NeuralPath Full-Stack Implementation Architecture

🏗️ Tech Stack Overview

Frontend

- **Framework**: Next.js 14 (App Router)
- **UI Library**: Tailwind CSS + Shadcn/ui
- **State Management**: Zustand + React Query
- **Authentication**: NextAuth.js
- **Real-time**: Socket.io
- **Animation**: Framer Motion

Backend

- **API Framework**: FastAPI (Python)
- **Database**: PostgreSQL + Prisma ORM
- **Cache**: Redis
- **Queue**: Celery + RabbitMQ
- **File Storage**: AWS S3 / Cloudflare R2

AI/ML Stack

- **LLM Framework**: LangChain + LlamaIndex
- **Vector DB**: Pinecone / Qdrant
- **ML Models**: HuggingFace Transformers
- **Agents**: AutoGPT / LangGraph
- **Embeddings**: OpenAI Ada-002 / Sentence Transformers
- **Interview Analysis**: Hume AI / DeepFace

Infrastructure

- **Deployment**: Vercel (Frontend) + Railway/Render (Backend)
- **Monitoring**: Sentry + Posthog
- **CI/CD**: GitHub Actions
- **Container**: Docker + Kubernetes (scale phase)

📦 Core Implementation Modules

Module 1: AI Career Roadmap Generator

python

```python
# backend/app/modules/roadmap/agent.py
from langchain.agents import initialize_agent, AgentType
from langchain.tools import Tool
from langchain.llms import OpenAI
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Pinecone
from langchain.chains import RetrievalQA
import asyncio

class CareerRoadmapAgent:
    def __init__(self):
        self.llm = OpenAI(temperature=0.7)
        self.embeddings = OpenAIEmbeddings()
        self.vector_store = Pinecone.from_existing_index(
            "career-transitions",
            self.embeddings
        )

    async def analyze_user_profile(self, user_data):
        """Analyze user's current skills and experience"""
        tools = [
            Tool(
                name="Skill_Analyzer",
                func=self._analyze_skills,
                description="Analyzes user's current technical skills"
            ),
            Tool(
                name="Gap_Identifier",
                func=self._identify_gaps,
                description="Identifies skill gaps for target role"
            ),
            Tool(
                name="Timeline_Calculator",
                func=self._calculate_timeline,
                description="Estimates learning timeline"
            )
        ]

        agent = initialize_agent(
            tools=tools,
            llm=self.llm,
            agent=AgentType.STRUCTURED_CHAT_ZERO_SHOT_REACT_DESCRIPTION,
            verbose=True
        )

        return await agent.arun(
            f"Create a roadmap for: {user_data}"
        )

    async def generate_personalized_roadmap(self, profile_analysis):
        """Generate detailed learning roadmap with milestones"""
        roadmap_chain = RetrievalQA.from_chain_type(
            llm=self.llm,
            chain_type="stuff",
            retriever=self.vector_store.as_retriever()
        )

        roadmap = await roadmap_chain.arun({
            "query": f"Generate roadmap based on: {profile_analysis}",
            "include": ["phases", "skills", "projects", "timeline"]
        })

        return self._structure_roadmap(roadmap)
```

typescript

```tsx
// frontend/app/roadmap/components/RoadmapVisualization.tsx
import React from 'react';
import { motion } from 'framer-motion';
import { Card } from '@/components/ui/card';

interface RoadmapPhase {
  id: string;
  title: string;
  duration: string;
  skills: Skill[];
  projects: Project[];
  status: 'completed' | 'active' | 'locked';
}

export const RoadmapVisualization: React.FC<{
  phases: RoadmapPhase[]
}> = ({ phases }) => {
  return (
    <div className="relative">
      {/* Timeline Line */}
      <div className="absolute left-1/2 transform -translate-x-1/2 w-1 h-full bg-gradi

      {phases.map((phase, index) => (
        <motion.div
          key={phase.id}
          initial={{ opacity: 0, y: 50 }}
          animate={{ opacity: 1, y: 0 }}
          transition={{ delay: index * 0.1 }}
          className={`relative ${
            index % 2 === 0 ? 'pr-1/2' : 'pl-1/2 ml-auto'
          }`}
        >
          <Card className={`p-6 ${
            phase.status === 'completed' ? 'opacity-60' : ''
          }`}>
            <h3 className="text-xl font-bold">{phase.title}</h3>
            <p className="text-sm text-gray-500">{phase.duration}</p>

            {/* Skills Grid */}
            <div className="mt-4 grid grid-cols-2 gap-2">
              {phase.skills.map(skill => (
                <SkillBadge key={skill.id} skill={skill} />
              ))}
            </div>

            {/* Projects */}
            <div className="mt-4">
              {phase.projects.map(project => (
                <ProjectCard key={project.id} project={project} />
              ))}
            </div>
          </Card>
        </motion.div>
      ))}
    </div>
  );
};
```

