format optimization
2. **Content Quality Analysis**
   - Action verb usage
   - Quantified achievements detection
   - Buzzword/cliché identification
   - Grammar and spelling check
3. **Skills Gap Analysis**
   - Compare to target job description
   - Identify missing technical skills
   - Suggest relevant certifications
   - Highlight transferable skills
4. **Optimization Suggestions**
   - Keyword placement recommendations
   - Section reordering advice
   - Length optimization (1-2 pages)
   - Formatting improvements

Rewriting Capabilities

- AI-powered content enhancement
- Industry-specific language adaptation
- Achievement quantification suggestions
- Professional tone adjustment
- Multiple rewrite suggestions per section
- Side-by-side comparison view

3.1.2 Technical Implementation

Resume Parser Architecture

Listing 3.1: Resume Parser Flow

```
class ResumeParser:
    def __init__(self):
        self.nlp = spacy.load("en_core_web_lg")
        self.ner_model = load_custom_ner_model()

    def parse(self, file_path: str) -> dict:
        # Extract text
        text = self.extract_text(file_path)

        # Identify sections
        sections = self.segment_sections(text)

        # Extract entities
        entities = self.extract_entities(sections)

        # Structure data
        structured_data = {
            'personal_info': entities['personal'],
            'education': entities['education'],
            'experience': entities['experience'],
            'skills': entities['skills'],
            'certifications': entities['certifications']
        }

        return structured_data
```

NLP Models & Techniques

Task	**Model/Library**	**Purpose**
Text Extraction	PyPDF2, python-docx	Extract text from documents
NER	spaCy + Custom Model	Extract names, dates, companies
Section Detection	Rule-based + ML	Identify resume sections
Skill Extraction	SkillNER, Custom NER	Extract technical & soft skills
Content Generation	GPT-3.5/GPT-4 API	Rewrite suggestions
Grammar Check	LanguageTool	Grammar and spelling
Keyword Extraction	TF-IDF, RAKE	Important keywords

Table 3.1: Resume Module NLP Stack

ATS Score Calculation

Listing 3.2: ATS Score Algorithm

```python
def calculate_ats_score(resume_data: dict) -> dict:
    score_components = {
        'format': check_format_compliance(resume_data),
        'keywords': analyze_keyword_density(resume_data),
        'structure': validate_section_structure(resume_data),
        'readability': calculate_readability_score(resume_data),
        'completeness': check_section_completeness(resume_data)
    }

    weights = {
        'format': 0.25,
        'keywords': 0.30,
        'structure': 0.20,
        'readability': 0.15,
        'completeness': 0.10
    }

    total_score = sum(
        score_components[k] * weights[k]
        for k in score_components
    )

    return {
        'total_score': round(total_score, 2),
        'breakdown': score_components,
        'recommendations': generate_recommendations(
           score_components)
    }
```

3.1.3 User Workflows

Workflow 1: Quick Analysis

1. User uploads resume
2. System parses and extracts data (30-60 seconds)
3. ATS score displayed immediately
4. Top 3 improvement suggestions shown
5. Option to proceed to detailed analysis

Workflow 2: Job-Targeted Optimization

1. User uploads resume
2. User provides target job description (paste or URL)
3. System performs comparative analysis
4. Skills gap identified

5. Keyword optimization suggestions
6. Tailored resume version generated

Workflow 3: Full Rewrite

1. User uploads resume
2. Selects target industry/role
3. AI generates enhanced version
4. Side-by-side comparison shown
5. User accepts/rejects suggestions section-by-section
6. Download optimized resume

3.1.4 Data Models

Listing 3.3: Resume Models

```
from django.db import models
from django.contrib.auth import get_user_model

User = get_user_model()

class Resume(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    original_file = models.FileField(upload_to='resumes/original/
       ')
    file_type = models.CharField(max_length=10)
    version = models.IntegerField(default=1)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

class ResumeContent(models.Model):
    resume = models.OneToOneField(Resume, on_delete=models.
       CASCADE)
    raw_text = models.TextField()
    personal_info = models.JSONField()
    education = models.JSONField()
    experience = models.JSONField()
    skills = models.JSONField()
    certifications = models.JSONField()

class ResumeAnalysis(models.Model):
    resume = models.OneToOneField(Resume, on_delete=models.
       CASCADE)
    ats_score = models.FloatField()
    score_breakdown = models.JSONField()
    strengths = models.JSONField()
```

```python
    weaknesses = models.JSONField()
    suggestions = models.JSONField()
    analyzed_at = models.DateTimeField(auto_now_add=True)

class ResumeRewrite(models.Model):
    resume = models.ForeignKey(Resume, on_delete=models.CASCADE)
    section = models.CharField(max_length=50)
    original_content = models.TextField()
    rewritten_content = models.TextField()
    status = models.CharField(max_length=20)  # pending, accepted, rejected
    created_at = models.DateTimeField(auto_now_add=True)
```

3.2 Module 2: AI Interviewer & Profiler

3.2.1 Functional Requirements

Interview Simulation

- **Job Role Selection**: Choose target position
- **Difficulty Levels**: Junior, Mid-level, Senior
- **Interview Types**:
  - Behavioral (STAR method)
  - Technical (role-specific)
  - Situational (problem-solving)
  - Culture Fit
- **Question Modes**:
  - Text-based (MVP)
  - Voice-enabled (future)
  - Video-based (future)

Response Analysis

Response Evaluation Criteria

1. **Content Quality** (40%)
   - Relevance to question
   - Completeness of answer
   - Use of STAR framework
   - Specific examples provided
2. **Communication Skills** (30%)
   - Clarity and coherence
   - Professional language
   - Grammar and vocabulary
   - Conciseness vs. verbosity
3. **Technical Accuracy** (20%)
   - Correct technical concepts
   - Industry best practices
   - Up-to-date knowledge
4. **Confidence Indicators** (10%)
   - Decisive language
   - Avoiding filler words
   - Positive framing

Feedback & Coaching

- Immediate response scoring (0-100)
- Detailed feedback for each answer
- Suggested improvements
- Model answers provided
- Overall interview performance report
- Strength & weakness summary
- Personalized practice recommendations

3.2.2 Technical Implementation

Question Generation

Listing 3.4: Interview Question Generator

```
class InterviewQuestionGenerator:
    def __init__(self):
        self.question_bank = QuestionBank()
        self.llm = OpenAI(api_key=settings.OPENAI_API_KEY)

    def generate_questions(self, job_role: str,
                           difficulty: str,
                           count: int = 10) -> list:
        # Get base questions from database
        base_questions = self.question_bank.get_questions(
            role=job_role,
            difficulty=difficulty,
            limit=count // 2
        )

        # Generate contextual questions using LLM
        prompt = f"""Generate␣{count␣//␣2}␣interview␣questions
␣␣␣␣␣␣␣␣for␣a␣{difficulty}␣{job_role}␣position.
␣␣␣␣␣␣␣␣Focus␣on:␣behavioral,␣technical,␣and␣situational␣aspects."""

        ai_questions = self.llm.generate(prompt)

        # Combine and randomize
        all_questions = base_questions + ai_questions
        random.shuffle(all_questions)

        return all_questions[:count]
```

Response Evaluation Pipeline

1. **Text Preprocessing**: Tokenization, normalization
2. **Sentiment Analysis**: Detect confidence level
3. **Keyword Extraction**: Identify key concepts
4. **Semantic Similarity**: Compare to model answers
5. **Structure Analysis**: Check STAR framework
6. **LLM Evaluation**: GPT-4 detailed assessment
7. **Score Aggregation**: Weighted scoring
8. **Feedback Generation**: Actionable suggestions

AI Models Used

3.2.3 Data Models

Component	Model	Purpose
Question Generation	GPT-3.5 Turbo	Dynamic question creation
Response Evaluation	GPT-4	Detailed answer assessment
Sentiment Analysis	BERT-based	Confidence detection
Semantic Similarity	Sentence-BERT	Compare to ideal answers
Speech Recognition	Whisper (future)	Voice interview support
Text-to-Speech	Google TTS (future)	Voice questions

Table 3.2: Interview Module AI Stack

Listing 3.5: Interview Models

```
class Interview(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    job_role = models.CharField(max_length=100)
    difficulty = models.CharField(max_length=20)
    interview_type = models.CharField(max_length=30)
    status = models.CharField(max_length=20)
    overall_score = models.FloatField(null=True)
    started_at = models.DateTimeField(auto_now_add=True)
    completed_at = models.DateTimeField(null=True)

class InterviewQuestion(models.Model):
    interview = models.ForeignKey(Interview, on_delete=models.
       CASCADE)
    question_text = models.TextField()
    question_type = models.CharField(max_length=30)
    category = models.CharField(max_length=50)
    difficulty = models.CharField(max_length=20)
    order = models.IntegerField()
    model_answer = models.TextField(null=True)

class InterviewResponse(models.Model):
    question = models.OneToOneField(InterviewQuestion,
                                   on_delete=models.CASCADE)
    response_text = models.TextField()
    score = models.FloatField()
    feedback = models.TextField()
    strengths = models.JSONField()
    improvements = models.JSONField()
    time_taken = models.IntegerField()  # seconds
    submitted_at = models.DateTimeField(auto_now_add=True)

class InterviewReport(models.Model):
    interview = models.OneToOneField(Interview,
                                    on_delete=models.CASCADE)
    overall_feedback = models.TextField()
    strengths_summary = models.JSONField()
    weaknesses_summary = models.JSONField()
    recommendations = models.JSONField()
    skill_scores = models.JSONField()
    generated_at = models.DateTimeField(auto_now_add=True)
```

3.3 Module 3: Job Matcher

3.3.1 Functional Requirements

Job Discovery

- Aggregate jobs from multiple sources
- Regional focus: MENA & Sub-Saharan Africa
- Real-time job scraping
- Manual job posting support
- Company profile integration

Matching Algorithm

Job Matching Factors

1. **Skills Match** (35%)
   - Technical skills overlap
   - Soft skills alignment
   - Required vs. preferred skills
2. **Experience Level** (25%)
   - Years of experience
   - Industry relevance
   - Role progression
3. **Location & Remote** (15%)
   - Geographic proximity
   - Remote work availability
   - Relocation willingness
4. **Education & Certifications** (15%)
   - Degree requirements
   - Relevant certifications
   - Continuous learning
5. **Company Culture** (10%)
   - Values alignment
   - Company size preference
   - Industry interest

Recommendation Features

- Personalized job feed
- Daily/weekly email digests
- Match score explanation
- ”Why this match?” insights
- Similar jobs suggestions
- Saved jobs & application tracking

3.3.2 Technical Implementation

Job Scraping Pipeline

Listing 3.6: Job Scraper

```
class JobScraper:
    def __init__(self):
        self.sources = {
            'linkedin': LinkedInScraper(),
            'indeed': IndeedScraper(),
            'glassdoor': GlassdoorScraper(),
            'local_boards': LocalJobBoardsScraper()
        }

    def scrape_jobs(self, region: str, keywords: list):
        all_jobs = []

        for source_name, scraper in self.sources.items():
            try:
                jobs = scraper.scrape(region, keywords)
                all_jobs.extend(jobs)
            except Exception as e:
                logger.error(f"Error␣scraping␣{source_name}:␣{e}")

        # Deduplicate
        unique_jobs = self.deduplicate_jobs(all_jobs)

        # Store in database
        self.save_jobs(unique_jobs)

        return unique_jobs
```

Matching Algorithm

Listing 3.7: Job Matching Engine

```
class JobMatcher:
    def __init__(self):
        self.skill_embedder = SentenceTransformer(
            'sentence-transformers/all-MiniLM-L6-v2'
        )

    def calculate_match_score(self, user_profile: dict,
                              job: dict) -> dict:
        # Skills similarity
        skills_score = self.compute_skills_similarity(
            user_profile['skills'],
            job['required_skills']
        )

        # Experience match
        exp_score = self.compute_experience_match(
            user_profile['experience_years'],
            job['experience_required']
        )

        # Location score
        loc_score = self.compute_location_score(
            user_profile['location'],
            job['location'],
            job['remote_allowed']
        )

        # Education match
        edu_score = self.compute_education_match(
            user_profile['education'],
            job['education_required']
        )

        # Weighted total
        total_score = (
            skills_score * 0.35 +
            exp_score * 0.25 +
            loc_score * 0.15 +
            edu_score * 0.15 +
            0.10  # culture (placeholder)
        )

        return {
            'total_score': total_score,
            'breakdown': {
                'skills': skills_score,
                'experience': exp_score,
                'location': loc_score,
                'education': edu_score
            },
            'explanation': self.generate_explanation(...)
        }
```

3.4 Module 4: Footprint Scanner

3.4.1 Functional Requirements

- **LinkedIn**: Profile completeness, endorsements, recommendations
- **GitHub**: Repository analysis, contribution patterns, code quality
- **StackOverflow**: Reputation, answered questions, expertise tags
- **Portfolio**: Website analysis, project showcase

- **Aggregate Score**: Overall online professional presence

3.4.2 Technical Implementation

Listing 3.8: Footprint Scanner

```
class FootprintScanner:
    def scan_linkedin(self, profile_url: str) -> dict:
        # Use Selenium or API (if available)
        profile_data = linkedin_scraper.extract(profile_url)

        score = calculate_linkedin_score(profile_data)

        return {
            'platform': 'LinkedIn',
            'score': score,
            'metrics': profile_data,
            'suggestions': generate_linkedin_suggestions(
                profile_data)
        }

    def scan_github(self, username: str) -> dict:
        # Use GitHub API
        repos = github_api.get_user_repos(username)
        contributions = github_api.get_contributions(username)

        score = calculate_github_score(repos, contributions)

        return {
            'platform': 'GitHub',
            'score': score,
            'metrics': {
                'repos': len(repos),
                'stars': sum(r['stars'] for r in repos),
                'contributions': contributions
            }
        }
```

Chapter 4

AI & Machine Learning Components

4.1 AI/ML Technology Stack

4.1.1 NLP Models & Frameworks

Pre-trained Models

Model	Provider	Use Case	Why
GPT-3.5/4	OpenAI	Resume rewriting, feedback	High quality, multilingual
BERT	Google	Text classification	Proven for NLP tasks
RoBERTa	Meta	Skill extraction	Better than BERT
Sentence-BERT	SBERT	Semantic similarity	Fast embeddings
spaCy (en, fr, ar)	Explosion AI	NER, parsing	Production-ready
LaBSE	Google	Multilingual embeddings	109+ languages

Table 4.1: NLP Models Selection

4.1.2 Model Training & Fine-tuning

Custom NER for Resume Parsing

Listing 4.1: Fine-tune NER Model

```
import spacy
from spacy.training import Example

# Load base model
nlp = spacy.load("en_core_web_lg")

# Add custom entity recognizer
ner = nlp.get_pipe("ner")

# Add custom labels
custom_labels = [
    "SKILL",
    "CERTIFICATION",
    "JOB_TITLE",
    "COMPANY",
    "EDUCATION_DEGREE"
]

for label in custom_labels:
    ner.add_label(label)

# Training data format
TRAIN_DATA = [
    ("I␣have␣experience␣in␣Python␣and␣Django",
     {"entities": [(25, 31, "SKILL"), (36, 42, "SKILL")]}),
    ("Certified␣AWS␣Solutions␣Architect",
     {"entities": [(10, 33, "CERTIFICATION")]}),
    # ... more training examples
]

# Train the model
optimizer = nlp.begin_training()
for epoch in range(30):
    random.shuffle(TRAIN_DATA)
    losses = {}

    for text, annotations in TRAIN_DATA:
        example = Example.from_dict(
            nlp.make_doc(text),
            annotations
        )
        nlp.update([example], sgd=optimizer, losses=losses)

    print(f"Epoch␣{epoch},␣Losses:␣{losses}")

# Save model
nlp.to_disk("./models/resume_ner_model")
```

