

# Comprehensive Analysis and System Architecture Plan

After thoroughly analyzing both GitHub repositories and the existing progress on the project, I can provide an in-depth assessment and strategic plan for moving forward with your job application automation and resume customization system.

## Current Codebase Analysis

### JobSpy Repository Strengths

- Robust multi-platform scraping (LinkedIn, Indeed, ZipRecruiter, Glassdoor, Google, Bayt)
- Well-structured scraper classes with consistent interfaces
- Comprehensive job metadata extraction
- Effective proxy and rate-limiting management
- Modular design with clear separation of concerns

### Auto\_job\_applier\_linkedin Repository Issues

Several critical issues need addressing before integration:

#### 1. Browser Automation Fragility:

- Heavy reliance on XPath and CSS selectors that break with LinkedIn UI changes
- Insufficient wait mechanisms (using static `sleep()` calls instead of explicit waits)
- The stealth mode implementation (`undetected_chromedriver`) may be detected by modern anti-bot systems

#### 2. Code Architecture Problems:

- Monolithic design with the main script (`runAiBot.py`) handling multiple responsibilities
- Configuration scattered across 5+ files without validation
- Global variables used extensively instead of proper state management
- Limited error recovery mechanisms (crashes on unexpected UI elements)

#### 3. Implementation Concerns:

- AI integration appears incomplete, with inconsistent client initialization
- Resume customization functionality is still in development
- Hard dependencies on specific Chrome browser versions
- Selenium usage doesn't leverage modern browser automation capabilities

#### 4. Resume Generation Implementation:

- The current test implementation (`test_with_job_posting.py`) shows potential but needs refinement
- Format extraction from existing resumes needs better error handling
- ReportLab template engine requires more robust styling system

## System Requirements and Constraints

### Functional Requirements

#### 1. Job Scraping:

- Extract complete job details from multiple platforms
- Support filtering by location, job type, keywords, etc.
- Handle pagination and rate limiting
- Detect and skip duplicate jobs across platforms

#### 2. Data Storage:

- Store job data in MongoDB with efficient indexing
- Maintain professional profile with detailed experience data
- Track application history with status and outcomes
- Store generated resumes with version control

#### 3. Resume Generation:

- Analyze job descriptions to extract key requirements
- Match user's professional experience to job requirements
- Generate tailored resumes with consistent formatting
- Support multiple resume templates/styles

#### 4. Application Automation:

- Navigate to job postings via stored URLs
- Fill application forms adaptively based on form structure
- Upload appropriate custom resume
- Handle multi-step application flows
- Track application status

### Non-Functional Requirements

#### 1. Performance:

- Scrape 100+ jobs in under 10 minutes

- Generate resumes in under 30 seconds each
- Complete application process in under 2 minutes per job

## 2. Reliability:

- 95%+ success rate for job scraping
- 98%+ formatting consistency for generated resumes
- 90%+ success rate for automated applications

## 3. Maintainability:

- Modular architecture with clear component boundaries
- Comprehensive test coverage (80%+)
- Well-documented API contracts
- Consistent coding style and patterns

## 4. Security:

- Secure storage of credentials
- Rate limiting to avoid detection
- Proxy rotation for high-volume scraping
- Data encryption for sensitive information

# Constraints

## 1. Technical:

- Python 3.10+ compatibility
- Local LLM support (ollama/mistral)
- Cross-platform operation
- Limited to web-accessible job boards

## 2. Legal/Ethical:

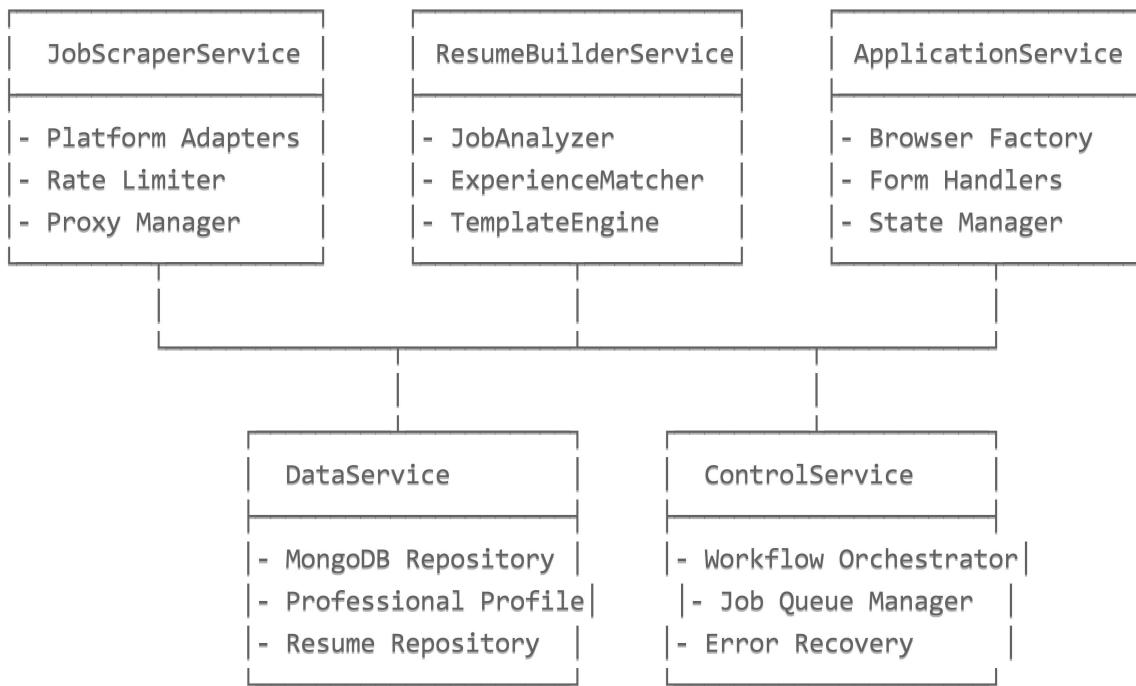
- Comply with job board Terms of Service where possible
- Respect rate limits and robots.txt
- No misrepresentation of user qualifications
- Data privacy compliance