Module 2: Smart Job Matching Engine

```
python
```

```python
# backend/app/modules/jobs/matching_engine.py
from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
from typing import List, Dict
import asyncio

class JobMatchingEngine:
    def __init__(self):
        self.model = SentenceTransformer('all-mpnet-base-v2')
        self.skill_extractor = SkillExtractionAgent()

    async def match_jobs(self, user_profile: Dict, jobs: List[Dict]) -> List[Dict]:
        """AI-powered job matching with skill gap analysis"""

        # Extract and embed user skills
        user_skills = await self.skill_extractor.extract(user_profile)
        user_embedding = self.model.encode(user_skills)

        matches = []
        for job in jobs:
            # Extract job requirements
            job_requirements = await self.skill_extractor.extract(job)
            job_embedding = self.model.encode(job_requirements)

            # Calculate similarity
            similarity = cosine_similarity(
                [user_embedding],
                [job_embedding]
            )[0][0]

            # Identify gaps and growth opportunities
            gaps = await self._analyze_gaps(user_skills, job_requirements)

            matches.append({
                "job": job,
                "match_score": similarity,
                "skill_gaps": gaps,
                "growth_potential": self._calculate_growth_potential(gaps),
                "estimated_prep_time": self._estimate_prep_time(gaps)
            })

        # Sort by optimal match (not just highest score)
        return sorted(
            matches,
            key=lambda x: self._calculate_opportunity_score(x),
            reverse=True
        )

    def _calculate_opportunity_score(self, match):
        """Balance between match score and growth potential"""
        # 70-85% match is often better than 95% (room to grow)
        if 0.7 <= match['match_score'] <= 0.85:
            growth_bonus = 0.2
        else:
            growth_bonus = 0

        return match['match_score'] + growth_bonus
```

python

```python
# backend/app/modules/jobs/job_tracking.py
from sqlalchemy.ext.asyncio import AsyncSession
from app.models import JobApplication, User
import asyncio
from datetime import datetime, timedelta

class ApplicationTracker:
    def __init__(self, db: AsyncSession):
        self.db = db
        self.notification_agent = NotificationAgent()

    async def track_application(self, user_id: int, job_id: str, data: Dict):
        """Track job application with AI-powered follow-up suggestions"""

        application = JobApplication(
            user_id=user_id,
            job_id=job_id,
            status="applied",
            applied_at=datetime.utcnow(),
            company_data=data
        )

        self.db.add(application)
        await self.db.commit()

        # Schedule AI follow-up reminders
        await self._schedule_followups(application)

        # Generate tailored resume suggestions
        resume_tips = await self._generate_resume_tips(data)

        return {
            "application_id": application.id,
            "resume_tips": resume_tips,
            "next_steps": await self._suggest_next_steps(data)
        }
```

Module 3: AI Interview Coach

python

```python
# backend/app/modules/interview/coach.py
from transformers import pipeline
import cv2
import numpy as np
from hume import HumeStreamClient
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory

class AIInterviewCoach:
    def __init__(self):
        self.llm = ChatOpenAI(model="gpt-4", temperature=0.7)
        self.emotion_client = HumeStreamClient(api_key=HUME_API_KEY)
        self.speech_analyzer = pipeline(
            "automatic-speech-recognition",
            model="openai/whisper-base"
        )

    async def conduct_mock_interview(self, job_role: str, company: str):
        """Conduct realistic mock interview with real-time feedback"""

        # Generate role-specific questions
        questions = await self._generate_interview_questions(
            role=job_role,
            company=company,
            difficulty="progressive"  # Start easy, get harder
        )

        # Initialize conversation chain with interview persona
        memory = ConversationBufferMemory()
        interview_chain = ConversationChain(
            llm=self.llm,
            memory=memory,
            prompt=self._create_interviewer_prompt(company, job_role)
        )

        return InterviewSession(
            questions=questions,
            chain=interview_chain,
            analyzer=self
        )

    async def analyze_response(self, video_stream, audio_stream):
        """Real-time analysis of interview response"""
        # Parallel analysis
        emotion_task = self._analyze_emotions(video_stream)
        speech_task = self._analyze_speech(audio_stream)
        content_task = self._analyze_content(audio_stream)

        emotion_data, speech_data, content_data = await asyncio.gather(
            emotion_task, speech_task, content_task
        )

        return {
            "emotion_analysis": emotion_data,
            "speech_patterns": speech_data,
            "content_quality": content_data,
            "suggestions": await self._generate_suggestions(
                emotion_data, speech_data, content_data
            )
        }

    async def _analyze_emotions(self, video_stream):
        """Analyze facial expressions and body language"""
        async with self.emotion_client.connect() as socket:
            result = await socket.send_video(video_stream)
            return {
                "confidence": result.predictions.confidence,
                "emotions": result.predictions.emotions,
                "suggestions": self._interpret_emotions(result)
            }
```