Skill Extraction with Custom Model

Listing 4.2: Skills Extractor

```
class SkillExtractor:
    def __init__(self):
        self.nlp = spacy.load("./models/resume_ner_model")
        self.skill_database = self.load_skill_taxonomy()

    def extract_skills(self, text: str) -> dict:
        doc = self.nlp(text)

        # Extract from NER
        ner_skills = [
            ent.text for ent in doc.ents
            if ent.label_ == "SKILL"
        ]

        # Pattern matching for technical skills
        pattern_skills = self.match_skill_patterns(text)

        # Fuzzy matching against database
        db_skills = self.fuzzy_match_skills(
            text,
            self.skill_database
        )

        # Combine and categorize
        all_skills = list(set(
            ner_skills + pattern_skills + db_skills
        ))

        categorized = self.categorize_skills(all_skills)

        return {
            'technical_skills': categorized['technical'],
            'soft_skills': categorized['soft'],
            'tools': categorized['tools'],
            'languages': categorized['languages'],
            'all_skills': all_skills
        }

    def load_skill_taxonomy(self):
        # Load from JSON/database
        # Contains hierarchical skill taxonomy
        # e.g., Programming > Python > Django
        return skill_taxonomy_data
```

4.1.3 Recommendation System

Collaborative Filtering

Listing 4.3: Job Recommendation Engine

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from scipy.sparse.linalg import svds

class CollaborativeJobRecommender:
    def __init__(self):
        self.user_job_matrix = None
        self.user_factors = None
        self.job_factors = None

    def train(self, user_job_interactions):
        """
␣␣␣␣␣␣␣␣user_job_interactions:␣sparse␣matrix
␣␣␣␣␣␣␣␣rows␣=␣users,␣cols␣=␣jobs,␣values␣=␣interaction␣score
```

```
␣␣␣␣␣␣␣␣"""
        # Matrix factorization using SVD
        U, sigma, Vt = svds(
            user_job_interactions,
            k=50
        )

        sigma = np.diag(sigma)

        self.user_factors = U
        self.job_factors = Vt.T
        self.sigma = sigma

        # Predicted ratings
        self.predicted_ratings = np.dot(
            np.dot(U, sigma),
            Vt
        )

    def recommend_jobs(self, user_id: int, top_n: int = 10):
        user_idx = user_id  # assuming user_id = index

        # Get predicted scores for all jobs
        user_predictions = self.predicted_ratings[user_idx]

        # Get top N job indices
        top_job_indices = np.argsort(user_predictions)[
            -top_n:
        ][::-1]

        return top_job_indices
```

Content-Based Filtering

Listing 4.4: Content-Based Recommender

```
from sentence_transformers import SentenceTransformer

class ContentBasedRecommender:
    def __init__(self):
        self.model = SentenceTransformer(
            'sentence-transformers/all-MiniLM-L6-v2'
        )
        self.job_embeddings = {}

    def encode_jobs(self, jobs: list):
        """Encode␣job␣descriptions"""
        for job in jobs:
            # Combine relevant fields
            job_text = f"""
␣␣␣␣␣␣␣␣␣␣␣␣{job['title']}
```

```
␣␣␣␣␣␣␣␣␣␣␣␣{job['description']}
␣␣␣␣␣␣␣␣␣␣␣␣{'␣'.join(job['required_skills'])}
␣␣␣␣␣␣␣␣␣␣␣␣"""

            embedding = self.model.encode(job_text)
            self.job_embeddings[job['id']] = embedding

    def encode_user_profile(self, user_profile: dict):
        """Encode␣user␣profile"""
        profile_text = f"""
␣␣␣␣␣␣␣␣{user_profile['desired_role']}
␣␣␣␣␣␣␣␣{'␣'.join(user_profile['skills'])}
␣␣␣␣␣␣␣␣{user_profile['experience_summary']}
␣␣␣␣␣␣␣␣"""

        return self.model.encode(profile_text)

    def recommend(self, user_profile: dict, top_n: int = 10):
        user_embedding = self.encode_user_profile(user_profile)

        # Calculate similarities
        similarities = {}
        for job_id, job_emb in self.job_embeddings.items():
            sim = cosine_similarity(
                [user_embedding],
                [job_emb]
            )[0][0]
            similarities[job_id] = sim

        # Get top N
        top_jobs = sorted(
            similarities.items(),
            key=lambda x: x[1],
            reverse=True
        )[:top_n]

        return top_jobs
```

Hybrid Recommendation

Listing 4.5: Hybrid Recommender

```
class HybridRecommender:
    def __init__(self):
        self.collaborative = CollaborativeJobRecommender()
        self.content_based = ContentBasedRecommender()

    def recommend(self, user_id: int, user_profile: dict,
                  top_n: int = 10, alpha: float = 0.5):
        """
␣␣␣␣␣␣␣␣alpha:␣weight␣for␣collaborative␣(1-alpha␣for␣content)
```

```
␣␣␣␣␣␣␣␣"""
        # Get collaborative recommendations
        collab_scores = self.collaborative.get_scores(user_id)

        # Get content-based recommendations
        content_scores = self.content_based.get_scores(
            user_profile
        )

        # Normalize scores
        collab_norm = self.normalize(collab_scores)
        content_norm = self.normalize(content_scores)

        # Combine
        hybrid_scores = {}
        all_job_ids = set(collab_norm.keys()) | set(
            content_norm.keys()
        )

        for job_id in all_job_ids:
            collab = collab_norm.get(job_id, 0)
            content = content_norm.get(job_id, 0)

            hybrid_scores[job_id] = (
                alpha * collab + (1 - alpha) * content
            )

        # Get top N
        top_jobs = sorted(
            hybrid_scores.items(),
            key=lambda x: x[1],
            reverse=True
        )[:top_n]

        return top_jobs
```

4.1.4 Interview Response Evaluation

STAR Framework Detector

Listing 4.6: STAR Framework Analyzer

```
class STARAnalyzer:
    """
␣␣␣␣Detect␣STAR␣framework␣components:
␣␣␣␣-␣Situation
␣␣␣␣-␣Task
␣␣␣␣-␣Action
␣␣␣␣-␣Result
␣␣␣␣"""

```

```
    def __init__(self):
        self.situation_patterns = [
            r"when␣(I|we)␣(was|were)",
            r"at␣(my|the)␣(previous|last)␣(job|company)",
            r"during␣my␣time␣at"
        ]

        self.task_patterns = [
            r"(I|we)␣(had␣to|needed␣to|was␣asked␣to)",
            r"my␣(role|responsibility)␣was"
        ]

        self.action_patterns = [
            r"(I|we)␣(did|created|implemented|developed)",
            r"(I|we)␣decided␣to",
            r"my␣approach␣was"
        ]

        self.result_patterns = [
            r"(as␣a␣result|consequently|this␣led␣to)",
            r"increased␣.*␣by␣\d+%",
            r"reduced␣.*␣by␣\d+%",
            r"achieved|accomplished"
        ]

    def analyze(self, response_text: str) -> dict:
        has_situation = self.check_patterns(
            response_text,
            self.situation_patterns
        )
        has_task = self.check_patterns(
            response_text,
            self.task_patterns
        )
        has_action = self.check_patterns(
            response_text,
            self.action_patterns
        )
        has_result = self.check_patterns(
            response_text,
            self.result_patterns
        )

        components_found = sum([
            has_situation, has_task, has_action, has_result
        ])

        return {
            ’has_situation’: has_situation,
            ’has_task’: has_task,
            ’has_action’: has_action,
            ’has_result’: has_result,
            ’star_score’: components_found / 4 * 100,
            ’missing_components’: self.get_missing(
                has_situation, has_task,
                has_action, has_result
            )
        }
```

Response Quality Scorer

Listing 4.7: Response Quality Evaluation

```
class ResponseQualityScorer:
    def __init__(self):
        self.sentiment_analyzer = pipeline(
            "sentiment-analysis",
            model="distilbert-base-uncased-finetuned-sst-2-english"
        )
        self.grammar_checker = LanguageTool(’en-US’)

    def score_response(self, question: str,
                       response: str,
                       ideal_answer: str = None) -> dict:

        # 1. Length check
        word_count = len(response.split())
        length_score = self.score_length(word_count)

        # 2. Grammar and spelling
        grammar_errors = len(
            self.grammar_checker.check(response)
        )
        grammar_score = max(0, 100 - grammar_errors * 5)

        # 3. Sentiment (confidence)
        sentiment = self.sentiment_analyzer(response)[0]
        confidence_score = (
            sentiment[’score’] * 100
            if sentiment[’label’] == ’POSITIVE’
            else 50
        )

        # 4. Relevance (semantic similarity)
        relevance_score = 0
        if ideal_answer:
            relevance_score = self.compute_similarity(
                response,
                ideal_answer
            )

        # 5. STAR framework
        star_analysis = STARAnalyzer().analyze(response)
        star_score = star_analysis['star_score']

        # 6. Technical accuracy (using LLM)
        technical_score = self.llm_evaluate_technical(
            question,
            response
        )

        # Weighted total
        total_score = (
            length_score * 0.10 +
            grammar_score * 0.15 +
            confidence_score * 0.10 +
            relevance_score * 0.20 +
            star_score * 0.25 +
            technical_score * 0.20
        )

        return {
            'total_score': round(total_score, 2),
            'breakdown': {
                'length': length_score,
                'grammar': grammar_score,
                'confidence': confidence_score,
                'relevance': relevance_score,
                'star_framework': star_score,
                'technical': technical_score
            },
            'feedback': self.generate_feedback(...),
            'star_analysis': star_analysis
        }
```

4.1.5 Model Deployment

Model Serving Architecture

Listing 4.8: Model Serving with FastAPI

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import torch

app = FastAPI()

# Load models at startup
class ModelManager:
    def __init__(self):
        self.resume_ner = spacy.load("./models/resume_ner")
        self.skill_extractor = SkillExtractor()
        self.sentence_transformer = SentenceTransformer(
            ’all-MiniLM-L6-v2’
        )

    def extract_skills(self, text: str):
        return self.skill_extractor.extract_skills(text)

model_manager = ModelManager()

class ResumeRequest(BaseModel):
    text: str

@app.post("/api/v1/ml/extract-skills")
async def extract_skills(request: ResumeRequest):
    try:
        skills = model_manager.extract_skills(request.text)
        return {"skills": skills}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.post("/api/v1/ml/compute-similarity")
async def compute_similarity(text1: str, text2: str):
    emb1 = model_manager.sentence_transformer.encode(text1)
    emb2 = model_manager.sentence_transformer.encode(text2)

    similarity = cosine_similarity([emb1], [emb2])[0][0]

    return {"similarity": float(similarity)}
```

4.1.6 Model Performance Optimization

Caching Strategy

Listing 4.9: Model Response Caching

```
import hashlib
from functools import wraps
import redis

redis_client = redis.Redis(
    host=’localhost’,
    port=6379,
    decode_responses=True
)

def cache_ml_response(expiry=3600):
    """Cache␣ML␣model␣responses"""
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Create cache key
            key_data = str(args) + str(kwargs)
            cache_key = f"ml:{func.__name__}:{hashlib.md5(
               key_data.encode()).hexdigest()}"

            # Check cache
            cached = redis_client.get(cache_key)
            if cached:
                return json.loads(cached)

            # Compute result
            result = func(*args, **kwargs)

            # Store in cache
            redis_client.setex(
                cache_key,
                expiry,
                json.dumps(result)
            )

            return result
        return wrapper
    return decorator

@cache_ml_response(expiry=7200)
def extract_resume_skills(resume_text: str):
    return model_manager.extract_skills(resume_text)
```

Batch Processing

Listing 4.10: Batch Inference

```
class BatchProcessor:
    def __init__(self, batch_size=32):
        self.batch_size = batch_size
        self.queue = []

    def add_to_queue(self, item):
        self.queue.append(item)

        if len(self.queue) >= self.batch_size:
            return self.process_batch()

        return None

    def process_batch(self):
        if not self.queue:
            return []

        # Process all items in batch
        texts = [item['text'] for item in self.queue]

        # Batch encoding (much faster than one-by-one)
        embeddings = model.encode(
            texts,
            batch_size=self.batch_size
        )

        results = []
        for item, embedding in zip(self.queue, embeddings):
            results.append({
                'id': item['id'],
                'embedding': embedding.tolist()
            })

        # Clear queue
        self.queue = []

        return results
```

4.1.7 Model Monitoring

Performance Tracking

Listing 4.11: ML Model Monitoring

```
import time
from prometheus_client import Counter, Histogram

# Metrics
ml_request_counter = Counter(
    'ml_requests_total',
    'Total␣ML␣requests',
    ['model', 'endpoint']
)

ml_latency_histogram = Histogram(
    'ml_request_duration_seconds',
    'ML␣request␣latency',
    ['model', 'endpoint']
)

ml_error_counter = Counter(
    'ml_errors_total',
    'Total␣ML␣errors',
    ['model', 'error_type']
)

def monitor_ml_performance(model_name, endpoint):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            start_time = time.time()

            try:
                result = func(*args, **kwargs)
                ml_request_counter.labels(
                    model=model_name,
                    endpoint=endpoint
                ).inc()

                return result

            except Exception as e:
                ml_error_counter.labels(
                    model=model_name,
                    error_type=type(e).__name__
                ).inc()
                raise

            finally:
                duration = time.time() - start_time
                ml_latency_histogram.labels(
                    model=model_name,
                    endpoint=endpoint
                ).observe(duration)

        return wrapper
    return decorator
```

Chapter 5

Security & Privacy Framework

5.1 Security-by-Design Principles

5.1.1 Core Security Requirements

CRITICAL: Security is a Challenge Criterion (10 points)

The challenge explicitly evaluates security aspects. Our platform MUST demonstrate:

- **Privacy-by-Design**: Data minimization, encryption, user consent
- **Secure Authentication**: Multi-factor, strong password policies
- **Data Protection**: GDPR compliance, right to deletion
- **Secure Communication**: HTTPS/TLS, encrypted data in transit
- **Vulnerability Management**: Regular scans, penetration testing

5.1.2 Authentication & Authorization

Multi-Factor Authentication (MFA)

Listing 5.1: MFA Implementation

```
from django.contrib.auth.models import AbstractBaseUser
from django_otp.plugins.otp_totp.models import TOTPDevice
import pyotp

class User(AbstractBaseUser):
    email = models.EmailField(unique=True)
    phone_number = models.CharField(max_length=15, blank=True)
    is_mfa_enabled = models.BooleanField(default=False)
    mfa_secret = models.CharField(max_length=32, blank=True)

    def enable_mfa(self):
        """Generate␣TOTP␣secret␣for␣MFA"""
        self.mfa_secret = pyotp.random_base32()
```

```python
        self.is_mfa_enabled = True
        self.save()

        # Generate QR code for authenticator apps
        totp_uri = pyotp.totp.TOTP(self.mfa_secret).provisioning_uri(
            name=self.email,
            issuer_name='UtopiaHire'
        )
        return totp_uri

    def verify_mfa_token(self, token: str) -> bool:
        """Verify␣MFA␣token"""
        if not self.is_mfa_enabled:
            return True

        totp = pyotp.TOTP(self.mfa_secret)
        return totp.verify(token, valid_window=1)

# Login view with MFA
class LoginView(APIView):
    def post(self, request):
        email = request.data.get('email')
        password = request.data.get('password')
        mfa_token = request.data.get('mfa_token')

        user = authenticate(email=email, password=password)

        if user is None:
            return Response(
                {'error': 'Invalid␣credentials'},
                status=401
            )

        # Check MFA if enabled
        if user.is_mfa_enabled:
            if not mfa_token:
                return Response({
                    'mfa_required': True,
                    'message': 'Please␣provide␣MFA␣token'
                }, status=200)

            if not user.verify_mfa_token(mfa_token):
                return Response(
                    {'error': 'Invalid␣MFA␣token'},
                    status=401
                )

        # Generate JWT token
        tokens = get_tokens_for_user(user)
        return Response(tokens, status=200)
```