# Core Architectural Decisions

## 1. System Architecture

I recommend a modular, event-driven architecture with the following components:

 Copy



## 2. Technology Choices

Component	Technology	Rationale
Job Scraping	JobSpy (refactored) + httpx	Leverage existing code, improve concurrency
Browser Automation	Playwright	Superior anti-detection, better performance than Selenium
LLM Integration	Provider-agnostic interface	Support both local (ollama) and cloud (OpenAI) options
PDF Generation	ReportLab + PyMuPDF	Pixel-perfect control + format extraction
Database	MongoDB	Flexible schema, good performance for document storage
API	FastAPI	Type safety, async support, automatic documentation
Job Queue	Redis + RQ	Reliable job processing, persistence, monitoring

## 3. Key Design Patterns

1. **Adapter Pattern** for job board integrations
2. **Factory Pattern** for browser and LLM providers
3. **Repository Pattern** for data access
4. **Strategy Pattern** for resume generation approaches
5. **State Pattern** for application flow management

## 6. Observer Pattern for system events and notifications

## 7. Circuit Breaker for resilient external service calls

# Data Models and Relationships

## 1. MongoDB Collections

python

 Copy

```
# Jobs Collection
{
    "_id": ObjectId("..."),
    "platform": "linkedin", # linkedin, indeed, ziprecruiter, etc.
    "platform_id": "12345", # ID from the source platform
    "title": "Senior Software Engineer",
    "company": {
        "name": "Acme Inc.",
        "url": "https://linkedin.com/company/acme",
        "logo_url": "https://..."
    },
    "location": {
        "city": "San Francisco",
        "state": "CA",
        "country": "USA",
        "remote": True
    },
    "description": "Full text description...",
    "requirements": {
        "required_skills": ["Python", "Django", "AWS"],
        "preferred_skills": ["React", "Docker"],
        "years_experience": 5,
        "education": "Bachelor's in Computer Science"
    },
    "compensation": {
        "min": 120000,
        "max": 150000,
        "currency": "USD",
        "interval": "yearly"
    },
    "application": {
        "url": "https://linkedin.com/jobs/view/12345",
        "easy_apply": True,
        "status": "pending", # pending, resume_generated, applied, failed
        "resume_id": ObjectId("..."),
        "applied_at": ISODate("2024-03-01"),
        "match_score": 0.85
    }
}
```

```
        },
        "metadata": {
            "scraped_at": ISODate("2024-03-01"),
            "updated_at": ISODate("2024-03-01"),
            "expires_at": ISODate("2024-04-01")
        }
    }

# Resumes Collection
{
    "_id": ObjectId("..."),
    "job_id": ObjectId("..."),
    "file": {
        "filename": "john_doe_acme_swe_20240301.pdf",
        "path": "/resumes/john_doe_acme_swe_20240301.pdf",
        "content_type": "application/pdf",
        "size": 245678,
        "created_at": ISODate("2024-03-01")
    },
    "customization": {
        "highlighted_skills": ["Python", "Django", "AWS"],
        "highlighted_experiences": [
            {
                "company": "Previous Company",
                "position": "Software Engineer",
                "highlights": ["Led development of...", "Improved system performance by..."]
            }
        ],
        "customized_summary": "Experienced software engineer with...",
        "tailoring_score": 0.85
    },
    "metadata": {
        "created_at": ISODate("2024-03-01"),
        "updated_at": ISODate("2024-03-01"),
        "version": 1
    }
}

# Professional Profile Collection
{
    "_id": ObjectId("..."),
    "user": {
        "name": {
            "first": "John",
            "last": "Doe"
        },
        "profile": {
            "summary": "Software Engineer with over 10 years of experience in Python and Django frameworks. Specialized in system architecture and performance optimization."}
    }
}
```

```
"contact": {
    "email": "john.doe@example.com",
    "phone": "555-123-4567",
    "linkedin": "https://linkedin.com/in/johndoe",
    "location": {
        "city": "San Francisco",
        "state": "CA"
    }
},
"title": "Senior Software Engineer"
},
"skills": [
{
    "name": "Python",
    "years": 5,
    "level": "Expert"
},
{
    "name": "Django",
    "years": 3,
    "level": "Advanced"
}
],
"experiences": [
{
    "company": "Acme Inc.",
    "title": "Senior Software Engineer",
    "start_date": "2020-01",
    "end_date": "Present",
    "description": "Led development of...",
    "achievements": [
        "Reduced system latency by 30%",
        "Implemented CI/CD pipeline"
    ]
}
],
"education": [
{
    "degree": "BS",
    "field": "Computer Science",
    "institution": "Stanford University",
    "graduation_date": "2015"
}
],
"certifications": [
{
```

```
        "name": "AWS Certified Solutions Architect",
        "issuer": "Amazon Web Services",
        "date": "2019"
    },
],
"preferences": {
    "desired_roles": ["Software Engineer", "Tech Lead"],
    "desired_location": ["San Francisco", "Remote"],
    "min_salary": 120000
},
"resume_templates": [
    {
        "name": "Standard",
        "file_path": "/templates/standard.pdf",
        "is_default": true
    }
]
}

# Application History Collection
{
    "_id": ObjectId("..."),
    "job_id": ObjectId("..."),
    "resume_id": ObjectId("..."),
    "status": "completed", # started, completed, failed
    "steps": [
        {
            "name": "navigate_to_job",
            "status": "completed",
            "started_at": ISODate("2024-03-01T10:00:00Z"),
            "completed_at": ISODate("2024-03-01T10:00:05Z"),
            "error": null
        },
        {
            "name": "click_apply",
            "status": "completed",
            "started_at": ISODate("2024-03-01T10:00:05Z"),
            "completed_at": ISODate("2024-03-01T10:00:07Z"),
            "error": null
        }
    ],
    "form_data": {
        "questions_answered": 12,
        "resume_uploaded": true,
        "cover_letter_submitted": false
    },
}
```

```
"started_at": ISODate("2024-03-01T10:00:00Z"),
"completed_at": ISODate("2024-03-01T10:02:30Z"),
"error": null
}
```