typescript

```tsx
// frontend/app/interview/components/InterviewSimulator.tsx
import React, { useState, useRef, useEffect } from 'react';
import { Camera, Mic, MicOff } from 'lucide-react';
import { useWebRTC } from '@/hooks/useWebRTC';
import { motion, AnimatePresence } from 'framer-motion';

export const InterviewSimulator: React.FC = () => {
  const { stream, startRecording, stopRecording } = useWebRTC();
  const [isInterviewing, setIsInterviewing] = useState(false);
  const [currentQuestion, setCurrentQuestion] = useState<Question | null>(null);
  const [feedback, setFeedback] = useState<Feedback | null>(null);

  const handleStartInterview = async () => {
    const session = await api.interview.createSession({
      role: selectedRole,
      company: selectedCompany,
      type: 'behavioral' // or 'technical', 'system-design'
    });

    setIsInterviewing(true);
    await startRecording();

    // Get first question
    const question = await session.getNextQuestion();
    setCurrentQuestion(question);
  };

  const handleAnswerComplete = async () => {
    const { video, audio } = await stopRecording();

    // Get real-time feedback
    const analysis = await api.interview.analyzeResponse({
      video,
      audio,
      question: currentQuestion
    });

    setFeedback(analysis);

    // Get next question or complete
    const nextQuestion = await session.getNextQuestion();
    if (nextQuestion) {
      setCurrentQuestion(nextQuestion);
      await startRecording();
    } else {
      await completeInterview();
    }
  };

  return (
    <div className="max-w-4xl mx-auto p-6">
      <div className="grid grid-cols-1 lg:grid-cols-2 gap-6">
        {/* Video Preview */}
        <div className="relative">
          <video
            ref={videoRef}
            autoPlay
            muted
            className="w-full rounded-lg bg-gray-900"
          />

          {/* Real-time emotion indicator */}
          <AnimatePresence>
            {feedback?.emotions && (
              <motion.div
                initial={{ opacity: 0, y: -20 }}
                animate={{ opacity: 1, y: 0 }}
                exit={{ opacity: 0 }}
                className="absolute top-4 right-4 bg-black/50 backdrop-blur rounded-lg
              >
                <EmotionIndicator emotions={feedback.emotions} />
              </motion.div>
            )}
          </AnimatePresence>
        </div>

        {/* Question & Controls */}
        <div className="space-y-6">
          {currentQuestion && (
            <Card className="p-6">
              <h3 className="text-lg font-semibold mb-2">
                Question {currentQuestion.number} of {totalQuestions}
              </h3>
              <p className="text-xl">{currentQuestion.text}</p>

              {/* Speaking tips */}
              <div className="mt-4 text-sm text-gray-500">
                <p>💡 Remember: Use STAR format for behavioral questions</p>
              </div>
            </Card>
          )}

          {/* Feedback Panel */}
          {feedback && (
            <Card className="p-6 border-blue-500">
              <h4 className="font-semibold mb-3">AI Feedback</h4>

              <div className="space-y-3">
                <FeedbackItem
                  label="Content Quality"
                  score={feedback.contentScore}
                  suggestion={feedback.contentSuggestion}
                />

                <FeedbackItem
                  label="Delivery"
                  score={feedback.deliveryScore}
                  suggestion={feedback.deliverySuggestion}
                />

                <FeedbackItem
                  label="Body Language"
                  score={feedback.bodyLanguageScore}
                  suggestion={feedback.bodyLanguageSuggestion}
                />
              </div>
            </Card>
          )}

          {/* Control Buttons */}
          <div className="flex gap-4">
            {!isInterviewing ? (
              <Button onClick={handleStartInterview} className="flex-1">
                <Camera className="mr-2" /> Start Interview
              </Button>
            ) : (
              <>
                <Button
                  variant="secondary"
                  onClick={handlePause}
                  className="flex-1"
```

```
                >
                  <MicOff className="mr-2" /> Pause
                </Button>
                <Button
                  onClick={handleAnswerComplete}
                  className="flex-1"
                >
                  Next Question →
                </Button>
              </>
            )}
          </div>
        </div>
      </div>
    </div>
  );
};
```

Module 4: Agentic Learning System

python

```python
# backend/app/modules/learning/adaptive_agent.py
from langchain.agents import Tool, AgentExecutor
from langchain.memory import ConversationSummaryMemory
from langchain.chains import LLMChain
from app.modules.learning.content_retriever import ContentRetriever

class AdaptiveLearningAgent:
    """Agentic AI that adapts learning content based on user progress"""

    def __init__(self):
        self.llm = ChatOpenAI(model="gpt-4")
        self.memory = ConversationSummaryMemory(llm=self.llm)
        self.content_retriever = ContentRetriever()

        self.tools = [
            Tool(
                name="assess_understanding",
                func=self._assess_understanding,
                description="Assess user's understanding of current topic"
            ),
            Tool(
                name="adjust_difficulty",
                func=self._adjust_difficulty,
                description="Adjust content difficulty based on performance"
            ),
            Tool(
                name="generate_exercise",
                func=self._generate_exercise,
                description="Generate practice exercise at appropriate level"
            ),
            Tool(
                name="provide_hint",
                func=self._provide_hint,
                description="Provide contextual hints when user is stuck"
            )
        ]

    async def guide_learning_session(self, user_id: str, topic: str):
        """Conduct an adaptive learning session"""

        # Get user's current level and learning history
        user_profile = await self._get_user_profile(user_id)
        # Initialize agent with user context
        agent = AgentExecutor.from_agent_and_tools(
            agent=self._create_learning_agent(),
            tools=self.tools,
            memory=self.memory,
            verbose=True
        )

        # Start adaptive session
        session_plan = await agent.arun(
            f"Create adaptive learning session for {user_profile} on {topic}"
        )

        return LearningSession(
            plan=session_plan,
            agent=agent,
            user_id=user_id
        )

    async def _assess_understanding(self, user_response: str) -> Dict:
        """Use AI to assess conceptual understanding"""
        assessment_prompt = PromptTemplate(
            template="""
            Assess the user's understanding based on their response.
            Response: {response}

            Evaluate:
            1. Conceptual accuracy (0-100)
            2. Depth of understanding (surface/intermediate/deep)
            3. Misconceptions identified
            4. Areas needing reinforcement

            Return structured assessment.
            """,
            input_variables=["response"]
        )

        chain = LLMChain(llm=self.llm, prompt=assessment_prompt)
        return await chain.arun(response=user_response)
```