JWT Token Management

Listing 5.2: Secure JWT Implementation

```python
from rest_framework_simplejwt.tokens import RefreshToken
from datetime import timedelta
import secrets

# settings.py
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=15),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=7),
    'ROTATE_REFRESH_TOKENS': True,
    'BLACKLIST_AFTER_ROTATION': True,
    'ALGORITHM': 'HS256',
    'SIGNING_KEY': settings.SECRET_KEY,
    'AUTH_HEADER_TYPES': ('Bearer',),
    'USER_ID_FIELD': 'id',
    'USER_ID_CLAIM': 'user_id',
}

# Token generation with additional claims
def get_tokens_for_user(user):
    refresh = RefreshToken.for_user(user)

    # Add custom claims
    refresh['email'] = user.email
    refresh['role'] = user.role
    refresh['jti'] = secrets.token_hex(16)  # JWT ID

    return {
        'refresh': str(refresh),
        'access': str(refresh.access_token),
        'expires_in': 900  # 15 minutes
    }
```

Role-Based Access Control (RBAC)

Listing 5.3: RBAC Implementation

```python
from rest_framework import permissions

class IsOwnerOrReadOnly(permissions.BasePermission):
    """Allow␣owners␣to␣edit,␣others␣to␣read␣only"""

    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            return True

        return obj.user == request.user

class CanAccessResumeAnalysis(permissions.BasePermission):
    """Check␣if␣user␣can␣access␣resume␣analysis"""

    def has_permission(self, request, view):
        # Premium feature check
        if not request.user.is_authenticated:
            return False

        # Check if user has remaining analysis credits
        if request.user.analysis_credits <= 0:
            return False

        return True

# Usage in views
class ResumeAnalysisView(APIView):
    permission_classes = [
        permissions.IsAuthenticated,
        CanAccessResumeAnalysis
    ]

    def post(self, request, resume_id):
        # Perform analysis
        ...
        # Deduct credit
        request.user.analysis_credits -= 1
        request.user.save()
        ...
```

5.1.3 Data Encryption

Encryption at Rest

Listing 5.4: Field-Level Encryption

```
from cryptography.fernet import Fernet
from django.conf import settings
import base64

class EncryptedField(models.TextField):
    """Custom␣encrypted␣field"""

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.cipher = Fernet(settings.FIELD_ENCRYPTION_KEY)

    def from_db_value(self, value, expression, connection):
        if value is None:
            return value

        # Decrypt when reading from database
        return self.cipher.decrypt(
            value.encode()
        ).decode()

    def get_prep_value(self, value):
        if value is None:
            return value

        # Encrypt before saving to database
        return self.cipher.encrypt(
            value.encode()
        ).decode()

# Usage
class Resume(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    content = EncryptedField()  # Encrypted resume content
    personal_info = EncryptedField()  # Encrypted PII
```

File Encryption

Listing 5.5: Resume File Encryption

```
from cryptography.fernet import Fernet
import os

class SecureFileStorage:
    def __init__(self):
        self.cipher = Fernet(settings.FILE_ENCRYPTION_KEY)

    def encrypt_file(self, file_path: str) -> str:
        """Encrypt␣uploaded␣file"""
        with open(file_path, ’rb’) as f:
            file_data = f.read()

        # Encrypt
        encrypted_data = self.cipher.encrypt(file_data)

        # Save encrypted file
        encrypted_path = f"{file_path}.encrypted"
        with open(encrypted_path, ’wb’) as f:
            f.write(encrypted_data)

        # Delete original
        os.remove(file_path)

        return encrypted_path

    def decrypt_file(self, encrypted_path: str) -> bytes:
        """Decrypt␣file␣for␣processing"""
        with open(encrypted_path, ’rb’) as f:
            encrypted_data = f.read()

        # Decrypt
        decrypted_data = self.cipher.decrypt(encrypted_data)

        return decrypted_data

# Usage in resume upload
def handle_resume_upload(uploaded_file, user):
    # Save temporarily
    temp_path = save_upload(uploaded_file)

    # Encrypt
    storage = SecureFileStorage()
    encrypted_path = storage.encrypt_file(temp_path)

    # Store reference
    resume = Resume.objects.create(
        user=user,
        encrypted_file_path=encrypted_path
    )

    return resume
```

5.1.4 Input Validation & Sanitization

Request Validation

Listing 5.6: Input Validation

```
from rest_framework import serializers
from django.core.validators import (
    EmailValidator,
    RegexValidator
)
import bleach

class UserRegistrationSerializer(serializers.Serializer):
    email = serializers.EmailField(
        validators=[EmailValidator()],
        required=True
    )

    password = serializers.CharField(
        min_length=12,
        max_length=128,
        required=True,
        write_only=True,
        validators=[
            RegexValidator(
                regex=r'^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)(?=.*[@$!%*?&])',
                message='Password␣must␣contain␣uppercase,␣lowercase,␣digit,␣and␣special␣character'
            )
        ]
    )

    full_name = serializers.CharField(
        max_length=100,
        required=True
    )

    def validate_full_name(self, value):
        # Sanitize HTML
        clean_name = bleach.clean(
            value,
            tags=[],
            strip=True
        )

        # Check for suspicious patterns
        if '<script>' in clean_name.lower():
            raise serializers.ValidationError(
                "Invalid␣characters␣in␣name"
            )

        return clean_name

    def validate_email(self, value):
        # Check if email already exists
        if User.objects.filter(email=value).exists():
            raise serializers.ValidationError(
                "Email␣already␣registered"
            )

        return value.lower()
```

SQL Injection Prevention

Listing 5.7: Safe Database Queries

```
# BAD - Vulnerable to SQL injection
def get_user_resumes_bad(user_id):
    query = f"SELECT␣*␣FROM␣resume␣WHERE␣user_id␣=␣{user_id}"
    cursor.execute(query)

# GOOD - Using Django ORM (safe)
def get_user_resumes_good(user_id):
    return Resume.objects.filter(user_id=user_id)

# GOOD - Using parameterized queries
def get_user_resumes_raw(user_id):
    return Resume.objects.raw(
        "SELECT␣*␣FROM␣resume␣WHERE␣user_id␣=␣%s",
        [user_id]
    )

# For complex queries, use Q objects
from django.db.models import Q

def search_resumes(search_term):
    return Resume.objects.filter(
        Q(title__icontains=search_term) |
        Q(content__icontains=search_term)
    )
```

XSS Prevention

Listing 5.8: XSS Protection

```
import bleach
from django.utils.html import escape

ALLOWED_TAGS = [
    ’p’, ’br’, ’strong’, ’em’, ’u’,
    ’h1’, ’h2’, ’h3’, ’ul’, ’ol’, ’li’
]

ALLOWED_ATTRIBUTES = {
    ’*’: [’class’],
    ’a’: [’href’, ’title’],
}

def sanitize_user_input(user_input: str) -> str:
    """Remove␣potentially␣dangerous␣HTML/JS"""

    # Clean HTML
    clean_html = bleach.clean(
        user_input,
        tags=ALLOWED_TAGS,
        attributes=ALLOWED_ATTRIBUTES,
        strip=True
    )

    # Escape any remaining special chars
    safe_output = escape(clean_html)

    return safe_output

# Usage in views
class CreateResumeView(APIView):
    def post(self, request):
        title = sanitize_user_input(request.data.get(’title’))
        content = sanitize_user_input(
            request.data.get('content')
        )

        resume = Resume.objects.create(
            user=request.user,
            title=title,
            content=content
        )

        return Response({...})
```

5.1.5 Rate Limiting & DDoS Protection

API Rate Limiting

Listing 5.9: Rate Limiting Implementation

```
from rest_framework.throttling import (
    UserRateThrottle,
    AnonRateThrottle
)
from django.core.cache import cache
import time

class ResumeAnalysisThrottle(UserRateThrottle):
    """Limit␣resume␣analysis␣to␣10␣per␣hour"""
    rate = '10/hour'
    scope = 'resume_analysis'

class InterviewThrottle(UserRateThrottle):
    """Limit␣interview␣sessions␣to␣5␣per␣day"""
    rate = '5/day'
    scope = 'interview'

class CustomRateLimit:
    """Custom␣rate␣limiting␣with␣Redis"""

    def __init__(self, key_prefix, limit, period):
        self.key_prefix = key_prefix
        self.limit = limit
        self.period = period  # in seconds

    def allow_request(self, identifier):
        key = f"{self.key_prefix}:{identifier}"

        current = cache.get(key, 0)

        if current >= self.limit:
            return False

        # Increment counter
        cache.set(key, current + 1, self.period)

        return True

# Usage
class ResumeAnalysisView(APIView):
    throttle_classes = [ResumeAnalysisThrottle]

    def post(self, request):
        # Process resume analysis
        ...
```

IP-Based Protection

Listing 5.10: IP Blocking Middleware

```
from django.core.cache import cache
from django.http import HttpResponseForbidden

class IPBlacklistMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        ip = self.get_client_ip(request)

        # Check if IP is blocked
        blocked_key = f"blocked_ip:{ip}"
        if cache.get(blocked_key):
            return HttpResponseForbidden(
                "Your␣IP␣has␣been␣temporarily␣blocked"
            )

        # Check request rate
        rate_key = f"ip_rate:{ip}"
        request_count = cache.get(rate_key, 0)

        if request_count > 100:  # 100 requests per minute
            # Block IP for 1 hour
            cache.set(blocked_key, True, 3600)
            return HttpResponseForbidden("Rate␣limit␣exceeded")

        # Increment counter
        cache.set(rate_key, request_count + 1, 60)

        response = self.get_response(request)
        return response

    def get_client_ip(self, request):
        x_forwarded_for = request.META.get('HTTP_X_FORWARDED_FOR')
        if x_forwarded_for:
            ip = x_forwarded_for.split(',')[0]
        else:
            ip = request.META.get('REMOTE_ADDR')
        return ip
```

5.1.6 Privacy Compliance (GDPR)

Data Minimization

Listing 5.11: Privacy-by-Design Models

```
class User(AbstractBaseUser):
    # Only essential data
    email = models.EmailField(unique=True)
    password = models.CharField(max_length=128)
    created_at = models.DateTimeField(auto_now_add=True)

    # Privacy settings
    data_processing_consent = models.BooleanField(default=False)
    marketing_consent = models.BooleanField(default=False)

    # Anonymization tracking
    is_anonymized = models.BooleanField(default=False)
    anonymized_at = models.DateTimeField(null=True)

class DataProcessingLog(models.Model):
    """Log␣all␣data␣processing␣activities"""
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    action = models.CharField(max_length=50)
    data_type = models.CharField(max_length=50)
    purpose = models.TextField()
    timestamp = models.DateTimeField(auto_now_add=True)
    ip_address = models.GenericIPAddressField()
```

Right to Deletion

Listing 5.12: GDPR Data Deletion

```
class UserDataDeletionView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def delete(self, request):
        user = request.user

        # Log deletion request
        DataDeletionRequest.objects.create(
            user=user,
            requested_at=timezone.now()
        )

        # Start async deletion process
        delete_user_data.delay(user.id)

        return Response({
            'message': 'Your␣data␣will␣be␣deleted␣within␣30␣days',
            'deletion_id': deletion_request.id
        })

@shared_task
def delete_user_data(user_id):
    """Async␣task␣to␣delete␣all␣user␣data"""
    user = User.objects.get(id=user_id)

    # Delete resumes and files
    for resume in user.resume_set.all():
        if resume.file_path:
            os.remove(resume.file_path)
        resume.delete()

    # Delete interviews
    user.interview_set.all().delete()

    # Delete job applications
    user.jobapplication_set.all().delete()

    # Anonymize user record (keep for analytics)
    user.email = f"deleted_user_{user.id}@anonymized.com"
    user.is_anonymized = True
    user.anonymized_at = timezone.now()
    user.save()

    # Send confirmation email
    send_deletion_confirmation(user)
```

Data Export (GDPR Right to Portability)

Listing 5.13: User Data Export

```
import json
from django.http import HttpResponse

class ExportUserDataView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def get(self, request):
        user = request.user

        # Collect all user data
        user_data = {
            'personal_info': {
                'email': user.email,
                'created_at': user.created_at.isoformat(),
            },
            'profile': {
                'full_name': user.profile.full_name,
                'location': user.profile.location,
                'skills': user.profile.skills,
            },
            'resumes': [
                {
                    'title': resume.title,
                    'created_at': resume.created_at.isoformat(),
                    'ats_score': resume.analysis.ats_score,
                }
                for resume in user.resume_set.all()
            ],
            'interviews': [
                {
                    'job_role': interview.job_role,
                    'score': interview.overall_score,
                    'date': interview.started_at.isoformat(),
                }
                for interview in user.interview_set.all()
            ],
            'job_applications': [
                {
                    'job_title': app.job.title,
                    'company': app.job.company,
                    'applied_at': app.applied_at.isoformat(),
                    'status': app.status,
                }
                for app in user.jobapplication_set.all()
            ]
        }

        # Create JSON response
        response = HttpResponse(
            json.dumps(user_data, indent=2),
            content_type='application/json'
        )
        response['Content-Disposition'] = \
            'attachment;␣filename="my_data.json"'

        return response
```

5.1.7 Security Auditing & Logging

Comprehensive Audit Logging

Listing 5.14: Security Audit Logs

```
import logging

security_logger = logging.getLogger(’security’)

class SecurityAuditMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # Log authentication attempts
        if request.path == ’/api/v1/auth/login’:
            self.log_login_attempt(request)

        response = self.get_response(request)

        # Log failed authentication
        if response.status_code == 401:
            self.log_auth_failure(request, response)

        return response

    def log_login_attempt(self, request):
        security_logger.info(
            f"Login␣attempt␣-␣Email:␣{request.data.get(’email’)},
               ␣"
            f"IP:␣{self.get_client_ip(request)}"
        )

    def log_auth_failure(self, request, response):
        security_logger.warning(
            f"Authentication␣failed␣-␣Path:␣{request.path},␣"
            f"IP:␣{self.get_client_ip(request)},␣"
            f"User:␣{getattr(request.user,␣’email’,␣’Anonymous’)}
               "
        )
```

5.1.8 Vulnerability Scanning

Dependencies Security Check

Listing 5.15: Automated Security Scanning

```
# requirements-security.txt
safety
bandit
pip-audit
```

```

# CI/CD pipeline security checks
- name: Security Scan
  run: |
    # Check Python dependencies
    pip install safety
    safety check --json

    # Scan for security issues in code
    pip install bandit
    bandit -r . -f json -o bandit-report.json

    # Check for known vulnerabilities
    pip install pip-audit
    pip-audit --desc
```

Chapter 6

Database Design & Data Models

6.1 Database Design Principles

6.1.1 Database Selection: PostgreSQL

Why PostgreSQL?