## API Contract Design

### 1. Job Scraper API

python

Copy

```
# Job Scraper API
class JobScraperService:
    async def search_jobs(
        self,
        search_terms: List[str],
        location: str = None,
        job_type: str = None,
        experience_level: str = None,
        date_posted: str = None,
        platforms: List[str] = None,
        max_results: int = 100
    ) -> Dict[str, Any]:
        """Search for jobs across multiple platforms"""
        pass

    async def get_job_details(
        self,
        platform: str,
        job_id: str
    ) -> Dict[str, Any]:
        """Get detailed information for a specific job"""
        pass
```

### 2. Resume Builder API

python

Copy

```
# Resume Builder API
class ResumeBuilderService:
    async def analyze_job(
        self,
        job_id: str
    ) -> Dict[str, Any]:
```

```
"""Analyze job description and extract requirements"""
pass

async def generate_resume(
    self,
    job_id: str,
    profile_id: str,
    template_name: str = "default"
) -> Dict[str, Any]:
    """Generate a custom resume for a specific job"""
    pass

async def get_resume(
    self,
    resume_id: str
) -> bytes:
    """Get the resume file content"""
    pass
```

### 3. Application Service API

python

 Copy

```
# Application Service API
class ApplicationService:
    async def apply_to_job(
        self,
        job_id: str,
        resume_id: str
    ) -> Dict[str, Any]:
        """Apply to a job with the specified resume"""
        pass

    async def get_application_status(
        self,
        job_id: str
    ) -> Dict[str, Any]:
        """Get the status of a job application"""
        pass
```

### 4. Profile Management API

python

 Copy

```
# Profile Management API
class ProfileService:
    async def create_profile(
        self,
        profile_data: Dict[str, Any]
    ) -> Dict[str, Any]:
        """Create a new professional profile"""
        pass

    async def update_profile(
        self,
        profile_id: str,
        profile_data: Dict[str, Any]
    ) -> Dict[str, Any]:
        """Update an existing professional profile"""
        pass

    async def get_profile(
        self,
        profile_id: str
    ) -> Dict[str, Any]:
        """Get a professional profile"""
        pass
```

## Security Considerations

### 1. Credential Management

- Use environment variables or a secure vault (like AWS Secrets Manager or HashiCorp Vault)
- Encrypt sensitive data at rest
- Implement role-based access control
- Rotate credentials regularly

### 2. Anti-Detection Measures

- Implement human-like browsing patterns (random delays, mouse movements)
- Rotate IPs and user agents
- Limit request rates to mimic human behavior
- Use browser fingerprint randomization
- Implement session management to maintain cookies

### 3. Data Protection

- Encrypt sensitive data in the database
- Implement proper access control
- Regular security audits
- Secure API endpoints with proper authentication

## 4. Error Handling and Logging

- Don't expose sensitive information in error messages
- Implement proper exception handling
- Log security events for audit
- Rate limit failed login attempts

### Code Review of test\_with\_job\_posting.py

The test file shows a promising approach but has several issues:

1. **Error Handling:** Minimal error handling with only basic try/except blocks.
2. **Configuration:** Hardcoded paths and configuration values.
3. **Validation:** No validation of LLM responses or generated PDFs.
4. **Integration:** Not well integrated with overall system architecture.

### Specific improvements needed:

```
python Copy  
  
# Instead of:  
llm_provider = LLMProviderFactory.get_provider(  
    "ollama",  
    base_url=os.getenv("OLLAMA_BASE_URL", "http://localhost:11434"),  
    model=os.getenv("OLLAMA_MODEL", "llama3.1:8b-instruct")  
)  
  
# Use a proper configuration system:  
config = ConfigurationService.load("resume_generation")  
llm_provider = LLMProviderFactory.get_provider(  
    config.llm.provider,  
    **config.llm.parameters  
)
```

## Next Steps and Implementation Roadmap

## Phase 1: Foundations (2-3 weeks)

1. Set up project structure and CI/CD pipeline
2. Implement MongoDB data models
3. Refactor JobSpy into modular service
4. Build LLM provider abstraction
5. Create base resume template engine

## Phase 2: Core Functionality (3-4 weeks)

1. Implement job analysis service
2. Build experience matching algorithm
3. Develop resume generation pipeline
4. Create application workflow engine
5. Integrate browser automation with Playwright

## Phase 3: Integration and Reliability (2-3 weeks)

1. Connect all services through workflow orchestrator
2. Implement extensive error handling and recovery
3. Add monitoring and logging
4. Build user interface for configuration
5. Develop testing suite for all components

## Phase 4: Enhancement and Optimization (2-3 weeks)

1. Optimize performance bottlenecks
2. Add additional job platforms
3. Enhance resume customization with more templates
4. Implement advanced analytics on applications
5. Add user notification system

## Specific Testing Recommendations

Before proceeding with `test_with_job_posting.py`, I recommend:

### 1. Fix the LLM Provider Interface:

- Make error handling more robust
- Add retry mechanisms

- Better validate responses

## 2. Enhance the Resume Template Engine:

- Add validation for generated PDFs
- Implement multiple template styles
- More robust font handling

## 3. Improve the Experience Matcher:

- Add scoring for skill matches
- Better highlight relevant experiences
- Implement keyword optimization

Would you like me to provide more specific code changes for any of these components? I can also develop a more detailed implementation plan for a specific part of the system.