Module 5: Progress Tracking & Analytics

python

```python
# backend/app/modules/analytics/progress_tracker.py
from datetime import datetime, timedelta
import pandas as pd
from app.ml.prediction_models import ProgressPredictor

class ProgressAnalytics:
    def __init__(self):
        self.predictor = ProgressPredictor()

    async def analyze_user_progress(self, user_id: str) -> Dict:
        """Comprehensive progress analysis with predictions"""

        # Fetch user's learning data
        learning_data = await self._fetch_learning_history(user_id)

        # Calculate metrics
        metrics = {
            "completion_rate": self._calculate_completion_rate(learning_data),
            "learning_velocity": self._calculate_velocity(learning_data),
            "skill_mastery": await self._assess_skill_mastery(learning_data),
            "engagement_score": self._calculate_engagement(learning_data)
        }

        # Predict future progress
        predictions = await self.predictor.predict_timeline(
            current_progress=metrics,
            target_role=learning_data['target_role']
        )

        # Identify at-risk areas
        risk_analysis = self._identify_risks(metrics, predictions)

        return {
            "current_metrics": metrics,
            "predictions": predictions,
            "risks": risk_analysis,
            "recommendations": await self._generate_recommendations(
                metrics, predictions, risk_analysis
            )
        }
```

Module 6: Real-time Communication

```
python
```

```python
# backend/app/modules/realtime/websocket_manager.py
from fastapi import WebSocket
from typing import Dict, List
import json

class ConnectionManager:
    def __init__(self):
        self.active_connections: Dict[str, List[WebSocket]] = {}
        self.user_sessions: Dict[str, UserSession] = {}

    async def connect(self, websocket: WebSocket, user_id: str):
        await websocket.accept()
        if user_id not in self.active_connections:
            self.active_connections[user_id] = []
        self.active_connections[user_id].append(websocket)

        # Initialize user session
        self.user_sessions[user_id] = UserSession(user_id)

    async def broadcast_progress_update(self, user_id: str, progress_data: Dict):
        """Send real-time progress updates"""
        if user_id in self.active_connections:
            message = {
                "type": "progress_update",
                "data": progress_data,
                "timestamp": datetime.utcnow().isoformat()
            }

            for connection in self.active_connections[user_id]:
                await connection.send_json(message)

    async def send_ai_suggestion(self, user_id: str, suggestion: Dict):
        """Send AI-generated suggestions in real-time"""
        if user_id in self.active_connections:
            message = {
                "type": "ai_suggestion",
                "data": suggestion,
                "priority": suggestion.get("priority", "normal")
            }

            for connection in self.active_connections[user_id]:
                await connection.send_json(message)
```

Module 7: API Gateway & Authentication

```
python
```

```python
# backend/app/api/v1/endpoints/roadmap.py
from fastapi import APIRouter, Depends, HTTPException
from app.auth import get_current_user
from app.modules.roadmap.agent import CareerRoadmapAgent

router = APIRouter()

@router.post("/generate")
async def generate_roadmap(
    profile_data: UserProfileSchema,
    current_user: User = Depends(get_current_user)
):
    """Generate personalized AI roadmap"""
    agent = CareerRoadmapAgent()

    # Analyze profile
    analysis = await agent.analyze_user_profile({
        **profile_data.dict(),
        "user_id": current_user.id
    })

    # Generate roadmap
    roadmap = await agent.generate_personalized_roadmap(analysis)

    # Save to database
    await save_roadmap(current_user.id, roadmap)

    # Trigger background jobs
    await queue_learning_content_preparation(current_user.id, roadmap)

    return {
        "roadmap": roadmap,
        "estimated_completion": roadmap.estimated_months,
        "first_action": roadmap.phases[0].first_action
    }

@router.get("/progress")
async def get_progress(
    current_user: User = Depends(get_current_user)
):
    """Get current progress with AI insights"""
    analytics = ProgressAnalytics()
    progress = await analytics.analyze_user_progress(current_user.id)
    return progress

@router.post("/update-progress")
async def update_progress(
    update: ProgressUpdateSchema,
    current_user: User = Depends(get_current_user)
):
    """Update progress and get AI recommendations"""
    # Update progress
    await update_user_progress(current_user.id, update)

    # Get AI recommendations for next steps
    agent = LearningAgent()
    next_steps = await agent.recommend_next_steps(
        current_user.id,
        update
    )

    # Broadcast update via WebSocket
    await manager.broadcast_progress_update(
        current_user.id,
        {"progress": update, "next_steps": next_steps}
    )

    return {"success": True, "next_steps": next_steps}
```

Module 8: Background Jobs & Task Queue

python

```python
# backend/app/workers/tasks.py
from celery import Celery
from app.modules.jobs.scraper import JobScraper
from app.modules.learning.content_generator import ContentGenerator

celery_app = Celery(
    'neuralpath',
    broker='redis://localhost:6379',
    backend='redis://localhost:6379'
)

@celery_app.task
def scrape_job_listings():
    """Periodically scrape and analyze job listings"""
    scraper = JobScraper()

    # Scrape from multiple sources
    sources = ['linkedin', 'indeed', 'angellist', 'ycombinator']

    for source in sources:
        jobs = scraper.scrape(source)

        # AI analysis of each job
        for job in jobs:
            analyze_job_requirements.delay(job)

@celery_app.task
def analyze_job_requirements(job_data):
    """Use AI to extract and structure job requirements"""
    analyzer = JobRequirementAnalyzer()

    requirements = analyzer.extract_requirements(job_data)
    skills = analyzer.extract_skills(job_data)

    # Store in vector database for matching
    vector_store.add_job(
        job_id=job_data['id'],
        requirements=requirements,
        skills=skills,
        embedding=analyzer.generate_embedding(job_data)
    )

@celery_app.task
```

```python
def generate_daily_learning_content(user_id):
    """Generate personalized daily learning content"""
    generator = ContentGenerator()
    user_profile = get_user_profile(user_id)

    # Generate adaptive content
    daily_content = generator.create_daily_plan(
        user_profile=user_profile,
        previous_performance=get_recent_performance(user_id),
        time_available=user_profile.daily_time_commitment
    )

    # Notify user
    send_notification(
        user_id,
        "Your daily learning plan is ready!",
        daily_content.summary
    )
```