- **ACID Compliance**: Ensures data integrity
- **JSON Support**: Native JSONB for flexible schemas
- **Full-Text Search**: Built-in search capabilities
- **Scalability**: Supports horizontal partitioning
- **Performance**: Advanced indexing, query optimization
- **Open Source**: No licensing costs

6.1.2 Complete Entity-Relationship Diagram

Core Entities

Listing 6.1: User & Profile Schema

```sql
-- Users table
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    is_active BOOLEAN DEFAULT TRUE,
    is_verified BOOLEAN DEFAULT FALSE,
    mfa_enabled BOOLEAN DEFAULT FALSE,
    mfa_secret VARCHAR(32),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_login TIMESTAMP
);

-- User profiles
```

```
CREATE TABLE user_profiles (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    full_name VARCHAR(200) NOT NULL,
    phone_number VARCHAR(20),
    location VARCHAR(100),
    preferred_language VARCHAR(10) DEFAULT ’en’,
    experience_years INTEGER,
    desired_role VARCHAR(100),
    bio TEXT,
    avatar_url VARCHAR(500),
    linkedin_url VARCHAR(500),
    github_url VARCHAR(500),
    portfolio_url VARCHAR(500),
    skills JSONB DEFAULT ’[]’,
    languages JSONB DEFAULT ’[]’,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_user_profiles_user_id ON user_profiles(user_id);
CREATE INDEX idx_user_profiles_skills ON user_profiles USING GIN (skills);
```

Listing 6.2: Resume Schema

```
-- Resumes
CREATE TABLE resumes (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    title VARCHAR(255) NOT NULL,
    original_filename VARCHAR(255),
    file_path VARCHAR(500),
    file_type VARCHAR(10),
    file_size INTEGER,
    version INTEGER DEFAULT 1,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Resume content (parsed data)
CREATE TABLE resume_contents (
    id SERIAL PRIMARY KEY,
    resume_id INTEGER REFERENCES resumes(id) ON DELETE CASCADE,
    raw_text TEXT,
    personal_info JSONB,
    education JSONB,
    experience JSONB,
    skills JSONB,
    certifications JSONB,
    languages JSONB,
```

```sql
    projects JSONB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Resume analysis
CREATE TABLE resume_analyses (
    id SERIAL PRIMARY KEY,
    resume_id INTEGER REFERENCES resumes(id) ON DELETE CASCADE,
    ats_score DECIMAL(5,2),
    format_score DECIMAL(5,2),
    keyword_score DECIMAL(5,2),
    structure_score DECIMAL(5,2),
    readability_score DECIMAL(5,2),
    completeness_score DECIMAL(5,2),
    strengths JSONB,
    weaknesses JSONB,
    suggestions JSONB,
    keyword_density JSONB,
    analyzed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Resume rewrites
CREATE TABLE resume_rewrites (
    id SERIAL PRIMARY KEY,
    resume_id INTEGER REFERENCES resumes(id) ON DELETE CASCADE,
    section VARCHAR(50),
    original_content TEXT,
    rewritten_content TEXT,
    status VARCHAR(20) DEFAULT 'pending',
    user_feedback VARCHAR(20),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    accepted_at TIMESTAMP
);

CREATE INDEX idx_resumes_user_id ON resumes(user_id);
CREATE INDEX idx_resume_contents_resume_id ON resume_contents(resume_id);
CREATE INDEX idx_resume_analyses_resume_id ON resume_analyses(resume_id);
```

Listing 6.3: Interview Schema

```sql
-- Interviews
CREATE TABLE interviews (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    job_role VARCHAR(100),
    difficulty VARCHAR(20),
    interview_type VARCHAR(30),
    status VARCHAR(20) DEFAULT 'in_progress',
    overall_score DECIMAL(5,2),
    duration_minutes INTEGER,
    started_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    completed_at TIMESTAMP
);

-- Interview questions
CREATE TABLE interview_questions (
    id SERIAL PRIMARY KEY,
    interview_id INTEGER REFERENCES interviews(id) ON DELETE
       CASCADE,
    question_text TEXT NOT NULL,
    question_type VARCHAR(30),
    category VARCHAR(50),
    difficulty VARCHAR(20),
    order_num INTEGER,
    model_answer TEXT,
    evaluation_criteria JSONB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Interview responses
CREATE TABLE interview_responses (
    id SERIAL PRIMARY KEY,
    question_id INTEGER REFERENCES interview_questions(id) ON
       DELETE CASCADE,
    response_text TEXT NOT NULL,
    score DECIMAL(5,2),
    content_score DECIMAL(5,2),
    communication_score DECIMAL(5,2),
    technical_score DECIMAL(5,2),
    confidence_score DECIMAL(5,2),
    feedback TEXT,
    strengths JSONB,
    improvements JSONB,
    time_taken_seconds INTEGER,
    submitted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Interview reports
CREATE TABLE interview_reports (
    id SERIAL PRIMARY KEY,
    interview_id INTEGER REFERENCES interviews(id) ON DELETE
       CASCADE,
    overall_feedback TEXT,
    strengths_summary JSONB,
    weaknesses_summary JSONB,
    recommendations JSONB,
    skill_scores JSONB,
    improvement_areas JSONB,
    generated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

```

```sql
CREATE INDEX idx_interviews_user_id ON interviews(user_id);
CREATE INDEX idx_interview_questions_interview_id ON interview_questions(interview_id);
CREATE INDEX idx_interview_responses_question_id ON interview_responses(question_id);
```

Listing 6.4: Jobs Schema

```sql
-- Jobs
CREATE TABLE jobs (
    id SERIAL PRIMARY KEY,
    external_id VARCHAR(100) UNIQUE,
    source VARCHAR(50),
    title VARCHAR(255) NOT NULL,
    company VARCHAR(200) NOT NULL,
    company_logo_url VARCHAR(500),
    location VARCHAR(200),
    country VARCHAR(100),
    is_remote BOOLEAN DEFAULT FALSE,
    employment_type VARCHAR(50),
    experience_level VARCHAR(50),
    description TEXT,
    requirements TEXT,
    responsibilities TEXT,
    benefits TEXT,
    salary_min DECIMAL(10,2),
    salary_max DECIMAL(10,2),
    salary_currency VARCHAR(10),
    required_skills JSONB,
    preferred_skills JSONB,
    education_required VARCHAR(100),
    application_url VARCHAR(500),
    is_active BOOLEAN DEFAULT TRUE,
    posted_at TIMESTAMP,
    expires_at TIMESTAMP,
    scraped_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Job applications
CREATE TABLE job_applications (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    job_id INTEGER REFERENCES jobs(id) ON DELETE CASCADE,
    resume_id INTEGER REFERENCES resumes(id) ON DELETE SET NULL,
    status VARCHAR(50) DEFAULT 'pending',
    match_score DECIMAL(5,2),
    cover_letter TEXT,
    applied_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(user_id, job_id)
);
```

```sql

-- Saved jobs
CREATE TABLE saved_jobs (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    job_id INTEGER REFERENCES jobs(id) ON DELETE CASCADE,
    notes TEXT,
    saved_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(user_id, job_id)
);

CREATE INDEX idx_jobs_location ON jobs(location);
CREATE INDEX idx_jobs_company ON jobs(company);
CREATE INDEX idx_jobs_skills ON jobs USING GIN (required_skills);
CREATE INDEX idx_job_applications_user_id ON job_applications(user_id);
CREATE INDEX idx_job_applications_job_id ON job_applications(job_id);
CREATE INDEX idx_saved_jobs_user_id ON saved_jobs(user_id);
```

Listing 6.5: Footprint & Career Insights Schema

```sql
-- Digital footprints
CREATE TABLE footprints (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    platform VARCHAR(50),
    profile_url VARCHAR(500),
    profile_data JSONB,
    score DECIMAL(5,2),
    metrics JSONB,
    suggestions JSONB,
    last_scanned_at TIMESTAMP,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(user_id, platform)
);

-- Career insights
CREATE TABLE career_insights (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) ON DELETE CASCADE,
    insight_type VARCHAR(50),
    title VARCHAR(255),
    content TEXT,
    priority VARCHAR(20),
    category VARCHAR(50),
    action_items JSONB,
    is_read BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    read_at TIMESTAMP
);
```

```

CREATE INDEX idx_footprints_user_id ON footprints(user_id);
CREATE INDEX idx_career_insights_user_id ON career_insights(user_id);
CREATE INDEX idx_career_insights_priority ON career_insights(priority);
```

6.1.3 Django Models Implementation

Listing 6.6: Complete Django Models

```python
from django.db import models
from django.contrib.auth.models import AbstractBaseUser, BaseUserManager
from django.contrib.postgres.fields import ArrayField
from django.utils import timezone

class UserManager(BaseUserManager):
    def create_user(self, email, password=None, **extra_fields):
        if not email:
            raise ValueError('Email␣is␣required')

        email = self.normalize_email(email)
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)
        return self.create_user(email, password, **extra_fields)

class User(AbstractBaseUser):
    email = models.EmailField(unique=True)
    is_active = models.BooleanField(default=True)
    is_verified = models.BooleanField(default=False)
    is_staff = models.BooleanField(default=False)
    is_superuser = models.BooleanField(default=False)
    mfa_enabled = models.BooleanField(default=False)
    mfa_secret = models.CharField(max_length=32, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    last_login = models.DateTimeField(null=True, blank=True)

    objects = UserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = []

    class Meta:
        db_table = ’users’
        ordering = [’-created_at’]

class UserProfile(models.Model):
    user = models.OneToOneField(
        User,
        on_delete=models.CASCADE,
        related_name=’profile’
    )
    full_name = models.CharField(max_length=200)
    phone_number = models.CharField(max_length=20, blank=True)
    location = models.CharField(max_length=100, blank=True)
    preferred_language = models.CharField(
        max_length=10,
        default=’en’
    )
    experience_years = models.IntegerField(null=True, blank=True)
    desired_role = models.CharField(max_length=100, blank=True)
    bio = models.TextField(blank=True)
    avatar_url = models.URLField(max_length=500, blank=True)
    linkedin_url = models.URLField(max_length=500, blank=True)
    github_url = models.URLField(max_length=500, blank=True)
    portfolio_url = models.URLField(max_length=500, blank=True)
    skills = models.JSONField(default=list)
    languages = models.JSONField(default=list)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        db_table = ’user_profiles’

    def __str__(self):
        return f"{self.full_name}␣({self.user.email})"

class Resume(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    title = models.CharField(max_length=255)
    original_filename = models.CharField(max_length=255, blank=True)
    file_path = models.CharField(max_length=500, blank=True)
    file_type = models.CharField(max_length=10, blank=True)
    file_size = models.IntegerField(null=True)
    version = models.IntegerField(default=1)
    is_active = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        db_table = ’resumes’
        ordering = [’-created_at’]

    def __str__(self):
        return f"{self.title}␣-␣{self.user.email}"

# Additional models (ResumeContent, ResumeAnalysis, etc.)
# Similar implementations following the SQL schema above
```

6.1.4 Database Optimization

Indexing Strategy

Listing 6.7: Custom Database Indexes

```
class Resume(models.Model):
    # ... fields ...

    class Meta:
        db_table = 'resumes'
        indexes = [
            models.Index(fields=['user_id']),
            models.Index(fields=['created_at']),
            models.Index(fields=['is_active', 'user_id']),
        ]

class Job(models.Model):
    # ... fields ...

    class Meta:
        db_table = 'jobs'
        indexes = [
            models.Index(fields=['location']),
            models.Index(fields=['company']),
            models.Index(fields=['is_active', 'posted_at']),
            # GIN index for JSONB field (defined in migration)
        ]
```

Query Optimization

Listing 6.8: Efficient Database Queries

```
from django.db.models import Prefetch, Count, Q

# BAD - N+1 query problem
resumes = Resume.objects.filter(user=user)
for resume in resumes:
    analysis = resume.analysis  # Additional query per resume!
    print(analysis.ats_score)

# GOOD - Use select_related for ForeignKey/OneToOne
resumes = Resume.objects.filter(
    user=user
).select_related('analysis')

for resume in resumes:
    print(resume.analysis.ats_score)  # No additional query

# GOOD - Use prefetch_related for reverse FKs and M2M
users = User.objects.prefetch_related(
    'resume_set',
    'interview_set'
).all()

# Complex prefetch with filtering
users = User.objects.prefetch_related(
    Prefetch(
        'resume_set',
        queryset=Resume.objects.filter(
            is_active=True
        ).select_related('analysis')
    )
)

# Annotate with aggregations
users_with_stats = User.objects.annotate(
    resume_count=Count('resume'),
    interview_count=Count('interview'),
    application_count=Count('jobapplication')
)
```

Database Connection Pooling

Listing 6.9: Connection Pooling Configuration

```
# settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'utopiahire_db',
        'USER': 'postgres_user',
        'PASSWORD': 'secure_password',
        'HOST': 'localhost',
        'PORT': '5432',
        'CONN_MAX_AGE': 600,  # Connection pooling
        'OPTIONS': {
            'connect_timeout': 10,
            'options': '-c␣statement_timeout=30000'  # 30 seconds
        }
    }
}

# For production with pgBouncer
```

```
DATABASES = {
    'default': {
        # ... same as above ...
        'HOST': 'pgbouncer_host',
        'PORT': '6432',  # pgBouncer port
    }
}
```

6.1.5 Data Migrations & Versioning

Safe Migration Strategy

Listing 6.10: Custom Migration Example

```
# migrations/0015_add_skills_gin_index.py
from django.contrib.postgres.operations import BtreeGinExtension
from django.db import migrations, models

class Migration(migrations.Migration):
    dependencies = [
        ('users', '0014_previous_migration'),
    ]

    operations = [
        # Enable btree_gin extension
        BtreeGinExtension(),

        # Add GIN index for JSONB skills field
        migrations.RunSQL(
            sql="""
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣CREATE␣INDEX␣idx_user_profiles_skills
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ON␣user_profiles␣USING␣GIN␣(skills);
␣␣␣␣␣␣␣␣␣␣␣␣""",
            reverse_sql="""
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣DROP␣INDEX␣IF␣EXISTS␣idx_user_profiles_skills;
␣␣␣␣␣␣␣␣␣␣␣␣"""
        ),
    ]
```

6.1.6 Data Backup & Recovery

Backup Strategy

Listing 6.11: Database Backup Script

```
#!/bin/bash
# backup_database.sh

BACKUP_DIR="/var/backups/utopiahire"
TIMESTAMP=$(date +"%Y%m%d_%H%M%S")
```

```
BACKUP_FILE="$BACKUP_DIR/db_backup_$TIMESTAMP.sql"

# Create backup directory if not exists
mkdir -p $BACKUP_DIR

# Perform backup
pg_dump -h localhost -U postgres_user utopiahire_db >
   $BACKUP_FILE

# Compress backup
gzip $BACKUP_FILE

# Delete backups older than 30 days
find $BACKUP_DIR -name "*.gz" -mtime +30 -delete

# Upload to cloud storage (optional)
aws s3 cp "$BACKUP_FILE.gz" s3://utopiahire-backups/

echo "Backup␣completed:␣$BACKUP_FILE.gz"
```

6.1.7 Database Seeding for Development

Listing 6.12: Data Seeding

```
# management/commands/seed_database.py
from django.core.management.base import BaseCommand
from faker import Faker
import random

class Command(BaseCommand):
    help = 'Seed␣database␣with␣fake␣data'

    def handle(self, *args, **kwargs):
        fake = Faker()

        # Create users
        for i in range(50):
            user = User.objects.create_user(
                email=fake.email(),
                password='testpass123'
            )

            # Create profile
            UserProfile.objects.create(
                user=user,
                full_name=fake.name(),
                location=fake.city(),
                skills=[
                    fake.job() for _ in range(random.randint(3,
                       8))
                ],
                experience_years=random.randint(0, 15)
            )

            # Create resumes
            for _ in range(random.randint(1, 3)):
                Resume.objects.create(
                    user=user,
                    title=f"Resume␣{fake.job()}",
                    version=1
                )

        self.stdout.write(
            self.style.SUCCESS('Successfully␣seeded␣database')
        )
```

Chapter 7

API Design & Integration

7.1 RESTful API Design

7.1.1 API Architecture Overview

API Design Principles

- **RESTful**: Resource-based URLs, HTTP methods
- **Versioned**: /api/v1/ for backward compatibility
- **Consistent**: Standard response format
- **Documented**: OpenAPI/Swagger documentation
- **Secure**: JWT authentication, rate limiting

7.1.2 API Endpoints Reference

Authentication Endpoints

Listing 7.1: Authentication API

```
POST   /api/v1/auth/register
Request: {
  "email": "user@example.com",
  "password": "SecurePass123!",
  "full_name": "John Doe"
}
Response: {
  "user": {...},
  "tokens": {
    "access": "jwt_access_token",
    "refresh": "jwt_refresh_token"
  }
}

POST   /api/v1/auth/login
POST   /api/v1/auth/logout
POST   /api/v1/auth/refresh
POST   /api/v1/auth/verify-email
POST   /api/v1/auth/forgot-password
POST   /api/v1/auth/reset-password
GET    /api/v1/auth/mfa/setup
POST   /api/v1/auth/mfa/verify
```

Resume Endpoints

Listing 7.2: Resume API

```
# List/Create resumes
GET    /api/v1/resumes
POST   /api/v1/resumes/upload

# Resume operations
GET    /api/v1/resumes/:id
PUT    /api/v1/resumes/:id
DELETE /api/v1/resumes/:id

# Analysis
POST   /api/v1/resumes/:id/analyze
GET    /api/v1/resumes/:id/analysis

# Rewriting
POST   /api/v1/resumes/:id/rewrite
GET    /api/v1/resumes/:id/rewrites
PUT    /api/v1/resumes/:id/rewrites/:rewrite_id/accept

# Export
GET    /api/v1/resumes/:id/download
GET    /api/v1/resumes/:id/export?format=pdf
```