**MR** Lets start with these items:

- Set up project structure and CI/CD pipeline
- Implement MongoDB data models
- Refactor JobSpy into modular service
- Build LLM provider abstraction
- Create base resume template engine

Architected comprehensive project blueprint for AI-driven resume system

🕒 4s ↗

## Implementation Plan for Phase 1: Foundations

I'll provide detailed implementation plans for each of the foundational components you want to focus on.

### 1. Project Structure and CI/CD Pipeline

#### Directory Structure

Copy

```
job-application-automation/
└── .github/
    └── workflows/
        └── ci.yml
```

```
|   └── deploy.yml
├── app/
│   ├── __init__.py
│   ├── config.py          # Configuration management
│   ├── core/
│   │   ├── __init__.py
│   │   ├── models.py        # Pydantic models
│   │   ├── exceptions.py   # Custom exceptions
│   │   └── logging.py       # Logging setup
│   ├── db/                 # Database access
│   │   ├── __init__.py
│   │   ├── mongodb.py      # MongoDB client
│   │   └── repositories/   # Repository pattern implementation
│   │       ├── __init__.py
│   │       ├── jobs.py
│   │       ├── resumes.py
│   │       └── profiles.py
│   ├── services/           # Business logic
│   │   ├── __init__.py
│   │   ├── job_scraper/    # Refactored JobSpy
│   │   │   ├── __init__.py
│   │   │   ├── linkedin.py
│   │   │   ├── indeed.py
│   │   │   └── ...
│   │   ├── llm/             # LLM abstraction
│   │   │   ├── __init__.py
│   │   │   ├── provider.py
│   │   │   ├── openai.py
│   │   │   └── ollama.py
│   │   ├── resume_builder/ # Resume generation
│   │   │   ├── __init__.py
│   │   │   ├── analyzer.py
│   │   │   ├── matcher.py
│   │   │   └── template.py
│   │   └── application/   # Job application
│   │       ├── __init__.py
│   │       ├── browser.py
│   │       └── workflow.py
│   └── utils/              # Utility functions
│       ├── __init__.py
│       ├── proxies.py
│       └── rate_limiter.py
└── tests/                # Tests
    ├── __init__.py
    ├── conftest.py
    └── test_job_scraper.py
```

```
|   └── test_llm.py
|   └── ...
|── scripts/           # Utility scripts
|   ├── setup_db.py
|   └── import_profile.py
|── templates/          # Resume templates
|   ├── standard.yaml
|   └── fonts/
|── resumes/            # Generated resumes
|── .env.example        # Example environment variables
|── .gitignore
|── docker-compose.yml  # For local development
|── Dockerfile
|── pyproject.toml      # Project metadata and dependencies
|── README.md
└── requirements.txt    # Dependencies
```

## CI/CD Pipeline with GitHub Actions

yaml

 Copy

```
# .github/workflows/ci.yml
name: CI Pipeline

on:
  push:
    branches: [ main, dev ]
  pull_request:
    branches: [ main, dev ]

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.10'
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install flake8 black isort mypy
          pip install -r requirements.txt
      - name: Lint with flake8
```

```

run: flake8 app tests
- name: Check formatting with black
  run: black --check app tests
- name: Check imports with isort
  run: isort --check-only --profile black app tests
- name: Type check with mypy
  run: mypy app tests

test:
  runs-on: ubuntu-latest
  services:
    mongodb:
      image: mongo:4.4
      ports:
        - 27017:27017
  steps:
    - uses: actions/checkout@v2
    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: '3.10'
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install pytest pytest-cov
        pip install -r requirements.txt
    - name: Test with pytest
      run: pytest --cov=app tests/
    - name: Upload coverage toCodecov
      uses: codecov/codecov-action@v1

```

## Environment Setup

yaml

 Copy

```

# .env.example
# MongoDB
MONGODB_URI=mongodb://localhost:27017
MONGODB_DB=job_automation

# LLM Providers
LLM_PROVIDER=ollama # or openai
OLLAMA_BASE_URL=http://localhost:11434
OLLAMA_MODEL=llama3.1:8b-instruct
OPENAI_API_KEY=sk-...

```

```
# Job Scraping
MAX_JOBS_PER_PLATFORM=50
USE_PROXYIES=false
PROXY_LIST=

# Application
HEADLESS_BROWSER=true
RESUME_OUTPUT_DIR=./resumes
TEMPLATE_DIR=./templates
```

## 2. MongoDB Data Models

First, let's create Pydantic models for validation and MongoDB repositories for data access.