🚀 Deployment Architecture

yaml

```yaml
# docker-compose.yml
version: '3.8'

services:
  frontend:
    build: ./frontend
    ports:
      - "3000:3000"
    environment:
      - NEXT_PUBLIC_API_URL=http://api:8000
      - NEXT_PUBLIC_WS_URL=ws://api:8000/ws

  api:
    build: ./backend
    ports:
      - "8000:8000"
    environment:
      - DATABASE_URL=postgresql://user:pass@db:5432/neuralpath
      - REDIS_URL=redis://redis:6379
      - OPENAI_API_KEY=${OPENAI_API_KEY}
    depends_on:
      - db
      - redis

  worker:
    build: ./backend
    command: celery -A app.workers worker --loglevel=info
    environment:
      - CELERY_BROKER_URL=redis://redis:6379
    depends_on:
      - redis

  db:
    image: postgres:15
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      - POSTGRES_DB=neuralpath
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass

  redis:
    image: redis:7-alpine
  vector_db:
    image: qdrant/qdrant
    ports:
      - "6333:6333"
    volumes:
      - qdrant_data:/qdrant/storage

volumes:
  postgres_data:
  qdrant_data:
```

🔧 Environment Setup

bash

```bash
# Backend setup
cd backend
python -m venv venv
source venv/bin/activate  # or `venv\Scripts\activate` on Windows
pip install -r requirements.txt

# Frontend setup
cd frontend
npm install

# Environment variables
cp .env.example .env
# Add your API keys and configuration

# Database setup
python -m alembic upgrade head

# Start development servers
# Terminal 1: Backend
uvicorn app.main:app --reload

# Terminal 2: Frontend
npm run dev

# Terminal 3: Celery worker
celery -A app.workers worker --loglevel=info

# Terminal 4: Celery beat (for scheduled tasks)
celery -A app.workers beat --loglevel=info
```

📊 Monitoring & Analytics

python

```python
# backend/app/monitoring/setup.py
import sentry_sdk
from sentry_sdk.integrations.fastapi import FastApiIntegration
from sentry_sdk.integrations.celery import CeleryIntegration
import posthog

# Sentry for error tracking
sentry_sdk.init(
    dsn=SENTRY_DSN,
    integrations=[
        FastApiIntegration(transaction_style="endpoint"),
        CeleryIntegration()
    ],
    traces_sample_rate=0.1,
    profiles_sample_rate=0.1,
)

# PostHog for product analytics
posthog.api_key = POSTHOG_API_KEY
posthog.host = 'https://app.posthog.com'

# Custom metrics tracking
class MetricsTracker:
    @staticmethod
    def track_learning_progress(user_id: str, event_data: Dict):
        posthog.capture(
            user_id,
            'learning_progress',
            properties=event_data
        )

    @staticmethod
    def track_job_application(user_id: str, job_data: Dict):
        posthog.capture(
            user_id,
            'job_application',
            properties=job_data
        )
```

🔐 Security Considerations

1. **Authentication**: JWT tokens with refresh mechanism
2. **API Rate Limiting**: Redis-based rate limiting per user
3. **Data Encryption**: AES-256 for sensitive data
4. **HTTPS**: SSL/TLS for all communications
5. **Input Validation**: Pydantic models for all inputs

🧪 Testing Strategy

Unit Tests

python

```python
# backend/tests/test_roadmap_agent.py
import pytest
from app.modules.roadmap.agent import CareerRoadmapAgent
from unittest.mock import Mock, patch

@pytest.mark.asyncio
async def test_roadmap_generation():
    agent = CareerRoadmapAgent()

    user_profile = {
        "current_role": "Software Engineer",
        "target_role": "ML Engineer",
        "experience_years": 5,
        "skills": ["Python", "JavaScript", "SQL"]
    }

    with patch('app.modules.roadmap.agent.OpenAI') as mock_llm:
        mock_llm.return_value.arun.return_value = "Mocked roadmap response"

        roadmap = await agent.generate_personalized_roadmap(user_profile)

        assert roadmap is not None
        assert len(roadmap.phases) > 0
        assert roadmap.estimated_months > 0

@pytest.mark.asyncio
async def test_skill_gap_analysis():
    agent = CareerRoadmapAgent()

    current_skills = ["Python", "SQL", "Git"]
    target_skills = ["Python", "TensorFlow", "PyTorch", "MLOps"]

    gaps = await agent._identify_gaps(current_skills, target_skills)

    assert "TensorFlow" in gaps
    assert "PyTorch" in gaps
    assert "Python" not in gaps  # Already has this skill
```