Interview Endpoints

Listing 7.3: Interview API

```
# Interview management
GET    /api/v1/interviews
POST   /api/v1/interviews/start
GET    /api/v1/interviews/:id
DELETE /api/v1/interviews/:id

# Interview process
GET    /api/v1/interviews/:id/questions
POST   /api/v1/interviews/:id/answer
POST   /api/v1/interviews/:id/complete

# Results
GET    /api/v1/interviews/:id/report
GET    /api/v1/interviews/:id/feedback
```

Jobs Endpoints

Listing 7.4: Jobs API

```
# Job discovery
GET    /api/v1/jobs
GET    /api/v1/jobs/:id
GET    /api/v1/jobs/search?q=developer&location=Tunisia

# Recommendations
GET    /api/v1/jobs/recommendations
POST   /api/v1/jobs/match

# Applications
POST   /api/v1/jobs/:id/apply
GET    /api/v1/applications
PUT    /api/v1/applications/:id

# Saved jobs
POST   /api/v1/jobs/:id/save
GET    /api/v1/jobs/saved
DELETE /api/v1/jobs/:id/unsave
```

7.1.3 Standard Response Format

Listing 7.5: Success Response

```
{
  "success": true,
  "data": {
    "id": 123,
    "title": "Resume Title",
    "ats_score": 85.5
  },
  "meta": {
    "timestamp": "2025-10-20T10:30:00Z",
    "version": "v1"
  }
}
```

Listing 7.6: Error Response

```
{
  "success": false,
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid input data",
    "details": [
      {
        "field": "email",
        "message": "Email already exists"
      }
    ]
  },
  "meta": {
    "timestamp": "2025-10-20T10:30:00Z",
    "request_id": "abc123"
  }
}
```

7.1.4 API Documentation with Swagger

Listing 7.7: Swagger Setup

```
# settings.py
INSTALLED_APPS = [
    # ...
    ’drf_yasg’,
]

# urls.py
from rest_framework import permissions
from drf_yasg.views import get_schema_view
from drf_yasg import openapi

schema_view = get_schema_view(
    openapi.Info(
        title="UtopiaHire␣API",
        default_version=’v1’,
        description="AI␣Career␣Architect␣API␣Documentation",
        terms_of_service="https://utopiahire.com/terms/",
        contact=openapi.Contact(email="api@utopiahire.com"),
        license=openapi.License(name="MIT␣License"),
    ),
    public=True,
    permission_classes=[permissions.AllowAny],
)

urlpatterns = [
    path(’swagger/’, schema_view.with_ui(’swagger’)),
    path(’redoc/’, schema_view.with_ui(’redoc’)),
]
```

7.1.5 Pagination

Listing 7.8: Cursor Pagination

```
# settings.py
REST_FRAMEWORK = {
    ’DEFAULT_PAGINATION_CLASS’:
        ’rest_framework.pagination.CursorPagination’,
    ’PAGE_SIZE’: 20
}

# Custom paginator
from rest_framework.pagination import CursorPagination

class StandardResultsSetPagination(CursorPagination):
    page_size = 20
    page_size_query_param = 'page_size'
    max_page_size = 100
    ordering = '-created_at'
```

7.1.6 API Versioning

Listing 7.9: URL Path Versioning

```
# urls.py
urlpatterns = [
    path('api/v1/', include('apps.api.v1.urls')),
    path('api/v2/', include('apps.api.v2.urls')),  # Future
]

# v1/urls.py
urlpatterns = [
    path('auth/', include('apps.users.urls')),
    path('resumes/', include('apps.resume.urls')),
    path('interviews/', include('apps.interview.urls')),
    path('jobs/', include('apps.jobs.urls')),
]
```

7.1.7 API Testing

Listing 7.10: API Test Example

```
from rest_framework.test import APITestCase
from rest_framework import status

class ResumeAPITestCase(APITestCase):
    def setUp(self):
        self.user = User.objects.create_user(
            email='test@example.com',
            password='testpass123'
        )
        self.client.force_authenticate(user=self.user)

    def test_upload_resume(self):
        with open('test_resume.pdf', 'rb') as resume_file:
            response = self.client.post(
                '/api/v1/resumes/upload',
                {'file': resume_file, 'title': 'My␣Resume'},
                format='multipart'
            )

        self.assertEqual(response.status_code, status.HTTP_201_CREATED)
        self.assertIn('id', response.data)

    def test_analyze_resume(self):
        resume = Resume.objects.create(
            user=self.user,
            title='Test␣Resume'
        )

        response = self.client.post(
            f'/api/v1/resumes/{resume.id}/analyze'
        )

        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertIn('ats_score', response.data)
```

7.2 Third-Party Integrations

7.2.1 OpenAI Integration

Listing 7.11: OpenAI Service

```
import openai
from django.conf import settings

class OpenAIService:
    def __init__(self):
        openai.api_key = settings.OPENAI_API_KEY

    def rewrite_resume_section(self, section_content, role):
        prompt = f"""
␣␣␣␣␣␣␣␣Rewrite␣this␣resume␣section␣for␣a␣{role}␣position.
␣␣␣␣␣␣␣␣Make␣it␣more␣impactful␣and␣ATS-friendly.

␣␣␣␣␣␣␣␣Original␣content:
␣␣␣␣␣␣␣␣{section_content}

␣␣␣␣␣␣␣␣Rewritten␣version:
␣␣␣␣␣␣␣␣"""

        response = openai.ChatCompletion.create(
            model="gpt-3.5-turbo",
            messages=[
                {"role": "system", "content": "You␣are␣an␣expert␣resume␣writer."},
                {"role": "user", "content": prompt}
            ],
            temperature=0.7,
            max_tokens=500
        )

        return response.choices[0].message.content

    def generate_interview_feedback(self, question, answer):
        prompt = f"""
␣␣␣␣␣␣␣␣Evaluate␣this␣interview␣answer␣and␣provide␣constructive␣feedback.

␣␣␣␣␣␣␣␣Question:␣{question}
␣␣␣␣␣␣␣␣Answer:␣{answer}

␣␣␣␣␣␣␣␣Provide:
␣␣␣␣␣␣␣␣1.␣Score␣(0-100)
␣␣␣␣␣␣␣␣2.␣Strengths
␣␣␣␣␣␣␣␣3.␣Areas␣for␣improvement
␣␣␣␣␣␣␣␣4.␣Suggestions
␣␣␣␣␣␣␣␣"""

        response = openai.ChatCompletion.create(
            model="gpt-4",
            messages=[
                {"role": "system", "content": "You␣are␣an␣expert␣interview␣coach."},
                {"role": "user", "content": prompt}
            ],
            temperature=0.5
        )

        return self.parse_feedback(response.choices[0].message.content)
```

7.2.2 LinkedIn API Integration

Listing 7.12: LinkedIn Scanner

```
import requests

class LinkedInScanner:
    def __init__(self, access_token):
        self.access_token = access_token
        self.base_url = "https://api.linkedin.com/v2"

    def get_profile(self):
        headers = {
            'Authorization': f'Bearer␣{self.access_token}'
        }

        response = requests.get(
            f"{self.base_url}/me",
            headers=headers
        )

        return response.json()

    def calculate_profile_score(self, profile_data):
        score = 0

        # Profile completeness
        if profile_data.get('summary'):
            score += 20
        if profile_data.get('experience'):
            score += 30
        if profile_data.get('education'):
            score += 20
        if profile_data.get('skills'):
            score += 20
        if profile_data.get('certifications'):
            score += 10

        return score
```

7.2.3 GitHub API Integration

Listing 7.13: GitHub Scanner

```
from github import Github

class GitHubScanner:
    def __init__(self, access_token=None):
        self.github = Github(access_token)

    def analyze_user(self, username):
        user = self.github.get_user(username)
        repos = user.get_repos()

        total_stars = sum(repo.stargazers_count for repo in repos
           )
        total_forks = sum(repo.forks_count for repo in repos)

        # Get contribution activity
        contributions = self.get_contribution_count(user)

        score = self.calculate_github_score(
            repos.totalCount,
            total_stars,
            total_forks,
            contributions
        )

        return {
            ’username’: username,
            ’public_repos’: repos.totalCount,
            ’total_stars’: total_stars,
            ’total_forks’: total_forks,
            ’contributions’: contributions,
            ’score’: score
        }

    def calculate_github_score(self, repos, stars, forks, contributions):
        score = 0
        score += min(repos * 5, 25)
        score += min(stars * 2, 25)
        score += min(forks * 3, 20)
        score += min(contributions / 100, 30)
        return min(score, 100)
```

7.2.4 Email Service

Listing 7.14: Email Notifications

```
from django.core.mail import send_mail
from django.template.loader import render_to_string

class EmailService:
    @staticmethod
    def send_verification_email(user):
        subject = ’Verify␣your␣UtopiaHire␣account’
        html_message = render_to_string(
            ’emails/verification.html’,
            {’user’: user, ’token’: user.verification_token}
        )

        send_mail(
            subject,
            ’’,
            ’noreply@utopiahire.com’,
            [user.email],
            html_message=html_message
        )

    @staticmethod
    def send_resume_analysis_complete(user, resume):
        subject = ’Your␣resume␣analysis␣is␣ready’
        html_message = render_to_string(
            ’emails/resume_analysis.html’,
            {
                ’user’: user,
                ’resume’: resume,
                'ats_score': resume.analysis.ats_score
            }
        )

        send_mail(
            subject,
            '',
            'noreply@utopiahire.com',
            [user.email],
            html_message=html_message
        )
```

7.2.5 Payment Integration (Future)

Listing 7.15: Stripe Integration

```
import stripe
from django.conf import settings

stripe.api_key = settings.STRIPE_SECRET_KEY

class PaymentService:
    @staticmethod
    def create_checkout_session(user, plan):
        session = stripe.checkout.Session.create(
            customer_email=user.email,
            payment_method_types=['card'],
            line_items=[{
                'price': plan.stripe_price_id,
                'quantity': 1,
            }],
            mode='subscription',
            success_url=settings.PAYMENT_SUCCESS_URL,
            cancel_url=settings.PAYMENT_CANCEL_URL,
        )

        return session.url
```

Chapter 8

User Interface & Experience

8.1 UI/UX Design Principles

8.1.1 Design Philosophy

- **User-Centric**: Simple, intuitive interfaces
- **Accessible**: WCAG 2.1 AA compliance
- **Responsive**: Mobile-first design
- **Fast**: Performance optimized
- **Inclusive**: Multilingual support

8.1.2 Key User Journeys

Journey 1: Resume Optimization

1. **Landing**: User sees value proposition
2. **Sign Up**: Quick registration (email + password)
3. **Upload**: Drag-and-drop resume upload
4. **Processing**: Progress indicator (30-60s)
5. **Results**: ATS score + visual breakdown
6. **Review**: Detailed suggestions with examples
7. **Action**: Accept/reject improvements
8. **Download**: Export optimized resume

Journey 2: Interview Practice

1. **Setup**: Choose role, difficulty
2. **Start**: View first question
3. **Respond**: Type/speak answer (timed)
4. **Feedback**: Immediate score + tips
5. **Continue**: Next question
6. **Complete**: Overall performance report
7. **Improve**: Practice recommended areas

8.1.3 Color Palette & Branding

Color	Hex	Usage
Primary Blue	#2563EB	Buttons, links, accents
Success Green	#10B981	Positive scores, success messages
Warning Orange	#F59E0B	Medium scores, warnings
Error Red	#EF4444	Low scores, errors
Neutral Gray	#6B7280	Text, borders
Background	#F9FAFB	Page background

Table 8.1: UtopiaHire Color System

8.1.4 Typography

- **Headings**: Inter (Google Fonts) - Bold
- **Body**: Inter - Regular
- **Code/Data**: JetBrains Mono
- **Sizes**:
  - H1: 36px
  - H2: 30px
  - H3: 24px
  - Body: 16px
  - Small: 14px

8.1.5 Component Library

Key Components

- **ScoreCard**: Visual display of ATS/interview scores
- **ProgressBar**: Analysis/upload progress
- **JobCard**: Job listing with match score
- **SuggestionPanel**: Improvement recommendations
- **ComparisonView**: Side-by-side original vs improved
- **FeedbackBadge**: Categorized feedback items
- **DashboardWidget**: Metric cards on dashboard

8.1.6 Accessibility

Listing 8.1: Accessible Component Example

```
function ScoreCard({ score, title, description }) {
  const getScoreColor = (score) => {
    if (score >= 80) return ’text-green-600’;
    if (score >= 60) return ’text-orange-500’;
    return ’text-red-600’;
  };

  return (
    <div
      className="score-card"
      role="region"
      aria-label={‘${title} score card‘}
    >
      <h3 id="score-title">{title}</h3>
      <div
        className={‘score-value ${getScoreColor(score)}‘}
        aria-describedby="score-desc"
        aria-label={‘Score: ${score} out of 100‘}
      >
        {score}/100
      </div>
      <p id="score-desc" className="text-sm text-gray-600">
        {description}
      </p>
    </div>
  );
}
```

8.1.7 Mobile Responsiveness

Listing 8.2: Responsive Layout

```
function ResumeAnalysisView() {
  return (
    <div className="container mx-auto px-4">
      {/* Mobile: Stack vertically */}
      <div className="grid grid-cols-1 md:grid-cols-2 gap-6">
        <div className="score-section">
          <ScoreCard score={85} title="ATS Score" />
        </div>

        <div className="suggestions-section">
          <SuggestionPanel suggestions={suggestions} />
        </div>
      </div>

      {/* Mobile: Full width, Desktop: Side-by-side */}
      <div className="mt-8 grid grid-cols-1 lg:grid-cols-2 gap-6">
        <ResumePreview type="original" />
        <ResumePreview type="improved" />
      </div>
    </div>
  );
}
```

8.1.8 Loading States

Listing 8.3: Loading Skeletons

```
function ResumeSkeleton() {
  return (
    <div className="animate-pulse">
      <div className="h-8 bg-gray-200 rounded w-1/3 mb-4"></div>
      <div className="h-4 bg-gray-200 rounded w-full mb-2"></div>
      <div className="h-4 bg-gray-200 rounded w-5/6 mb-2"></div>
      <div className="h-4 bg-gray-200 rounded w-4/6"></div>
    </div>
  );
}
```

8.1.9 Error Handling

Listing 8.4: User-Friendly Error Messages

```
function ErrorBoundary({ error }) {
  const errorMessages = {
    ’UPLOAD_FAILED’: {
      title: 'Upload Failed',
      message: 'We couldn\'t upload your resume. Please try again
         .',
      action: 'Retry Upload'
    },
    'ANALYSIS_ERROR': {
      title: 'Analysis Error',
      message: 'Something went wrong while analyzing your resume
         .',
      action: 'Try Again'
    },
    'NETWORK_ERROR': {
      title: 'Connection Error',
      message: 'Please check your internet connection.',
      action: 'Reload'
    }
  };

  const errorInfo = errorMessages[error.code] || {
    title: 'Something went wrong',
    message: 'Please try again later.',
    action: 'Go Back'
  };

  return (
    <div className="error-container text-center p-8">
      <AlertCircle className="w-16 h-16 text-red-500 mx-auto mb
         -4" />
      <h2 className="text-2xl font-bold mb-2">{errorInfo.title}</
         h2>
      <p className="text-gray-600 mb-6">{errorInfo.message}</p>
      <button className="btn-primary" onClick={handleAction}>
        {errorInfo.action}
      </button>
    </div>
  );
}
```

