System Architecture Overview

The system will consist of:

- **Job Scraper Service**: Web scraping/API integration for job discovery
- **Resume Tailoring Service**: NLP-based resume optimization
- **Cover Letter Generation Service**: LLM-powered content generation
- **Application Submission Service**: Form automation/API integration
- **Orchestration Layer**: Workflow management and state tracking

Phase 1: Core Infrastructure (Week 1-2)

1.1 Project Setup

```
# Project structure
job-application-automation/
├── services/
│   ├── job_scraper/
│   ├── resume_tailor/
│   ├── cover_letter_gen/
│   └── application_submitter/
├── orchestration/
├── shared/
│   ├── models/
│   └── utils/
├── config/
└── tests/
```

1.2 Data Models

```
from pydantic import BaseModel
from typing import List, Optional, Dict
from datetime import datetime

class JobPosting(BaseModel):
    id: str
    title: str
    company: str
```

```
    description: str
    requirements: List[str]
    nice_to_haves: List[str]
    keywords: List[str]
    application_url: str
    source: str
    scraped_at: datetime

class Resume(BaseModel):
    sections: Dict[str, str]  # {section_name: content}
    skills: List[str]
    experiences: List[Dict]

class Application(BaseModel):
    job_id: str
    tailored_resume: str
    cover_letter: str
    status: str  # pending, submitted, failed
    submitted_at: Optional[datetime]
```

Phase 2: Job Discovery Service (Week 2-3)

2.1 Multi-Source Scraper

```
import asyncio
from abc import ABC, abstractmethod
from playwright.async_api import async_playwright

class JobScraperInterface(ABC):
    @abstractmethod
    async def search_jobs(self, query: str, filters: Dict) -> List[JobPosting]:
        pass

class LinkedInScraper(JobScraperInterface):
    async def search_jobs(self, query: str, filters: Dict):
        async with async_playwright() as p:
            browser = await p.chromium.launch()
            # Implementation with anti-detection measures

class IndeedAPIScraper(JobScraperInterface):
    # Use Indeed's API if available, fallback to scraping
```

2.2 Job Relevance Scoring

```
from sentence_transformers import SentenceTransformer
import numpy as np

class JobRelevanceScorer:
    def __init__(self):
        self.model = SentenceTransformer('all-MiniLM-L6-v2')

    def score_relevance(self, job: JobPosting, profile: Dict) -> float:
        job_embedding = self.model.encode(job.description)
        profile_embedding = self.model.encode(profile['summary'])

        # Cosine similarity + keyword matching
        similarity = np.dot(job_embedding, profile_embedding) / (
            np.linalg.norm(job_embedding) * np.linalg.norm(profile_embedding)
        )

        keyword_match_score = self._calculate_keyword_overlap(
            job.keywords, profile['skills']
        )

        return 0.7 * similarity + 0.3 * keyword_match_score
```

Phase 3: Resume Tailoring Service (Week 3-4)

3.1 Resume Parser & Analyzer

```
class ResumeAnalyzer:
    def extract_key_sections(self, resume_text: str) -> Resume:
        # Use spaCy or similar for NER
        # Extract: experience, skills, education, projects

    def identify_transferable_skills(self, resume: Resume, job: JobPosting) -> List[str]:
        # Match skills to job requirements
        # Use semantic similarity for non-exact matches
```

3.2 LLM-Based Resume Tailoring

```
from openai import OpenAI

class ResumeTailor:
    def __init__(self, api_key: str):
        self.client = OpenAI(api_key=api_key)

    async def tailor_resume(self, resume: Resume, job: JobPosting) -> str:
        # Strategic approach:
        # 1. Reorder experiences by relevance
        # 2. Emphasize matching skills
        # 3. Quantify achievements related to job requirements

        prompt = self._build_tailoring_prompt(resume, job)

        response = await self.client.chat.completions.create(
            model="gpt-4",
            messages=[
                {"role": "system", "content": "You are an expert resume writer..."},
                {"role": "user", "content": prompt}
            ],
            temperature=0.3  # Lower temperature for consistency
        )

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

Phase 4: Cover Letter Generation (Week 4)

4.1 Intelligent Cover Letter Generator

```
class CoverLetterGenerator:
    def __init__(self, api_key: str):
        self.client = OpenAI(api_key=api_key)

    async def generate_cover_letter(
        self,
        job: JobPosting,
        resume: Resume,
```

```
        tone: str = "professional_friendly"
    ) -> str:
        # Extract company culture indicators from job posting
        culture_hints = self._analyze_company_tone(job.description)

        # Build structured prompt with:
        # - Specific achievements that match requirements
        # - Company-specific interest points
        # - Appropriate tone matching

        prompt = f"""
        Generate a cover letter for {job.title} at {job.company}.