### Pydantic Models ([app/core/models.py](#))

python

 Copy

```
from datetime import datetime
from enum import Enum
from typing import List, Optional, Dict, Any, Union
from pydantic import BaseModel, Field, HttpUrl

class JobStatus(str, Enum):
    PENDING = "pending"
    ANALYZED = "analyzed"
    RESUME_GENERATED = "resume_generated"
    APPLIED = "applied"
    FAILED = "failed"

class JobPlatform(str, Enum):
    LINKEDIN = "linkedin"
    INDEED = "indeed"
    ZIPRECRUITER = "ziprecruiter"
    GLASSDOOR = "glassdoor"
    GOOGLE = "google"

class CompanyModel(BaseModel):
    name: str
    url: Optional[HttpUrl] = None
    logo_url: Optional[HttpUrl] = None

class LocationModel(BaseModel):
    city: Optional[str] = None
    state: Optional[str] = None
```

```
country: Optional[str] = None
remote: bool = False

class RequirementsModel(BaseModel):
    required_skills: List[str] = Field(default_factory=list)
    preferred_skills: List[str] = Field(default_factory=list)
    years_experience: Optional[int] = None
    education: Optional[str] = None

class CompensationModel(BaseModel):
    min: Optional[int] = None
    max: Optional[int] = None
    currency: str = "USD"
    interval: str = "yearly"

class ApplicationModel(BaseModel):
    url: HttpUrl
    easy_apply: bool = False
    status: JobStatus = JobStatus.PENDING
    resume_id: Optional[str] = None
    applied_at: Optional[datetime] = None
    match_score: Optional[float] = None

class JobMetadataModel(BaseModel):
    scraped_at: datetime = Field(default_factory=datetime.utcnow)
    updated_at: datetime = Field(default_factory=datetime.utcnow)
    expires_at: Optional[datetime] = None

class JobModel(BaseModel):
    id: Optional[str] = None
    platform: JobPlatform
    platform_id: str
    title: str
    company: CompanyModel
    location: LocationModel
    description: str
    requirements: RequirementsModel = Field(default_factory=RequirementsModel)
    compensation: Optional[CompensationModel] = None
    application: ApplicationModel
    metadata: JobMetadataModel = Field(default_factory=JobMetadataModel)

class Config:
    populate_by_name = True
    json_encoders = {
        datetime: lambda v: v.isoformat()
    }
```

## MongoDB Repository ( [app/db/repositories/jobs.py](#) )

python

Copy

```
from typing import List, Optional, Dict, Any
from datetime import datetime
from pymongo import MongoClient, ASCENDING, DESCENDING
from bson.objectid import ObjectId

from app.core.models import JobModel, JobStatus
from app.db.mongodb import get_database

class JobRepository:
    def __init__(self):
        self.db = get_database()
        self.collection = self.db.jobs

        # Create indexes for efficient queries
        self.collection.create_index([("platform", ASCENDING), ("platform_id", ASCENDING)],
        self.collection.create_index([("application.status", ASCENDING)])
        self.collection.create_index([("metadata.scraped_at", DESCENDING)])

    async def create(self, job: JobModel) -> str:
        """Insert a new job and return its ID"""
        job_dict = job.dict(exclude={'id'})

        # Convert datetime objects to proper format
        for key in ['scraped_at', 'updated_at', 'expires_at']:
            if job_dict['metadata'].get(key):
                job_dict['metadata'][key] = job_dict['metadata'][key]

        result = await self.collection.insert_one(job_dict)
        return str(result.inserted_id)

    async def get_by_id(self, job_id: str) -> Optional[JobModel]:
        """Get a job by ID"""
        result = await self.collection.find_one({"_id": ObjectId(job_id)})
        if result:
            result['id'] = str(result.pop('_id'))
            return JobModel(**result)
        return None

    async def update(self, job_id: str, update_data: Dict[str, Any]) -> bool:
        """Update a job document"""
        update_data['metadata.updated_at'] = datetime.utcnow()
```

```

        result = await self.collection.update_one(
            {"_id": ObjectId(job_id)},
            {"$set": update_data}
        )
        return result.modified_count > 0

    async def get_by_status(self, status: JobStatus, limit: int = 100) -> List[JobModel]:
        """Get jobs with a specific status"""
        cursor = self.collection.find({"application.status": status}).limit(limit)
        jobs = []
        async for document in cursor:
            document['id'] = str(document.pop('_id'))
            jobs.append(JobModel(**document))
        return jobs

    async def get_by_platform_id(self, platform: str, platform_id: str) -> Optional[JobMode
        """Get a job by platform and platform-specific ID"""
        result = await self.collection.find_one({
            "platform": platform,
            "platform_id": platform_id
        })
        if result:
            result['id'] = str(result.pop('_id'))
            return JobModel(**result)
        return None

```

Similar repositories would be created for Resumes, Profiles, and ApplicationHistory.

### 3. Refactor JobSpy into Modular Service

JobSpy is already well-structured, but we'll refactor it to fit our architecture and improve its maintainability.

#### Scraper Interface ( `app/services/job_scraper/__init__.py` )

python

 Copy

```

from abc import ABC, abstractmethod
from typing import List, Dict, Any, Optional
from app.core.models import JobModel

class JobScraperInterface(ABC):
    """Base interface for all job scrapers"""

    @abstractmethod