Integration Tests

python

```python
# backend/tests/integration/test_job_matching.py
import pytest
from fastapi.testclient import TestClient
from app.main import app
from app.database import get_db

client = TestClient(app)

@pytest.mark.integration
async def test_job_matching_flow():
    # Create test user
    user_response = client.post("/api/v1/auth/register", json={
        "email": "test@example.com",
        "password": "testpass123",
        "current_role": "Data Analyst"
    })

    token = user_response.json()["access_token"]
    headers = {"Authorization": f"Bearer {token}"}

    # Generate roadmap
    roadmap_response = client.post(
        "/api/v1/roadmap/generate",
        headers=headers,
        json={
            "target_role": "Data Scientist",
            "time_commitment": "part_time"
        }
    )

    assert roadmap_response.status_code == 200

    # Get job matches
    jobs_response = client.get(
        "/api/v1/jobs/matches",
        headers=headers
    )

    assert jobs_response.status_code == 200
    jobs = jobs_response.json()
    assert len(jobs["matches"]) > 0
    assert all(0.6 <= job["match_score"] <= 0.9 for job in jobs["matches"])
```

Frontend Tests

typescript

```typescript
// frontend/__tests__/RoadmapVisualization.test.tsx
import { render, screen, waitFor } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import { RoadmapVisualization } from '@/app/roadmap/components/RoadmapVisualization';

describe('RoadmapVisualization', () => {
  const mockPhases = [
    {
      id: '1',
      title: 'Mathematical Foundations',
      duration: '2 months',
      skills: [
        { id: 's1', name: 'Linear Algebra', level: 'beginner' },
        { id: 's2', name: 'Statistics', level: 'beginner' }
      ],
      projects: [
        { id: 'p1', name: 'Gradient Descent Implementation' }
      ],
      status: 'active' as const
    }
  ];

  it('renders roadmap phases correctly', () => {
    render(<RoadmapVisualization phases={mockPhases} />);

    expect(screen.getByText('Mathematical Foundations')).toBeInTheDocument();
    expect(screen.getByText('2 months')).toBeInTheDocument();
    expect(screen.getByText('Linear Algebra')).toBeInTheDocument();
  });

  it('animates phase cards on mount', async () => {
    const { container } = render(<RoadmapVisualization phases={mockPhases} />);

    const phaseCard = container.querySelector('.phase-card');
    expect(phaseCard).toHaveStyle({ opacity: 0 });

    await waitFor(() => {
      expect(phaseCard).toHaveStyle({ opacity: 1 });
    });
  });
});
```

📈 Scaling Considerations

Horizontal Scaling

```
yaml
```

```yaml
# kubernetes/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: neuralpath-api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: neuralpath-api
  template:
    metadata:
      labels:
        app: neuralpath-api
    spec:
      containers:
      - name: api
        image: neuralpath/api:latest
        ports:
        - containerPort: 8000
        env:
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: neuralpath-secrets
              key: database-url
        resources:
          requests:
            memory: "512Mi"
            cpu: "500m"
          limits:
            memory: "1Gi"
            cpu: "1000m"
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: neuralpath-api-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: neuralpath-api
  minReplicas: 3
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
```

Caching Strategy

python

```python
# backend/app/cache/strategy.py
from functools import wraps
import hashlib
import json
from app.cache import redis_client

def cache_result(expiration=3600):
    """Decorator for caching function results"""
    def decorator(func):
        @wraps(func)
        async def wrapper(*args, **kwargs):
            # Generate cache key
            cache_key = f"{func.__name__}:{hashlib.md5(
                json.dumps([args, kwargs], sort_keys=True).encode()
            ).hexdigest()}"

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

            # Execute function
            result = await func(*args, **kwargs)

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

            return result
        return wrapper
    return decorator

# Usage example
@cache_result(expiration=7200)  # 2 hours
async def get_job_recommendations(user_id: str):
    # Expensive operation
    return await matching_engine.get_recommendations(user_id)
```

🚀 Performance Optimization

Database Query Optimization

```python
# backend/app/database/optimizations.py
from sqlalchemy import select, func
from sqlalchemy.orm import selectinload, joinedload

class OptimizedQueries:
    @staticmethod
    async def get_user_with_progress(db: AsyncSession, user_id: int):
        """Optimized query with eager loading"""
        result = await db.execute(
            select(User)
            .options(
                selectinload(User.roadmap)
                .selectinload(Roadmap.phases)
                .selectinload(Phase.skills),
                selectinload(User.progress_records),
                selectinload(User.job_applications)
            )
            .where(User.id == user_id)
        )
        return result.scalar_one_or_none()

    @staticmethod
    async def get_learning_analytics(db: AsyncSession, user_id: int):
        """Aggregated analytics query"""
        result = await db.execute(
            select(
                func.count(LearningSession.id).label('total_sessions'),
                func.avg(LearningSession.duration).label('avg_duration'),
                func.sum(LearningSession.completed_exercises).label('total_exercises')
            )
            .where(LearningSession.user_id == user_id)
            .where(LearningSession.created_at >= func.now() - timedelta(days=30))
        )
        return result.first()
```