8.1.10 Internationalization (i18n)

Listing 8.5: Multilingual Support

```
import { useTranslation } from 'react-i18next';

function WelcomeSection() {
  const { t, i18n } = useTranslation();

  return (
    <div>
      <h1>{t('welcome.title')}</h1>
      <p>{t('welcome.description')}</p>

      <select
        value={i18n.language}
        onChange={(e) => i18n.changeLanguage(e.target.value)}
      >
        <option value="en">English</option>
        <option value="fr">Fran ais</option>
        <option value="ar">              </option>
      </select>
    </div>
  );
}

// locales/en.json
{
  "welcome": {
    "title": "Welcome to UtopiaHire",
    "description": "Your AI-powered career assistant"
  },
  "resume": {
    "upload": "Upload Resume",
    "analyze": "Analyze Resume",
    "score": "ATS Score"
  }
}
```

Chapter 9

Testing Strategy

9.1 Testing Strategy Overview

9.1.1 Testing Pyramid

- **Unit Tests (70%)**: Individual functions, components
- **Integration Tests (20%)**: Module interactions
- **E2E Tests (10%)**: Complete user workflows

9.1.2 Backend Testing

Unit Tests with pytest

Listing 9.1: Model Unit Tests

```
import pytest
from apps.users.models import User, UserProfile

@pytest.mark.django_db
class TestUserModel:
    def test_create_user(self):
        user = User.objects.create_user(
            email='test@example.com',
            password='TestPass123!'
        )
        assert user.email == 'test@example.com'
        assert user.check_password('TestPass123!')
        assert not user.is_staff

    def test_create_superuser(self):
        admin = User.objects.create_superuser(
            email='admin@example.com',
            password='AdminPass123!'
        )
        assert admin.is_staff
        assert admin.is_superuser

    def test_enable_mfa(self):
        user = User.objects.create_user(
            email=’test@example.com’,
            password=’pass’
        )
        totp_uri = user.enable_mfa()

        assert user.is_mfa_enabled
        assert user.mfa_secret is not None
        assert ’otpauth://’ in totp_uri
```

Listing 9.2: Service Layer Tests

```
import pytest
from unittest.mock import Mock, patch
from apps.resume.services.analyzer import ResumeAnalyzer

@pytest.fixture
def resume_analyzer():
    return ResumeAnalyzer()

@pytest.fixture
def sample_resume_text():
    return """
␣␣␣␣John␣Doe
␣␣␣␣Software␣Engineer
␣␣␣␣Skills:␣Python,␣Django,␣React
␣␣␣␣Experience:␣5␣years␣at␣Tech␣Company
␣␣␣␣"""

class TestResumeAnalyzer:
    def test_extract_skills(self, resume_analyzer,
       sample_resume_text):
        skills = resume_analyzer.extract_skills(
           sample_resume_text)

        assert ’Python’ in skills
        assert ’Django’ in skills
        assert ’React’ in skills

    def test_calculate_ats_score(self, resume_analyzer):
        resume_data = {
            ’has_contact_info’: True,
            ’has_experience’: True,
            ’has_skills’: True,
            ’keyword_density’: 0.15
        }

        score = resume_analyzer.calculate_ats_score(resume_data)

        assert 0 <= score <= 100
        assert isinstance(score, float)

    @patch('apps.resume.services.analyzer.OpenAI')
    def test_rewrite_section(self, mock_openai, resume_analyzer):
        mock_openai.return_value.generate.return_value = \
            "Improved␣content"

        original = "Basic␣job␣description"
        rewritten = resume_analyzer.rewrite_section(original)

        assert rewritten == "Improved␣content"
        mock_openai.return_value.generate.assert_called_once()
```

API Integration Tests

Listing 9.3: API Integration Tests

```
import pytest
from rest_framework.test import APIClient
from rest_framework import status

@pytest.fixture
def api_client():
    return APIClient()

@pytest.fixture
def authenticated_client(api_client):
    user = User.objects.create_user(
        email='test@example.com',
        password='TestPass123!'
    )
    api_client.force_authenticate(user=user)
    return api_client, user

@pytest.mark.django_db
class TestResumeAPI:
    def test_upload_resume_unauthenticated(self, api_client):
        response = api_client.post('/api/v1/resumes/upload')
        assert response.status_code == status.HTTP_401_UNAUTHORIZED

    def test_upload_resume_success(self, authenticated_client, tmp_path):
        client, user = authenticated_client

        # Create temporary PDF file
        resume_file = tmp_path / "resume.pdf"
        resume_file.write_bytes(b"PDF␣content")

        with open(resume_file, 'rb') as f:
            response = client.post(
                '/api/v1/resumes/upload',
        {'file': f, 'title': 'My␣Resume'},
        format='multipart'
      )

    assert response.status_code == status.HTTP_201_CREATED
    assert 'id' in response.data
    assert response.data['title'] == 'My␣Resume'

  def test_analyze_resume(self, authenticated_client):
    client, user = authenticated_client

    # Create resume
    resume = Resume.objects.create(
      user=user,
      title='Test␣Resume'
    )

    response = client.post(
      f'/api/v1/resumes/{resume.id}/analyze'
    )

    assert response.status_code == status.HTTP_200_OK
    assert 'ats_score' in response.data
    assert 0 <= response.data['ats_score'] <= 100
```

9.1.3 Frontend Testing

Component Tests with Jest & React Testing Library

Listing 9.4: React Component Tests

```
import { render, screen, fireEvent } from '@testing-library/react';
import { ScoreCard } from '../components/ScoreCard';

describe('ScoreCard Component', () => {
 test('renders score correctly', () => {
  render(<ScoreCard score={85} title="ATS Score" />);

  expect(screen.getByText('85/100')).toBeInTheDocument();
  expect(screen.getByText('ATS Score')).toBeInTheDocument();
 });

 test('applies correct color for high score', () => {
  const { container } = render(
   <ScoreCard score={90} title="Score" />
  );

  const scoreElement = container.querySelector('.score-value');
  expect(scoreElement).toHaveClass('text-green-600');
 });
```

```

  test('applies correct color for low score', () => {
    const { container } = render(
      <ScoreCard score={45} title="Score" />
    );

    const scoreElement = container.querySelector('.score-value');
    expect(scoreElement).toHaveClass('text-red-600');
  });
});

describe('ResumeUploader Component', () => {
  test('handles file upload', async () => {
    const mockUpload = jest.fn();
    render(<ResumeUploader onUpload={mockUpload} />);

    const file = new File(['resume content'], 'resume.pdf', {
      type: 'application/pdf'
    });

    const input = screen.getByLabelText(/upload resume/i);
    fireEvent.change(input, { target: { files: [file] } });

    expect(mockUpload).toHaveBeenCalledWith(file);
  });

  test('shows error for invalid file type', () => {
    render(<ResumeUploader />);

    const file = new File(['content'], 'file.exe', {
      type: 'application/exe'
    });

    const input = screen.getByLabelText(/upload resume/i);
    fireEvent.change(input, { target: { files: [file] } });

    expect(screen.getByText(/invalid file type/i)).toBeInTheDocument();
  });
});
```

9.1.4 End-to-End Testing

E2E Tests with Playwright

Listing 9.5: E2E Test Example

```
import { test, expect } from '@playwright/test';

test.describe('Resume Analysis Flow', () => {
```

```
test('complete resume upload and analysis', async ({ page }) =>
   {
  // 1. Navigate to login
  await page.goto('http://localhost:3000/login');

  // 2. Login
  await page.fill('input[name="email"]', 'test@example.com');
  await page.fill('input[name="password"]', 'TestPass123!');
  await page.click('button[type="submit"]');

  // 3. Wait for redirect to dashboard
  await expect(page).toHaveURL(/.*dashboard/);

  // 4. Navigate to resume upload
  await page.click('text=Upload Resume');

  // 5. Upload file
  const fileInput = await page.locator('input[type="file"]');
  await fileInput.setInputFiles('./test-data/sample-resume.pdf
     ');

  // 6. Fill resume title
  await page.fill('input[name="title"]', 'Test Resume');

  // 7. Click upload
  await page.click('button:has-text("Upload")');

  // 8. Wait for analysis to complete
  await page.waitForSelector('.ats-score', { timeout: 60000 });

  // 9. Verify score is displayed
  const scoreText = await page.textContent('.ats-score');
  expect(scoreText).toMatch(/\d+\/100/);

  // 10. Check suggestions are shown
  const suggestions = await page.locator('.suggestion-item');
  await expect(suggestions).toHaveCount.greaterThan(0);
});

test('interview simulation flow', async ({ page }) => {
  await page.goto('http://localhost:3000/dashboard');

  // Start interview
  await page.click('text=Start Interview');

  // Select job role
  await page.selectOption('select[name="jobRole"]', 'Software
     Engineer');
  await page.selectOption('select[name="difficulty"]', 'Mid-
     level');
  await page.click('button:has-text("Begin")');
```

```

    // Answer first question
    await page.waitForSelector('.interview-question');
    await page.fill(
      'textarea[name="answer"]',
      'This is my test answer to the interview question.'
    );
    await page.click('button:has-text("Submit Answer")');

    // Wait for feedback
    await page.waitForSelector('.answer-feedback');
    const score = await page.textContent('.answer-score');
    expect(score).toMatch(/\d+/);
  });
});
```

9.1.5 Performance Testing

Listing 9.6: Load Testing with Locust

```
from locust import HttpUser, task, between

class UtopiaHireUser(HttpUser):
    wait_time = between(1, 3)

    def on_start(self):
        # Login
        response = self.client.post('/api/v1/auth/login', json={
            'email': 'test@example.com',
            'password': 'TestPass123!'
        })
        self.token = response.json()['access']
        self.client.headers.update({
            'Authorization': f'Bearer␣{self.token}'
        })

    @task(3)
    def view_dashboard(self):
        self.client.get('/api/v1/dashboard')

    @task(2)
    def list_resumes(self):
        self.client.get('/api/v1/resumes')

    @task(1)
    def analyze_resume(self):
        # Upload and analyze
        with open('test_resume.pdf', 'rb') as f:
            response = self.client.post(
                '/api/v1/resumes/upload',
                files={'file': f}
            )

        if response.status_code == 201:
            resume_id = response.json()['id']
            self.client.post(
                f'/api/v1/resumes/{resume_id}/analyze'
            )
```

9.1.6 Test Coverage

Listing 9.7: Coverage Configuration

```
# .coveragerc
[run]
source = apps/
omit =
    */migrations/*
    */tests/*
    */venv/*
    */__pycache__/*

[report]
precision = 2
show_missing = True
skip_covered = False

# Run tests with coverage
pytest --cov=apps --cov-report=html --cov-report=term

# Coverage targets
# Backend: 80%+ coverage
# Frontend: 70%+ coverage
# Critical paths: 95%+ coverage
```

9.1.7 CI/CD Test Pipeline

Listing 9.8: GitHub Actions Test Workflow

```
name: Test Suite

on: [push, pull_request]

jobs:
  backend-tests:
    runs-on: ubuntu-latest

    services:
      postgres:
        image: postgres:15
        env:
          POSTGRES_PASSWORD: postgres
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5

    steps:
      - uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: ’3.11’

      - name: Install dependencies
        run: |
          pip install -r requirements/test.txt

      - name: Run tests
        run: |
          pytest --cov=apps --cov-report=xml
        env:
          DATABASE_URL: postgres://postgres:postgres@localhost/test_db

      - name: Upload coverage
        uses: codecov/codecov-action@v3

  frontend-tests:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3

      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: ’18’

      - name: Install dependencies
        run: npm ci

      - name: Run tests
        run: npm test -- --coverage

      - name: Upload coverage
        uses: codecov/codecov-action@v3

  e2e-tests:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3

      - name: Install Playwright
        run: npx playwright install

      - name: Run E2E tests
        run: npx playwright test

      - name: Upload test results
        if: always()
        uses: actions/upload-artifact@v3
        with:
          name: playwright-report
          path: playwright-report/
```

9.1.8 Test Data Management

Listing 9.9: Test Fixtures

```
# conftest.py
import pytest
from apps.users.models import User
from apps.resume.models import Resume

@pytest.fixture
def sample_user(db):
    return User.objects.create_user(
        email='test@example.com',
        password='TestPass123!'
    )

@pytest.fixture
def sample_resume(sample_user):
    return Resume.objects.create(
        user=sample_user,
        title='Test␣Resume',
        file_path='/path/to/resume.pdf'
    )

@pytest.fixture
def sample_resume_content():
    return {
        'personal_info': {
            'name': 'John␣Doe',
            'email': 'john@example.com',
            'phone': '+1234567890'
        },
        'experience': [
            {
                'company': 'Tech␣Corp',
                'position': 'Software␣Engineer',
                'duration': '2020-2023'
            }
        ],
        'skills': ['Python', 'Django', 'React']
    }
```

Chapter 10

Deployment & DevOps

10.1 Deployment Architecture

10.1.1 Environment Strategy

Environment	Purpose	Configuration
Development	Local dev	SQLite, Debug mode
Staging	Pre-production testing	PostgreSQL, Production-like
Production	Live system	PostgreSQL, Optimized

10.1.2 Docker Configuration

Listing 10.1: Dockerfile for Backend

```
# Backend Dockerfile
FROM python:3.11-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    gcc \
    postgresql-client \
    && rm -rf /var/lib/apt/lists/*

# Install Python dependencies
COPY requirements/production.txt .
RUN pip install --no-cache-dir -r production.txt

# Copy application
COPY . .

# Collect static files
RUN python manage.py collectstatic --noinput

EXPOSE 8000
```

```

CMD ["gunicorn", "config.wsgi:application", "--bind", "0.0.0.0:8000"]
```

Listing 10.2: Docker Compose for Production

```
version: ’3.8’

services:
  db:
    image: postgres:15
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: utopiahire
      POSTGRES_PASSWORD: ${DB_PASSWORD}

  redis:
    image: redis:7-alpine
    volumes:
      - redis_data:/data

  backend:
    build: ./backend
    command: gunicorn config.wsgi:application --bind 0.0.0.0:8000
    volumes:
      - ./backend:/app
      - static_volume:/app/staticfiles
      - media_volume:/app/media
    depends_on:
      - db
      - redis
    environment:
      DATABASE_URL: postgres://postgres:${DB_PASSWORD}@db:5432/utopiahire
      REDIS_URL: redis://redis:6379

  celery:
    build: ./backend
    command: celery -A config worker -l info
    depends_on:
      - db
      - redis

  frontend:
    build: ./frontend
    ports:
      - "3000:3000"

  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
      - static_volume:/app/staticfiles
      - media_volume:/app/media
    depends_on:
      - backend
      - frontend

volumes:
  postgres_data:
  redis_data:
  static_volume:
  media_volume:
```

10.1.3 Cloud Deployment Options

Option 1: AWS

- **Compute**: ECS/Fargate for containers
- **Database**: RDS PostgreSQL
- **Storage**: S3 for resumes/files
- **CDN**: CloudFront
- **Load Balancer**: ALB
- **Cache**: ElastiCache Redis

Option 2: Google Cloud Platform

- **Compute**: Cloud Run
- **Database**: Cloud SQL
- **Storage**: Cloud Storage
- **CDN**: Cloud CDN

Option 3: Digital Ocean (Budget-Friendly)

- **Compute**: Droplets or App Platform
- **Database**: Managed PostgreSQL
- **Storage**: Spaces (S3-compatible)
- **Load Balancer**: DO Load Balancer

10.1.4 CI/CD Pipeline

Listing 10.3: Complete CI/CD Workflow

```yaml
name: Deploy to Production

on:
  push:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Run tests
        run: |
          docker-compose -f docker-compose.test.yml up --abort-on-container-exit

  build:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Build and push Docker images
        run: |
          docker build -t utopiahire/backend:${{ github.sha }} ./backend
          docker push utopiahire/backend:${{ github.sha }}

  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to production
        run: |
          # Deploy to cloud provider
          # Example: AWS ECS, GCP Cloud Run, etc.
```