```

```
async def search_jobs(
    self,
    search_terms: List[str],
    location: Optional[str] = None,
    job_type: Optional[str] = None,
    experience_level: Optional[str] = None,
    date_posted: Optional[str] = None,
    max_results: int = 50
) -> List[JobModel]:
    """Search for jobs and return standardized job models"""
    pass

@abstractmethod
async def get_job_details(self, job_id: str) -> Optional[JobModel]:
    """Get detailed information for a specific job"""
    pass

class JobScraperFactory:
    """Factory for creating job scrapers"""

    @staticmethod
    def get_scraper(platform: str, **kwargs) -> JobScraperInterface:
        """Get a job scraper for the specified platform"""
        if platform.lower() == "linkedin":
            from app.services.job_scraper.linkedin import LinkedInScraper
            return LinkedInScraper(**kwargs)
        elif platform.lower() == "indeed":
            from app.services.job_scraper.indeed import IndeedScraper
            return IndeedScraper(**kwargs)
        elif platform.lower() == "ziprecruiter":
            from app.services.job_scraper.ziprecruiter import ZipRecruiterScraper
            return ZipRecruiterScraper(**kwargs)
        elif platform.lower() == "glassdoor":
            from app.services.job_scraper.glassdoor import GlassdoorScraper
            return GlassdoorScraper(**kwargs)
        elif platform.lower() == "google":
            from app.services.job_scraper.google import GoogleJobsScraper
            return GoogleJobsScraper(**kwargs)
        else:
            raise ValueError(f"Unsupported platform: {platform}")
```

## LinkedIn Scraper Example ( [app/services/job\\_scraper/linkedin.py](#) )

python

 Copy

```
import asyncio
from typing import List, Dict, Any, Optional
import httpx
from bs4 import BeautifulSoup

from app.core.models import JobModel, JobPlatform, CompanyModel, LocationModel, Application
from app.services.job_scraper import JobScraperInterface
from app.utils.rate_limiter import RateLimiter
from app.utils.proxies import ProxyManager

class LinkedInScraper(JobScraperInterface):
    def __init__(
        self,
        use_proxies: bool = False,
        rate_limit: int = 1, # requests per second
    ):
        self.base_url = "https://www.linkedin.com"
        self.headers = {
            "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/110.0.5380.126 Safari/537.36",
            "Accept-Language": "en-US,en;q=0.9",
        }
        self.rate_limiter = RateLimiter(rate_limit)
        self.proxy_manager = ProxyManager() if use_proxies else None

    async def search_jobs(
        self,
        search_terms: List[str],
        location: Optional[str] = None,
        job_type: Optional[str] = None,
        experience_level: Optional[str] = None,
        date_posted: Optional[str] = None,
        max_results: int = 50
    ) -> List[JobModel]:
        """Search for jobs on LinkedIn"""
        all_jobs = []

        for term in search_terms:
            # Build the search URL with parameters
            url = f"{self.base_url}/jobs/search/?"
            params = {"keywords": term}

            if location:
                params["location"] = location

            if job_type:
```

```
job_type_map = {
    "fulltime": "F",
    "parttime": "P",
    "contract": "C",
    "internship": "I"
}
if job_type.lower() in job_type_map:
    params["f_JT"] = job_type_map[job_type.lower()]

if experience_level:
    exp_level_map = {
        "internship": "1",
        "entry": "2",
        "associate": "3",
        "mid-senior": "4",
        "director": "5",
        "executive": "6"
    }
    if experience_level.lower() in exp_level_map:
        params["f_E"] = exp_level_map[experience_level.lower()]

if date_posted:
    date_map = {
        "24h": "r86400",
        "week": "r604800",
        "month": "r2592000"
    }
    if date_posted.lower() in date_map:
        params["f_TPR"] = date_map[date_posted.lower()]

# Start iterating through pages
start = 0
while len(all_jobs) < max_results:
    params["start"] = start

    # Apply rate limiting
    await self.rate_limiter.wait()

    # Get proxy if using proxies
    proxy = self.proxy_manager.get_proxy() if self.proxy_manager else None

try:
    async with httpx.AsyncClient(proxies=proxy) as client:
        response = await client.get(
            url,
            params=params,
```

```
        headers=self.headers,
        follow_redirects=True,
        timeout=30.0
    )
    response.raise_for_status()

    # Parse the HTML
    jobs = self._parse_search_results(response.text)

    if not jobs:
        break # No more jobs found

    # Add jobs to results
    all_jobs.extend(jobs)

    # Move to next page
    start += len(jobs)

except Exception as e:
    # Log the error and continue
    print(f"Error searching LinkedIn jobs: {e}")
    break

# Limit the number of results
return all_jobs[:max_results]

async def get_job_details(self, job_id: str) -> Optional[JobModel]:
    """Get detailed information for a specific LinkedIn job"""
    url = f"{self.base_url}/jobs/view/{job_id}"

    # Apply rate limiting
    await self.rate_limiter.wait()

    # Get proxy if using proxies
    proxy = self.proxy_manager.get_proxy() if self.proxy_manager else None

try:
    async with httpx.AsyncClient(proxies=proxy) as client:
        response = await client.get(
            url,
            headers=self.headers,
            follow_redirects=True,
            timeout=30.0
        )
        response.raise_for_status()
```

```
# Parse the HTML to extract job details
    return self._parse_job_details(response.text, job_id)

except Exception as e:
    # Log the error
    print(f"Error getting LinkedIn job details: {e}")
    return None

def _parse_search_results(self, html: str) -> List[Dict[str, Any]]:
    """Parse LinkedIn job search results HTML"""
    soup = BeautifulSoup(html, 'html.parser')
    job_cards = soup.select('div.base-search-card')

    jobs = []
    for card in job_cards:
        try:
            # Extract job info
            title_tag = card.select_one('h3.base-search-card__title')
            company_tag = card.select_one('h4.base-search-card__subtitle')
            location_tag = card.select_one('span.job-search-card__location')
            link_tag = card.select_one('a.base-card__full-link')

            if title_tag and company_tag and location_tag and link_tag:
                title = title_tag.text.strip()
                company = company_tag.text.strip()
                location_text = location_tag.text.strip()
                link = link_tag['href']

                # Extract job ID from link
                job_id = link.split('-')[-1].split('?')[0]

                # Parse location
                location_parts = location_text.split(', ')
                city = location_parts[0] if len(location_parts) > 0 else None
                state = location_parts[1] if len(location_parts) > 1 else None
                country = location_parts[2] if len(location_parts) > 2 else None

                # Create job model
                job = JobModel(
                    platform=JobPlatform.LINKEDIN,
                    platform_id=job_id,
                    title=title,
                    company=CompanyModel(name=company),
                    location=LocationModel(
                        city=city,
                        state=state,
```

```
        country=country,
        remote='Remote' in location_text
    ),
    description="", # Will be filled in get_job_details
    application=ApplicationModel(url=link)
)

jobs.append(job)
except Exception as e:
    # Log parsing error and continue
    print(f"Error parsing LinkedIn job card: {e}")

return jobs

def _parse_job_details(self, html: str, job_id: str) -> Optional[JobModel]:
    """Parse LinkedIn job details HTML"""
    soup = BeautifulSoup(html, 'html.parser')

    try:
        # Extract job info
        title_tag = soup.select_one('h1.job-title')
        company_tag = soup.select_one('a.company-name')
        location_tag = soup.select_one('span.location')
        description_tag = soup.select_one('div.description_text')

        if title_tag and company_tag and location_tag and description_tag:
            title = title_tag.text.strip()
            company = company_tag.text.strip()
            company_url = company_tag['href'] if 'href' in company_tag.attrs else None
            location_text = location_tag.text.strip()
            description = description_tag.text.strip()

            # Parse location
            location_parts = location_text.split(', ')
            city = location_parts[0] if len(location_parts) > 0 else None
            state = location_parts[1] if len(location_parts) > 1 else None
            country = location_parts[2] if len(location_parts) > 2 else None

            # Extract job requirements
            required_skills = []
            preferred_skills = []
            years_experience = None
            education = None

            # Look for common patterns in job descriptions
            # ...
    
```

```

# Create job model
job = JobModel(
    platform=JobPlatform.LINKEDIN,
    platform_id=job_id,
    title=title,
    company=CompanyModel(
        name=company,
        url=company_url
    ),
    location=LocationModel(
        city=city,
        state=state,
        country=country,
        remote='Remote' in location_text
    ),
    description=description,
    requirements=RequirementsModel(
        required_skills=required_skills,
        preferred_skills=preferred_skills,
        years_experience=years_experience,
        education=education
    ),
    application=ApplicationModel(
        url=f"{self.base_url}/jobs/view/{job_id}",
        easy_apply=bool(soup.select_one('button.jobs-apply-button'))
    )
)

return job
except Exception as e:
    # Log parsing error
    print(f"Error parsing LinkedIn job details: {e}")

return None