AI Model Optimization

python

```python
# backend/app/ml/optimization.py
import torch
from transformers import AutoModel, AutoTokenizer
from functools import lru_cache

class OptimizedEmbeddings:
    def __init__(self):
        self.device = torch.device(
            "cuda" if torch.cuda.is_available() else "cpu"
        )
        self.model = AutoModel.from_pretrained(
            "sentence-transformers/all-mpnet-base-v2"
        ).to(self.device)
        self.tokenizer = AutoTokenizer.from_pretrained(
            "sentence-transformers/all-mpnet-base-v2"
        )

        # Enable optimization
        if self.device.type == "cuda":
            self.model = torch.compile(self.model)  # PyTorch 2.0+

    @lru_cache(maxsize=10000)
    def get_embedding(self, text: str):
        """Cached embedding generation"""
        inputs = self.tokenizer(
            text,
            return_tensors="pt",
            truncation=True,
            padding=True,
            max_length=512
        ).to(self.device)

        with torch.no_grad():
            outputs = self.model(**inputs)
            embeddings = outputs.last_hidden_state.mean(dim=1)

        return embeddings.cpu().numpy()

    def batch_embeddings(self, texts: List[str], batch_size: int = 32):
        """Efficient batch processing"""
        embeddings = []

        for i in range(0, len(texts), batch_size):
            batch = texts[i:i + batch_size]
            inputs = self.tokenizer(
                batch,
                return_tensors="pt",
                truncation=True,
                padding=True,
                max_length=512
            ).to(self.device)

            with torch.no_grad():
                outputs = self.model(**inputs)
                batch_embeddings = outputs.last_hidden_state.mean(dim=1)
                embeddings.extend(batch_embeddings.cpu().numpy())

        return embeddings
```

📊 Real-time Analytics Dashboard

typescript

```tsx
// frontend/app/analytics/components/ProgressDashboard.tsx
import React, { useEffect, useState } from 'react';
import { Line, Bar, Doughnut } from 'react-chartjs-2';
import { useWebSocket } from '@/hooks/useWebSocket';

export const ProgressDashboard: React.FC = () => {
  const [analytics, setAnalytics] = useState<Analytics | null>(null);
  const { data: realtimeData } = useWebSocket('/ws/analytics');

  useEffect(() => {
    // Update analytics when receiving real-time data
    if (realtimeData) {
      setAnalytics(prev => ({
        ...prev,
        ...realtimeData
      }));
    }
  }, [realtimeData]);

  const learningVelocityData = {
    labels: analytics?.dates || [],
    datasets: [{
      label: 'Learning Velocity',
      data: analytics?.velocity || [],
      borderColor: 'rgb(79, 172, 254)',
      backgroundColor: 'rgba(79, 172, 254, 0.1)',
      tension: 0.4
    }]
  };

  const skillMasteryData = {
    labels: analytics?.skills || [],
    datasets: [{
      label: 'Mastery Level',
      data: analytics?.masteryLevels || [],
      backgroundColor: [
        'rgba(79, 172, 254, 0.8)',
        'rgba(0, 242, 254, 0.8)',
        'rgba(102, 126, 234, 0.8)',
        'rgba(118, 75, 162, 0.8)'
      ]
    }]
  };
```

```
return (
  <div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-6">
    {/* Learning Streak */}
    <Card className="p-6">
      <h3 className="text-lg font-semibold mb-4">Learning Streak</h3>
      <div className="text-4xl font-bold text-blue-500">
        {analytics?.streak || 0} days
      </div>
      <p className="text-sm text-gray-500 mt-2">
        Keep it up! You're in the top {analytics?.percentile || 0}%
      </p>
    </Card>

    {/* Progress to Goal */}
    <Card className="p-6">
      <h3 className="text-lg font-semibold mb-4">Progress to AI Engineer</h3>
      <CircularProgress
        value={analytics?.overallProgress || 0}
        size={120}
        strokeWidth={10}
      />
      <p className="text-center mt-4">
        {analytics?.monthsRemaining || 0} months to go
      </p>
    </Card>

    {/* Learning Velocity Chart */}
    <Card className="p-6 col-span-full lg:col-span-2">
      <h3 className="text-lg font-semibold mb-4">Learning Velocity</h3>
      <Line data={learningVelocityData} options={chartOptions} />
    </Card>

    {/* Skill Mastery */}
    <Card className="p-6">
      <h3 className="text-lg font-semibold mb-4">Skill Mastery</h3>
      <Doughnut data={skillMasteryData} options={doughnutOptions} />
    </Card>

    {/* AI Insights */}
    <Card className="p-6 col-span-full border-blue-500">
      <h3 className="text-lg font-semibold mb-4">
        🤖 AI Insights
      </h3>
```

```
        <div className="space-y-3">
          {analytics?.insights?.map((insight, idx) => (
            <InsightCard key={idx} insight={insight} />
          ))}
        </div>
      </Card>
    </div>
  );
};
```

🌐 Multi-tenant Architecture

python

```python
# backend/app/models/tenant.py
from sqlalchemy import Column, String, Integer, ForeignKey
from sqlalchemy.orm import relationship
from app.database import Base

class Organization(Base):
    __tablename__ = "organizations"

    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    domain = Column(String, unique=True)
    plan = Column(String, default="free")

    # Relationships
    users = relationship("User", back_populates="organization")
    custom_roadmaps = relationship("CustomRoadmap", back_populates="organization")

class TenantContext:
    """Middleware for multi-tenant data isolation"""

    def __init__(self, request: Request):
        self.request = request
        self.tenant_id = None

    async def __call__(self, request: Request, call_next):
        # Extract tenant from subdomain or header
        host = request.headers.get("host", "")
        subdomain = host.split(".")[0]

        if subdomain and subdomain != "www":
            org = await get_org_by_domain(subdomain)
            if org:
                request.state.tenant_id = org.id

        response = await call_next(request)
        return response
```