10.1.5 Environment Variables

Listing 10.4: Production Environment Variables

```
# .env.production
DEBUG=False
SECRET_KEY=<generate-strong-secret>
ALLOWED_HOSTS=utopiahire.com,www.utopiahire.com

# Database
```

```
DATABASE_URL=postgres://user:pass@host:5432/utopiahire

# Redis
REDIS_URL=redis://redis:6379/0

# OpenAI
OPENAI_API_KEY=sk-xxxxxxxxxxxxx

# AWS S3
AWS_ACCESS_KEY_ID=xxxxx
AWS_SECRET_ACCESS_KEY=xxxxx
AWS_STORAGE_BUCKET_NAME=utopiahire-files

# Email
EMAIL_HOST=smtp.gmail.com
EMAIL_PORT=587
EMAIL_HOST_USER=noreply@utopiahire.com
EMAIL_HOST_PASSWORD=xxxxx

# Security
SECURE_SSL_REDIRECT=True
SESSION_COOKIE_SECURE=True
CSRF_COOKIE_SECURE=True
```

10.1.6 Monitoring & Logging

Listing 10.5: Production Logging

```
# settings/production.py
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format': '{levelname}␣{asctime}␣{module}␣{message}',
            'style': '{',
        },
    },
    'handlers': {
        'file': {
            'level': 'INFO',
            'class': 'logging.handlers.RotatingFileHandler',
            'filename': '/var/log/utopiahire/django.log',
            'maxBytes': 1024*1024*15,  # 15MB
            'backupCount': 10,
            'formatter': 'verbose',
        },
        'console': {
            'level': 'INFO',
            'class': 'logging.StreamHandler',
            'formatter': 'verbose'
        },
    },
    'loggers': {
        'django': {
            'handlers': ['file', 'console'],
            'level': 'INFO',
            'propagate': False,
        },
        'apps': {
            'handlers': ['file', 'console'],
            'level': 'INFO',
            'propagate': False,
        },
    },
}
```

10.1.7 Scaling Strategy

Horizontal Scaling

- Multiple backend instances behind load balancer
- Stateless application design
- Session storage in Redis
- Database read replicas

Vertical Scaling

- Increase server resources as needed
- Database connection pooling
- Query optimization
- Caching strategy

10.1.8 Backup & Disaster Recovery

Listing 10.6: Automated Backup Script

```
#!/bin/bash
# Daily backup script

# Database backup
pg_dump $DATABASE_URL | gzip > backup_$(date +%Y%m%d).sql.gz

# Upload to S3
aws s3 cp backup_$(date +%Y%m%d).sql.gz s3://backups/

# Backup user files
```

```
tar -czf files_$(date +%Y%m%d).tar.gz /var/media/
aws s3 cp files_$(date +%Y%m%d).tar.gz s3://backups/

# Retention: Keep last 30 days
find ./backups -mtime +30 -delete
```

Chapter 11

Project Management & Timeline

11.1 Project Timeline

11.1.1 Development Phases

Phase	Duration	Deadline	Deliverables
Phase 0	Week 1-2	Oct 25	Planning, architecture, team setup
Phase 1	Week 3-6	Nov 16	MVP, PDF report, demo video, GitHub
Phase 2	Week 7-10	Dec 21	Final prototype, presentation

Table 11.1: Challenge Timeline

11.1.2 Detailed Sprint Plan

Sprint 1 (Week 1-2): Foundation

- **Week 1**:
  - Team formation & role assignment
  - Requirements analysis
  - Architecture design
  - Technology stack finalization
  - Database schema design
  - Git repository setup
- **Week 2**:
  - Backend project scaffolding
  - Frontend project setup
  - Authentication system
  - User registration/login
  - Basic UI components
  - CI/CD pipeline setup

Sprint 2 (Week 3-4): Core Modules

- **Week 3**:
  - Resume upload & parsing
  - NLP model integration
  - Resume analysis algorithm
  - Database models completion
  - API endpoints for resume module
- **Week 4**:
  - Resume rewriting feature
  - Interview module backend
  - Question generation
  - Response evaluation
  - Frontend: Resume upload UI
  - Frontend: Analysis dashboard

Sprint 3 (Week 5-6): Integration & Polish

- **Week 5**:
  - Job matching algorithm
  - Footprint scanner
  - Career insights generation
  - Frontend: Interview simulator
  - Frontend: Job recommendations
  - Integration testing
- **Week 6**:
  - Security hardening
  - Performance optimization
  - Bug fixes
  - **Documentation (PDF report)**
  - **Demo video creation**
  - **GitHub cleanup & documentation**
  - **Submission (Nov 16)**

Sprint 4 (Week 7-8): Post-Submission Enhancement

- Address feedback from Phase 1
- UI/UX improvements
- Additional features
- Performance tuning
- Advanced analytics

Sprint 5 (Week 9-10): Final Phase

- Prototype refinement
- Presentation preparation
- Booth materials
- Final testing
- Deployment to production
- **Final submission (Dec 21)**

11.1.3 Team Roles & Responsibilities

Role	**Responsibilities**
Team Lead	Coordination, decision making, timeline management
Backend Developer 1	Resume & interview modules, API development
Backend Developer 2	Jobs module, AI/ML integration, database
Frontend Developer 1	UI components, state management, routing
Frontend Developer 2	Interview UI, job matching UI, responsive design
DevOps/Security	CI/CD, deployment, security, testing

Table 11.2: Team Structure

11.1.4 Risk Management

11.1.5 Communication Plan

- **Daily Standups**: 15-minute sync (async on Slack if needed)
- **Weekly Planning**: Sunday evenings
- **Code Reviews**: All PRs require 1 approval
- **Sprint Reviews**: End of each 2-week sprint
- **Tools**:
  - GitHub: Code repository, issues, project board

Risk	**Probability**	**Mitigation**
API rate limits (OpenAI)	High	Implement caching, use fallback models
Timeline delays	Medium	Buffer time in schedule, prioritize MVP
Integration issues	Medium	Early integration testing, clear APIs
Team member unavailability	Low	Cross-training, documentation
Security vulnerabilities	Low	Security audits, code reviews

Table 11.3: Risk Management Matrix

- Slack/Discord: Team communication
- Notion: Documentation, meeting notes
- Figma: UI/UX designs

11.1.6 Definition of Done

For a feature to be considered ”done”:

1. Code written and reviewed
2. Unit tests written (80%+ coverage)
3. Integration tests passing
4. API documented (Swagger)
5. UI implemented and responsive
6. Manual testing completed
7. Security review passed
8. Merged to main branch
9. Deployed to staging

Chapter 12

Deliverables Checklist

12.1 Phase 1 Deliverables (Nov 16, 2025)

12.1.1 1. PDF Report

Report Structure

1. **Executive Summary** (1 page)
   - Problem statement
   - Solution overview
   - Key features
   - Impact
2. **Problem Understanding** (2-3 pages)
   - Regional employment challenges
   - Target user personas
   - Market research
   - User needs analysis
3. **Technical Approach** (5-7 pages)
   - System architecture
   - Technology stack justification
   - AI/ML models used
   - Module specifications
   - Database design
4. **Security & Privacy** (2-3 pages)
   - Security measures
   - Data encryption
   - Privacy compliance (GDPR)
   - Ethical AI practices

5. **Implementation** (3-4 pages)
   - Development methodology
   - Testing strategy
   - Challenges & solutions
   - Future roadmap
6. **Results & Impact** (2 pages)
   - Performance metrics
   - User testing results
   - Expected impact
   - Scalability plan

Total: 15-20 pages, Professional format, Anonymous

12.1.2 2. GitHub Repository

Repository Structure

Listing 12.1: Required Repository Structure

```
utopiahire/
          README.md (Comprehensive, anonymous)
          ARCHITECTURE.md
          SETUP.md
          CONTRIBUTING.md
          LICENSE
          .gitignore
          docker-compose.yml
          backend/
                README.md
                Dockerfile
                requirements/
                apps/
                tests/
                docs/
          frontend/
                README.md
                Dockerfile
                src/
                tests/
                docs/
          docs/
                API.md
                DATABASE.md
                DEPLOYMENT.md
                screenshots/
          .github/
              workflows/
```

README.md Template

Listing 12.2: README.md Structure

~~~~
# UtopiaHire - AI Career Architect

> AI-powered platform for career development in Sub-Saharan Africa and MENA

## Overview
Brief description of the project and its goals.

## Features
- Resume Reviewer/Rewriter
- AI Interviewer & Profiler
- Job Matcher
- Footprint Scanner

## Technology Stack
### Backend
- Django 4.2
- PostgreSQL 15
- Redis
- Celery

### Frontend
- React 18
- Redux Toolkit
- TailwindCSS

### AI/ML
- OpenAI GPT-3.5/4
- spaCy
- Sentence-BERT

## Quick Start
‘‘‘bash
# Clone repository
git clone https://github.com/anonymous/utopiahire.git

# Start with Docker
docker-compose up

# Access application
http://localhost:3000
‘‘‘

## Documentation
- [Architecture](docs/ARCHITECTURE.md)
- [API Documentation](docs/API.md)
- [Setup Guide](SETUP.md)

~~~~

```
## Testing
‘‘‘bash
# Backend tests
pytest

# Frontend tests
npm test

# E2E tests
npx playwright test
‘‘‘

## License
MIT License

## Contact
For inquiries: [KEEP ANONYMOUS]
```

Code Quality Checklist

Clean, readable code

Consistent naming conventions

Comprehensive comments

Type hints (Python)

ESLint/Prettier (JavaScript)

No hardcoded secrets

Environment variables properly used

Git history clean (meaningful commits)

No personal information

All tests passing

12.1.3 3. Demo Video

Video Script (3-5 minutes)

1. **Introduction (30s)**
   - Hook: Problem statement
   - Solution: UtopiaHire overview
   - Value proposition
2. **Resume Optimization Demo (1 min)**
   - Upload resume

   - Show ATS analysis
   - Display suggestions
   - Accept improvements
   - Download optimized resume

3. **Interview Simulator Demo (1 min)**
   - Start interview session
   - Answer question
   - Show real-time feedback
   - Display performance report
4. **Job Matching Demo (45s)**
   - Browse job recommendations
   - Show match scores
   - Explain matching algorithm
5. **Technical Highlights (45s)**
   - Architecture overview
   - AI/ML models used
   - Security features
   - Scalability
6. **Impact & Conclusion (30s)**
   - Expected impact
   - Future roadmap
   - Call to action

Video Production Checklist

High-quality screen recording (1080p minimum)

Clear voiceover or captions

Background music (royalty-free)

Professional transitions

Anonymous (no personal info)

Uploaded to YouTube (unlisted)

Length: 3-5 minutes

Subtitle file included

Tools for Video Creation

- **Screen Recording**: OBS Studio, Loom
- **Video Editing**: DaVinci Resolve, Adobe Premiere
- **Voiceover**: Audacity, Adobe Audition
- **Music**: YouTube Audio Library, Epidemic Sound

12.1.4 Booth Presentation (Phase 1)

Booth Materials

- **Poster**: A1 size, key features, architecture diagram
- **Live Demo**: Running application on laptop
- **Handout**: One-page summary (anonymous)
- **QR Code**: Link to GitHub repository
- **Presentation Slides**: Backup slides

5-Minute Pitch Structure

1. **Problem (1 min)**: Regional employment challenges
2. **Solution (2 min)**: UtopiaHire features + live demo
3. **Technology (1 min)**: Technical approach, AI/ML, security
4. **Impact (1 min)**: Expected outcomes, scalability

12.2 Phase 2 Deliverables (Dec 21, 2025)

12.2.1 Final Presentation

- Duration: 10-15 minutes
- Format: PowerPoint/Google Slides
- Content:
  - Improvements since Phase 1
  - Performance metrics
  - User feedback integration
  - Production deployment
  - Future roadmap

12.2.2 Prototype Showcase

- Fully functional MVP
- Deployed to production
- All core features working
- Performance optimized
- Security hardened

12.3 Scoring Optimization

12.3.1 Scoring Breakdown & Strategy

Criterion	**Points**	**Our Focus**
Problem Understanding	10	Deep regional research, clear personas
Technical Approach	45	Solid architecture, proven AI, scalability
Deliverables Quality	15	Professional docs, clean code, good video
Booth Presentation	10	Engaging demo, clear communication
Security	10	Encryption, GDPR, ethical AI
Presentation Quality	20	Storytelling, visual aids, impact
Prototype Function	20	Working features, smooth UX
Bonus	4	CS member, CN members, mentoring
TOTAL	134	Aim for 120+ points

Table 12.1: Challenge Scoring Strategy

12.3.2 Bonus Points Strategy

- **CS Member (1 pt)**: Ensure at least one CS member
- **CN Members (1 pt)**: Multiple CN members if possible
- **CyberSecurity LG (1 pt)**: All members join Collabratec
- **Expert Mentoring (1 pt)**: Engage with mentors early

12.3.3 Pre-Submission Checklist

PDF Report

15-20 pages, professional format

All sections complete

Diagrams and visualizations included

Proofread (no typos)

Anonymous (no names, SB info)

PDF/A format

GitHub Repository

Comprehensive README

Clean commit history

All documentation complete

Code quality high

Tests passing

No secrets exposed

Anonymous

Public/unlisted

Demo Video

3-5 minutes length

High quality (1080p)

Clear audio

Shows all features

Highlights technical approach

Anonymous

Uploaded to YouTube

Subtitles/captions

Final Verification

All deliverables submitted before deadline

Submission confirmation received

Backup copies saved

Team members notified

Appendix A

Technology Stack Reference

A.1 Backend Technologies

A.1.1 Python & Django

- **Django**: 4.2 LTS - Web framework
- **Django REST Framework**: 3.14+ - API development
- **Celery**: 5.3+ - Async task processing
- **Gunicorn**: 21.2+ - WSGI HTTP Server

A.1.2 Databases

- **PostgreSQL**: 15+ - Primary database
- **Redis**: 7+ - Cache & message broker
- **psycopg2**: PostgreSQL adapter

A.1.3 AI/ML Libraries

- **OpenAI**: GPT-3.5/4 API client
- **spaCy**: 3.7+ - NLP processing
- **transformers**: 4.35+ - HuggingFace models
- **sentence-transformers**: Embeddings
- **scikit-learn**: 1.3+ - ML utilities
- **TensorFlow/PyTorch**: Deep learning (optional)

A.1.4 Document Processing

- **PyPDF2**: PDF parsing
- **python-docx**: DOCX parsing
- **pdfplumber**: Advanced PDF extraction
- **pytesseract**: OCR for scanned documents

A.1.5 Authentication & Security

- **djangorestframework-simplejwt**: JWT authentication
- **django-cors-headers**: CORS handling
- **cryptography**: Encryption
- **pyotp**: TOTP/MFA
- **django-ratelimit**: Rate limiting

A.1.6 Testing & Quality

- **pytest**: 7.4+ - Testing framework
- **pytest-django**: Django integration
- **pytest-cov**: Code coverage
- **black**: Code formatting
- **flake8**: Linting
- **mypy**: Type checking

A.2 Frontend Technologies

A.2.1 React Ecosystem

- **React**: 18+ - UI library
- **React Router**: 6+ - Routing
- **Redux Toolkit**: State management
- **React Hook Form**: Form handling
- **Yup**: Validation

A.2.2 UI Components & Styling

- **TailwindCSS**: 3+ - Utility-first CSS
- **Material-UI (MUI)**: Alternative component library
- **React Icons**: Icon library
- **Recharts**: Data visualization
- **Framer Motion**: Animations

A.2.3 HTTP & Data Fetching

- **Axios**: HTTP client
- **React Query**: Server state management
- **SWR**: Alternative data fetching

A.2.4 Build Tools

- **Vite**: 5+ - Build tool
- **Webpack**: Alternative bundler
- **Babel**: JavaScript compiler

A.2.5 Testing

- **Jest**: Testing framework
- **React Testing Library**: Component testing
- **Playwright**: E2E testing
- **MSW**: API mocking

A.3 DevOps & Infrastructure

A.3.1 Containerization

- **Docker**: 24+ - Containerization
- **Docker Compose**: Multi-container orchestration
- **Kubernetes**: Production orchestration (optional)

A.3.2 CI/CD

- **GitHub Actions**: CI/CD pipeline
- **GitLab CI**: Alternative
- **pre-commit**: Git hooks