```

## 4. Build LLM Provider Abstraction

Now let's create a flexible abstraction for working with different LLM providers.

### LLM Provider Interface ([app/services/llm/provider.py](#))

python

 Copy

```
from abc import ABC, abstractmethod
from typing import Dict, Any, Optional, List, Union

class LLMPromoter(ABC):
    """Abstract base class for LLM providers"""

    @abstractmethod
    async def generate(
        self,
        prompt: str,
        system_message: Optional[str] = None,
        temperature: float = 0.2,
        max_tokens: Optional[int] = None
    ) -> str:
        """Generate text from the LLM"""
        pass

    @abstractmethod
    async def generate_structured(
        self,
        prompt: str,
        output_schema: Dict[str, Any],
        system_message: Optional[str] = None,
        temperature: float = 0.1
    ) -> Dict[str, Any]:
        """Generate structured data from the LLM"""
        pass

class LLMPromoterFactory:
    """Factory for creating LLM providers"""

    @staticmethod
    def get_provider(provider_type: str, **kwargs) -> LLMPromoter:
        """Get an LLM provider of the specified type"""
        if provider_type.lower() == "openai":
            from app.services.llm.openai import OpenAIProvider
            return OpenAIProvider(**kwargs)
        elif provider_type.lower() == "ollama":
            from app.services.llm.ollama import OllamaProvider
            return OllamaProvider(**kwargs)
        else:
            raise ValueError(f"Unsupported LLM provider: {provider_type}")
```