🔄 Continuous Learning Pipeline

python

```python
# backend/app/ml/continuous_learning.py
from datetime import datetime
import mlflow
from app.ml.models import JobMatchingModel

class ContinuousLearningPipeline:
    def __init__(self):
        self.mlflow_uri = "http://mlflow:5000"
        mlflow.set_tracking_uri(self.mlflow_uri)

    async def retrain_matching_model(self):
        """Retrain job matching model with new data"""

        # Collect recent user feedback
        feedback_data = await self._collect_feedback_data()

        # Prepare training dataset
        X_train, y_train = self._prepare_training_data(feedback_data)

        with mlflow.start_run():
            # Train new model
            model = JobMatchingModel()
            model.train(X_train, y_train)

            # Evaluate performance
            metrics = model.evaluate(X_test, y_test)

            # Log metrics
            mlflow.log_metrics(metrics)

            # Compare with production model
            if metrics['accuracy'] > self.production_metrics['accuracy']:
                # Register new model
                mlflow.sklearn.log_model(
                    model,
                    "job_matching_model",
                    registered_model_name="JobMatchingModel"
                )

                # Deploy new model
                await self._deploy_new_model(model)

    async def _deploy_new_model(self, model):
        """Blue-green deployment for ML models"""

        # Save model to S3
        model_path = f"models/job_matching/{datetime.now().isoformat()}"
        await upload_to_s3(model, model_path)

        # Update model registry
        await update_model_registry({
            "model_name": "job_matching",
            "version": model.version,
            "path": model_path,
            "status": "staging"
        })

        # Gradual rollout
        await self._gradual_rollout(model_path)
```

🎯 A/B Testing Framework

python

```python
# backend/app/experiments/ab_testing.py
from typing import Dict, Any
import random
from app.analytics import track_event

class ExperimentManager:
    def __init__(self):
        self.experiments = {
            "onboarding_flow": {
                "variants": ["control", "guided", "ai_assisted"],
                "traffic_split": [0.33, 0.33, 0.34]
            },
            "matching_algorithm": {
                "variants": ["baseline", "neural_collaborative"],
                "traffic_split": [0.5, 0.5]
            }
        }

    def get_variant(self, user_id: str, experiment_name: str) -> str:
        """Deterministic variant assignment"""
        experiment = self.experiments.get(experiment_name)
        if not experiment:
            return "control"

        # Hash user_id for consistent assignment
        hash_value = int(hashlib.md5(
            f"{user_id}:{experiment_name}".encode()
        ).hexdigest(), 16)

        # Assign variant based on traffic split
        roll = (hash_value % 100) / 100
        cumulative = 0

        for variant, split in zip(
            experiment["variants"],
            experiment["traffic_split"]
        ):
            cumulative += split
            if roll < cumulative:
                # Track assignment
                track_event(
                    user_id,
                    "experiment_assigned",
                    {
                        "experiment": experiment_name,
                        "variant": variant
                    }
                )
                return variant

        return experiment["variants"][-1]

    async def track_conversion(
        self,
        user_id: str,
        experiment_name: str,
        metric: str,
        value: Any
    ):
        """Track experiment conversions"""
        variant = self.get_variant(user_id, experiment_name)

        await track_event(
            user_id,
            f"experiment_{metric}",
            {
                "experiment": experiment_name,
                "variant": variant,
                "value": value
            }
        )
```

📱 Mobile App Architecture (React Native)

typescript

```tsx
// mobile/src/screens/RoadmapScreen.tsx
import React from 'react';
import {
  View,
  ScrollView,
  Text,
  StyleSheet,
  Animated
} from 'react-native';
import { useRoadmap } from '@/hooks/useRoadmap';
import { PhaseCard } from '@/components/PhaseCard';
import { ProgressHeader } from '@/components/ProgressHeader';

export const RoadmapScreen: React.FC = () => {
  const { roadmap, currentPhase, updateProgress } = useRoadmap();
  const scrollY = new Animated.Value(0);

  const headerOpacity = scrollY.interpolate({
    inputRange: [0, 100],
    outputRange: [1, 0],
    extrapolate: 'clamp'
  });

  return (
    <View style={styles.container}>
      <Animated.View style={[styles.header, { opacity: headerOpacity }]}>
        <ProgressHeader
          currentPhase={currentPhase}
          totalPhases={roadmap?.phases.length || 0}
        />
      </Animated.View>

      <Animated.ScrollView
        onScroll={Animated.event(
          [{ nativeEvent: { contentOffset: { y: scrollY } } }],
          { useNativeDriver: true }
        )}
        scrollEventThrottle={16}
      >
        {roadmap?.phases.map((phase, index) => (
          <PhaseCard
            key={phase.id}
            phase={phase}
```

```
              index={index}
              isActive={phase.id === currentPhase?.id}
              onComplete={() => updateProgress(phase.id)}
            />
          ))}
        </Animated.ScrollView>
      </View>
    );
  };

  const styles = StyleSheet.create({
    container: {
      flex: 1,
      backgroundColor: '#0a0a0a'
    },
    header: {
      position: 'absolute',
      top: 0,
      left: 0,
      right: 0,
      zIndex: 100
    }
  });
```

This complete implementation provides:

1. **Modular Architecture**: Each feature is a self-contained module
2. **Agentic AI Integration**: LangChain agents for intelligent automation
3. **Real-time Features**: WebSocket for live updates
4. **Scalable Infrastructure**: Kubernetes-ready deployment
5. **Performance Optimization**: Caching, batch processing, and compiled models
6. **Testing Strategy**: Comprehensive unit and integration tests
7. **Continuous Learning**: ML model retraining pipeline
8. **A/B Testing**: Built-in experimentation framework
9. **Multi-tenant Support**: Enterprise-ready architecture
10. **Mobile Support**: React Native implementation

The architecture is designed to start simple (MVP) but scale to millions of users with enterprise features when needed.