A.3.3 Monitoring & Logging

- **Prometheus**: Metrics collection
- **Grafana**: Visualization
- **ELK Stack**: Logging (Elasticsearch, Logstash, Kibana)
- **Sentry**: Error tracking

A.3.4 Cloud Providers

- **AWS**: EC2, RDS, S3, CloudFront
- **Google Cloud**: Cloud Run, Cloud SQL
- **Digital Ocean**: Droplets, App Platform
- **Vercel/Netlify**: Frontend hosting

A.4 Third-Party Services

A.4.1 AI & ML

- **OpenAI API**: GPT models
- **HuggingFace**: Pre-trained models
- **Google Cloud AI**: Alternative ML services

A.4.2 External APIs

- **LinkedIn API**: Profile data
- **GitHub API**: Repository analysis
- **StackOverflow API**: Reputation data

A.4.3 Communication

- **SendGrid/Mailgun**: Email service
- **Twilio**: SMS (optional)
- **Pusher**: Real-time notifications (optional)

A.4.4 Analytics

- **Google Analytics**: User analytics
- **Mixpanel**: Product analytics
- **Hotjar**: User behavior (optional)

A.5 Development Tools

A.5.1 IDE & Editors

- **VS Code**: Recommended IDE
- **PyCharm**: Python IDE
- **WebStorm**: JavaScript IDE

A.5.2 Collaboration

- **GitHub**: Version control
- **Slack/Discord**: Team communication
- **Notion**: Documentation
- **Figma**: UI/UX design
- **Miro**: Diagramming

A.5.3 API Development

- **Postman**: API testing
- **Insomnia**: Alternative API client
- **Swagger UI**: API documentation

A.6 Complete requirements.txt

Listing A.1: requirements/base.txt

```
# Core Django
Django==4.2.7
djangorestframework==3.14.0
django-cors-headers==4.3.0
django-filter==23.3

# Database
psycopg2-binary==2.9.9
dj-database-url==2.1.0

# Authentication
djangorestframework-simplejwt==5.3.0
pyotp==2.9.0

# Celery
celery==5.3.4
redis==5.0.1
django-celery-beat==2.5.0
```

```

# File Processing
PyPDF2==3.0.1
python-docx==1.1.0
pdfplumber==0.10.3
pytesseract==0.3.10
Pillow==10.1.0

# AI/ML
openai==1.3.5
spacy==3.7.2
transformers==4.35.2
sentence-transformers==2.2.2
scikit-learn==1.3.2
torch==2.1.0

# NLP
nltk==3.8.1
textblob==0.17.1
langdetect==1.0.9

# Utilities
python-decouple==3.8
requests==2.31.0
bleach==6.1.0
cryptography==41.0.7

# Monitoring
sentry-sdk==1.38.0
```

Listing A.2: requirements/development.txt

```
-r base.txt

# Testing
pytest==7.4.3
pytest-django==4.7.0
pytest-cov==4.1.0
factory-boy==3.3.0
faker==20.1.0

# Code Quality
black==23.11.0
flake8==6.1.0
mypy==1.7.1
isort==5.12.0
pylint==3.0.2

# Development
django-debug-toolbar==4.2.0
ipython==8.18.1
django-extensions==3.2.3
```

Listing A.3: requirements/production.txt

```
-r base.txt

# Production server
gunicorn==21.2.0
whitenoise==6.6.0

# Monitoring
prometheus-client==0.19.0
```

A.7 Complete package.json

Listing A.4: package.json

```
{
  "name": "utopiahire-frontend",
  "version": "1.0.0",
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-router-dom": "^6.20.0",
    "@reduxjs/toolkit": "^1.9.7",
    "react-redux": "^8.1.3",
    "axios": "^1.6.2",
    "@tanstack/react-query": "^5.8.4",
    "react-hook-form": "^7.48.2",
    "yup": "^1.3.3",
    "tailwindcss": "^3.3.5",
    "@headlessui/react": "^1.7.17",
    "@heroicons/react": "^2.0.18",
    "recharts": "^2.10.3",
    "framer-motion": "^10.16.5",
    "react-i18next": "^13.5.0",
    "i18next": "^23.7.6",
    "date-fns": "^2.30.0",
    "clsx": "^2.0.0"
  },
  "devDependencies": {
    "@vitejs/plugin-react": "^4.2.0",
    "vite": "^5.0.4",
    "@types/react": "^18.2.41",
    "@types/react-dom": "^18.2.17",
    "typescript": "^5.3.2",
    "@testing-library/react": "^14.1.2",
    "@testing-library/jest-dom": "^6.1.5",
    "@testing-library/user-event": "^14.5.1",
    "@playwright/test": "^1.40.1",
    "jest": "^29.7.0",
    "eslint": "^8.54.0",
    "prettier": "^3.1.0",
```

```json
    "autoprefixer": "^10.4.16",
    "postcss": "^8.4.32"
  },
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview",
    "test": "jest",
    "test:watch": "jest --watch",
    "test:e2e": "playwright test",
    "lint": "eslint src --ext js,jsx,ts,tsx",
    "format": "prettier --write \"src/**/*.{js,jsx,ts,tsx,json,css,md}\""
  }
}
```

Appendix B

Glossary & Acronyms

B.1 Acronyms

B.2 Key Terms

B.2.1 A

- **API Gateway**: Entry point for all API requests
- **ATS Score**: Applicant Tracking System compatibility score
- **Authentication**: Process of verifying user identity
- **Authorization**: Process of determining user permissions

B.2.2 B

- **Backend**: Server-side application logic
- **Batch Processing**: Processing multiple items together
- **BERT**: Transformer-based NLP model by Google

B.2.3 C

- **Caching**: Storing frequently accessed data for quick retrieval
- **Celery**: Distributed task queue for Python
- **Container**: Lightweight, standalone executable package
- **CRUD Operations**: Create, Read, Update, Delete

B.2.4 D

- **Django**: High-level Python web framework
- **Docker**: Platform for containerizing applications
- **Database Migration**: Version control for database schema

B.2.5 E

- **Embedding**: Dense vector representation of text
- **Encryption**: Converting data to secure format
- **Endpoint**: URL where API can be accessed

B.2.6 F

- **Frontend**: Client-side user interface
- **Footprint**: Digital presence across platforms

B.2.7 G

- **GPT**: Generative Pre-trained Transformer model
- **GDPR**: EU data protection regulation

B.2.8 H

- **Horizontal Scaling**: Adding more machines
- **HTTP**: Protocol for web communication

B.2.9 J

- **JSON**: Lightweight data interchange format
- **JWT**: Secure token format for authentication

B.2.10 L

- **Load Balancer**: Distributes traffic across servers
- **LLM**: Large Language Model (e.g., GPT-4)

B.2.11 M

- **Microservices**: Independent, loosely-coupled services
- **Middleware**: Software between OS and applications
- **Migration**: Database schema change

B.2.12 N

- **NER**: Named Entity Recognition
- **NLP**: Natural Language Processing
- **Normalization**: Standardizing data format

B.2.13 O

- **ORM**: Object-Relational Mapping
- **OAuth**: Authorization framework

B.2.14 P

- **Pagination**: Dividing data into pages
- **PostgreSQL**: Advanced open-source database
- **Prompt**: Input text for AI model

B.2.15 R

- **Redis**: In-memory data structure store
- **REST**: Architectural style for APIs
- **Refactoring**: Restructuring code without changing behavior

B.2.16 S

- **Serialization**: Converting objects to transmittable format
- **Semantic Search**: Search based on meaning, not keywords
- **STAR Method**: Behavioral interview framework

B.2.17 T

- **Tokenization**: Breaking text into tokens
- **Transformer**: Neural network architecture for NLP
- **Throttling**: Limiting request rate

B.2.18 V

- **Validation**: Checking data correctness
- **Vertical Scaling**: Adding more power to existing machine

B.2.19 W

- **Webhook**: Automated HTTP callback
- **WebSocket**: Full-duplex communication protocol

Acronym	**Definition**
AI	Artificial Intelligence
API	Application Programming Interface
ATS	Applicant Tracking System
BERT	Bidirectional Encoder Representations from Transformers
CI/CD	Continuous Integration/Continuous Deployment
CORS	Cross-Origin Resource Sharing
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DRF	Django REST Framework
E2E	End-to-End
ELK	Elasticsearch, Logstash, Kibana
GDPR	General Data Protection Regulation
GPT	Generative Pre-trained Transformer
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
JWT	JSON Web Token
LLM	Large Language Model
MENA	Middle East and North Africa
MFA	Multi-Factor Authentication
ML	Machine Learning
MVP	Minimum Viable Product
NER	Named Entity Recognition
NLP	Natural Language Processing
OCR	Optical Character Recognition
ORM	Object-Relational Mapping
RBAC	Role-Based Access Control
REST	Representational State Transfer
SBERT	Sentence-BERT
SPA	Single Page Application
SQL	Structured Query Language
SSL	Secure Sockets Layer
STAR	Situation, Task, Action, Result
TLS	Transport Layer Security
TOTP	Time-based One-Time Password
UI	User Interface
URL	Uniform Resource Locator
UX	User Experience
WCAG	Web Content Accessibility Guidelines
WSGI	Web Server Gateway Interface
XSS	Cross-Site Scripting

Table B.1: Common Acronyms

Appendix C

Resources & References

C.1 Official Documentation

C.1.1 Frameworks & Libraries

- **Django**: https://docs.djangoproject.com/
- **Django REST Framework**: https://www.django-rest-framework.org/
- **React**: https://react.dev/
- **Redux Toolkit**: https://redux-toolkit.js.org/
- **PostgreSQL**: https://www.postgresql.org/docs/
- **Redis**: https://redis.io/documentation

C.1.2 AI/ML Resources

- **OpenAI API**: https://platform.openai.com/docs
- **HuggingFace**: https://huggingface.co/docs
- **spaCy**: https://spacy.io/usage
- **Sentence-BERT**: https://www.sbert.net/
- **Transformers**: https://huggingface.co/docs/transformers

C.1.3 DevOps & Infrastructure

- **Docker**: https://docs.docker.com/
- **Kubernetes**: https://kubernetes.io/docs/
- **GitHub Actions**: https://docs.github.com/actions
- **AWS**: https://docs.aws.amazon.com/
- **Google Cloud**: https://cloud.google.com/docs

C.2 Tutorials & Courses

C.2.1 Django & Backend

- Django for Beginners - William Vincent
- Django REST Framework Tutorial - https://testdriven.io/
- Two Scoops of Django - Best practices book

C.2.2 React & Frontend

- React Official Tutorial - https://react.dev/learn
- Full Stack Open - https://fullstackopen.com/
- Epic React - Kent C. Dodds

C.2.3 AI/ML

- Natural Language Processing with Transformers - O'Reilly
- Practical NLP - https://www.oreilly.com/
- OpenAI Cookbook - https://github.com/openai/openai-cookbook

C.3 Research Papers

C.3.1 NLP & Transformers

- Attention Is All You Need (Vaswani et al., 2017)
- BERT: Pre-training of Deep Bidirectional Transformers (Devlin et al., 2018)
- Language Models are Few-Shot Learners (Brown et al., 2020) - GPT-3

C.3.2 Recommendation Systems

- Matrix Factorization Techniques for Recommender Systems (Koren et al., 2009)
- Deep Learning based Recommender System: A Survey (Zhang et al., 2019)

C.3.3 Resume Analysis

- Automated Resume Screening using NLP (Various research)
- Skill Extraction from Job Descriptions using NER

C.4 Community Resources

C.4.1 Forums & QA

- Stack Overflow: https://stackoverflow.com/
- Django Forum: https://forum.djangoproject.com/
- Reddit r/django: https://www.reddit.com/r/django/
- Reddit r/reactjs: https://www.reddit.com/r/reactjs/

C.4.2 GitHub Repositories

- Awesome Django: https://github.com/wsvincent/awesome-django
- Awesome React: https://github.com/enaqx/awesome-react
- Awesome NLP: https://github.com/keon/awesome-nlp
- Resume Parser Examples: Search GitHub for resume parsers

C.4.3 Blogs & Articles

- Real Python: https://realpython.com/
- Django Stars Blog: https://djangostars.com/blog/
- Towards Data Science: https://towardsdatascience.com/
- Dev.to: https://dev.to/

C.5 Tools & Utilities

C.5.1 Design Resources

- Figma: https://www.figma.com/ - UI/UX design
- TailwindUI: https://tailwindui.com/ - UI components
- Icons: HeroIcons, FontAwesome, Material Icons
- Color Palettes: Coolors, Adobe Color

C.5.2 Data & Testing

- Faker: Generate fake data
- Mockaroo: Mock data generator
- Postman: API testing
- Insomnia: API client

C.5.3 Productivity

- Notion: Documentation & notes
- Miro: Collaborative whiteboard
- Linear: Issue tracking
- Slack/Discord: Team communication

C.6 Datasets (for Training/Testing)

C.6.1 Resume Datasets

- Kaggle Resume Datasets
- LinkedIn Public Profiles (with permission)
- Synthetic resume generation tools

C.6.2 Job Listings

- Common Crawl job postings
- Indeed/LinkedIn job APIs
- Public job board datasets

C.6.3 NLP Datasets

- HuggingFace Datasets: https://huggingface.co/datasets
- Common Crawl: https://commoncrawl.org/
- Wikipedia dumps for training

C.7 Compliance & Legal

C.7.1 Data Protection

- GDPR Official Text: https://gdpr.eu/
- Privacy by Design Guidelines
- Data Processing Agreements templates

C.7.2 Security Standards

- OWASP Top 10: https://owasp.org/
- CWE/SANS Top 25 Software Errors
- NIST Cybersecurity Framework

C.7.3 Accessibility

- WCAG 2.1 Guidelines: https://www.w3.org/WAI/WCAG21/quickref/
- A11y Project: https://www.a11yproject.com/
- WebAIM Resources: https://webaim.org/

C.8 Inspiration & Similar Projects

C.8.1 Career Platforms

- LinkedIn Resume Builder
- Indeed Resume Services
- Jobscan (ATS optimization)
- VMock (AI resume review)
- InterviewBuddy (interview practice)

C.8.2 AI Writing Assistants

- Grammarly
- Jasper AI
- Copy.ai
- ChatGPT Resume Builder plugins

C.9 Challenge-Specific Resources

C.9.1 IEEE TSYP

- Challenge announcement page
- IEEE Tunisia Section website
- IEEE CyberSecurity Local Group
- Collabratec platform

C.9.2 Regional Context

- Youth unemployment statistics - MENA
- African Development Bank reports
- World Bank employment data
- Regional job market studies

C.10 Recommended Reading

C.10.1 Technical Books

- ”Designing Data-Intensive Applications” - Martin Kleppmann
- ”Clean Code” - Robert C. Martin
- ”System Design Interview” - Alex Xu
- ”Building Microservices” - Sam Newman

C.10.2 AI/ML Books

- ”Speech and Language Processing” - Jurafsky & Martin
- ”Hands-On Machine Learning” - Aurélien Géron
- ”Natural Language Processing with Python” - Bird, Klein, Loper

C.10.3 Product Development

- ”The Lean Startup” - Eric Ries
- ”Inspired” - Marty Cagan
- ”Don’t Make Me Think” - Steve Krug (UX)

C.11 Contact & Support

C.11.1 Project Team

- GitHub Issues: For bug reports and feature requests
- Documentation: Comprehensive guides in /docs
- Email: [Keep anonymous for submission]

C.11.2 Challenge Organizers

- Email: cn@ieee.tn, cs@ieee.tn, cybersecurity@ieee.tn
- Official forms and submission portals

C.12 License Information

C.12.1 Open Source Licenses

- MIT License (recommended for project)
- Apache 2.0 (alternative)
- GPL v3 (if required)

C.12.2 Third-Party Licenses

- Review all dependencies licenses
- Ensure commercial use allowed
- Attribute properly in documentation

This guide is a comprehensive resource for developing UtopiaHire.
For the latest updates, refer to the project repository.

Good luck with the IEEE TSYP13 Technical Challenge!