        Key requirements to address:
        {json.dumps(job.requirements[:3])}

        Relevant achievements:
        {self._extract_relevant_achievements(resume, job)}

        Tone: {tone} matching company culture: {culture_hints}

        Constraints:
        - Maximum 300 words
        - No generic statements
        - Include specific company/role details
        """

        # Generate with chain-of-thought for better quality
```

Phase 5: Application Submission (Week 5)

5.1 Form Automation Engine

```
class ApplicationSubmitter:
    def __init__(self):
        self.browser_manager = BrowserManager()

    async def submit_application(
        self,
        job: JobPosting,
        resume_path: str,
        cover_letter: str
```

```
    ) -> bool:
        # Strategy pattern for different application systems
        if "workday" in job.application_url:
            return await self._submit_workday(job, resume_path, cover_letter)
        elif "greenhouse" in job.application_url:
            return await self._submit_greenhouse(job, resume_path, cover_letter)
        else:
            return await self._submit_generic(job, resume_path, cover_letter)

    async def _submit_generic(self, job, resume_path, cover_letter):
        page = await self.browser_manager.get_page()

        # Intelligent form field detection
        await page.goto(job.application_url)

        # Use computer vision or DOM analysis to identify:
        # - Resume upload button
        # - Cover letter field
        # - Required fields

        form_analyzer = FormAnalyzer(page)
        fields = await form_analyzer.detect_application_fields()

        # Fill and submit
```

Phase 6: Orchestration & Monitoring (Week 5-6)

6.1 Workflow Orchestrator

```
from celery import Celery
from celery.exceptions import MaxRetriesExceededError

class JobApplicationWorkflow:
    def __init__(self):
        self.celery = Celery('job_app', broker='redis://localhost:6379')

    @celery.task(bind=True, max_retries=3)
    def process_job_application(self, job_id: str, user_profile: Dict):
        try:
            # 1. Fetch job details
            job = JobService.get_job(job_id)
```

```
            # 2. Tailor resume
            tailored_resume = ResumeTailor.tailor(
                user_profile['resume'], job
            )

            # 3. Generate cover letter
            cover_letter = CoverLetterGenerator.generate(
                job, tailored_resume
            )

            # 4. Submit application
            success = ApplicationSubmitter.submit(
                job, tailored_resume, cover_letter
            )

            # 5. Update tracking
            ApplicationTracker.update_status(
                job_id, "submitted" if success else "failed"
            )

        except Exception as e:
            self.retry(countdown=60)
```

6.2 Rate Limiting & Anti-Detection

```
class RateLimiter:
    def __init__(self):
        self.redis = redis.Redis()

    async def check_rate_limit(self, source: str) -> bool:
        key = f"rate_limit:{source}"
        current = await self.redis.incr(key)

        if current == 1:
            await self.redis.expire(key, 3600)  # 1 hour window

        return current <= self.get_limit_for_source(source)

class AntiDetection:
    @staticmethod
    def randomize_browser_fingerprint(page):
        # Rotate user agents
```

```
        # Add random delays
        # Simulate human-like mouse movements
        # Use residential proxies for high-value targets
```

Phase 7: MVP Testing & Deployment (Week 6)

7.1 Testing Strategy

```
# Unit tests for each service
# Integration tests for workflow
# Mock external services for testing

class TestResumeTailor:
    def test_keyword_matching(self):
        # Test skill extraction and matching

    def test_experience_reordering(self):
        # Verify relevant experiences are prioritized
```

7.2 Deployment Configuration

```
# docker-compose.yml
version: '3.8'
services:
  api:
    build: .
    environment:
      - OPENAI_API_KEY=${OPENAI_API_KEY}
      - DATABASE_URL=postgresql://...

  redis:
    image: redis:alpine

  celery:
    build: .
    command: celery -A orchestration.celery worker

  postgres:
```

```
image: postgres:14
```

Key Technical Considerations

1. **API Rate Limits**: Implement exponential backoff and quota management for all external APIs
2. **Data Privacy**: Encrypt stored resumes and personal data, implement data retention policies
3. **Scalability**: Design for horizontal scaling of scraping and submission services
4. **Monitoring**: Implement comprehensive logging and alerting for failed applications
5. **Legal Compliance**: Respect robots.txt, implement CAPTCHA handling ethically

MVP Success Metrics

- Successfully discover 50+ relevant jobs per search
- Achieve 80%+ relevance score for tailored resumes
- Generate cover letters in <30 seconds
- Successfully submit to at least 3 major ATS platforms
- Process 10 applications per hour per user

This MVP provides a solid foundation that can be extended with features like interview scheduling, application tracking, and success rate analytics.