## OpenAI Provider Implementation ( [app/services/llm/openai.py](#) )

python

 Copy

```
import json
from typing import Dict, Any, Optional, List, Union
import httpx

from app.services.llm.provider import LLMPProvider
from app.core.exceptions import LLMPProviderError

class OpenAIPrompter(LLMPProvider):
    """Implementation for OpenAI API"""

    def __init__(
        self,
        api_key: str,
        model: str = "gpt-4o-mini",
        base_url: str = "https://api.openai.com/v1"
    ):
        self.api_key = api_key
        self.model = model
        self.base_url = base_url

    async def generate(
        self,
        prompt: str,
        system_message: Optional[str] = None,
        temperature: float = 0.2,
        max_tokens: Optional[int] = None
    ) -> str:
        """Generate text from OpenAI API"""
        messages = []

        if system_message:
            messages.append({"role": "system", "content": system_message})

        messages.append({"role": "user", "content": prompt})

        payload = {
            "model": self.model,
            "messages": messages,
            "temperature": temperature
        }

        if max_tokens:
            payload["max_tokens"] = max_tokens
```

```
headers = {
    "Content-Type": "application/json",
    "Authorization": f"Bearer {self.api_key}"
}

try:
    async with httpx.AsyncClient() as client:
        response = await client.post(
            f"{self.base_url}/chat/completions",
            headers=headers,
            json=payload,
            timeout=60.0
        )

        if response.status_code != 200:
            raise LLMProviderError(f"OpenAI API error: {response.status_code} - {re
                result = response.json()
                return result["choices"][0]["message"]["content"]

except httpx.TimeoutException:
    raise LLMProviderError("OpenAI API request timed out")
except httpx.RequestError as e:
    raise LLMProviderError(f"OpenAI API request error: {str(e)}")
except Exception as e:
    raise LLMProviderError(f"Error generating text from OpenAI: {str(e)}")

async def generate_structured(
    self,
    prompt: str,
    output_schema: Dict[str, Any],
    system_message: Optional[str] = None,
    temperature: float = 0.1
) -> Dict[str, Any]:
    """Generate structured JSON data from OpenAI API"""
    # Add schema to system message
    schema_json = json.dumps(output_schema, indent=2)

    enhanced_system = f"""
{system_message if system_message else ""}

You must respond with a valid JSON object matching this schema:
{schema_json}

Provide ONLY the JSON in your response, no other text.
"""

    
```

```
# Update the prompt to request JSON
enhanced_prompt = f"{prompt}\n\nRespond with valid JSON matching the specified schema"

# Make the API call with function calling
messages = [
    {"role": "system", "content": enhanced_system},
    {"role": "user", "content": enhanced_prompt}
]

payload = {
    "model": self.model,
    "messages": messages,
    "temperature": temperature,
    "response_format": {"type": "json_object"}
}

headers = {
    "Content-Type": "application/json",
    "Authorization": f"Bearer {self.api_key}"
}

try:
    async with httpx.AsyncClient() as client:
        response = await client.post(
            f"{self.base_url}/chat/completions",
            headers=headers,
            json=payload,
            timeout=60.0
        )

        if response.status_code != 200:
            raise LLMProviderError(f"OpenAI API error: {response.status_code} - {response.text}")

        result = response.json()
        content = result["choices"][0]["message"]["content"]

        # Parse the JSON
        try:
            return json.loads(content)
        except json.JSONDecodeError:
            raise LLMProviderError(f"Failed to parse JSON from OpenAI response: {content}")

except httpx.TimeoutException:
    raise LLMProviderError("OpenAI API request timed out")
except httpx.RequestError as e:
```

```
raise LLMPromptError(f"OpenAI API request error: {str(e)}")
except Exception as e:
    raise LLMPromptError(f"Error generating structured data from OpenAI: {str(e)}
```

## Ollama Provider Implementation ( [app/services/llm/ollama.py](#) )

python

Copy

```
import json
import re
from typing import Dict, Any, Optional, List, Union
import httpx

from app.services.llm.provider import LLMPrompt
from app.core.exceptions import LLMPromptError

class OllamaProvider(LLMPrompt):
    """Implementation for Ollama API"""

    def __init__(
        self,
        base_url: str = "http://localhost:11434",
        model: str = "llama3.1:8b-instruct"
    ):
        self.base_url = base_url
        self.model = model

    async def generate(
        self,
        prompt: str,
        system_message: Optional[str] = None,
        temperature: float = 0.2,
        max_tokens: Optional[int] = None
    ) -> str:
        """Generate text from Ollama API"""
        payload = {
            "model": self.model,
            "prompt": prompt,
            "temperature": temperature,
            "system": system_message if system_message else "",
            "stream": False
        }

        if max_tokens:
```

```

payload["num_predict"] = max_tokens

try:
    async with httpx.AsyncClient() as client:
        response = await client.post(
            f"{self.base_url}/api/generate",
            json=payload,
            timeout=60.0
        )

        if response.status_code != 200:
            raise LLMPromptError(f"Ollama API error: {response.status_code} - {re
                result = response.json()
                return result["response"]

except httpx.TimeoutException:
    raise LLMPromptError("Ollama API request timed out")
except httpx.RequestError as e:
    raise LLMPromptError(f"Ollama API request error: {str(e)}")
except Exception as e:
    raise LLMPromptError(f"Error generating text from Ollama: {str(e)}")

async def generate_structured(
    self,
    prompt: str,
    output_schema: Dict[str, Any],
    system_message: Optional[str] = None,
    temperature: float = 0.1,
    max_retries: int = 3
) -> Dict[str, Any]:
    """Generate structured JSON data from Ollama API"""
    schema_json = json.dumps(output_schema, indent=2)

    enhanced_system = f"""
{system_message if system_message else ""}

You must respond with a valid JSON object matching this schema:
{schema_json}

Your response must contain ONLY the JSON object, no other text.
Do not wrap the JSON in markdown code blocks or any other formatting.
Ensure all JSON keys and values match the schema exactly.
"""

    enhanced_prompt = f"""

```

{prompt}

Remember to respond ONLY with a valid JSON object matching the specified schema.

"""

```
for attempt in range(max_retries):
    try:
        response_text = await self.generate(
            prompt=enhanced_prompt,
            system_message=enhanced_system,
            temperature=temperature
        )

        # Extract JSON from response
        json_match = re.search(r'(\{.*\})', response_text, re.DOTALL)
        if json_match:
            json_str = json_match.group(1)
        else:
            # Try to find JSON without the outer braces pattern
            json_start = response_text.find('{')
            json_end = response_text.rfind('}') + 1

            if json_start >= 0 and json_end > json_start:
                json_str = response_text[json_start:json_end]
            else:
                raise LLMProviderError("No JSON object found in response")
```

# Parse the JSON

```
try:
    return json.loads(json_str)
except json.JSONDecodeError as e:
    if attempt < max_retries - 1:
        # Try again with more explicit instructions
        enhanced_prompt = f"""
{prompt}
```

The previous response contained invalid JSON: {str(e)}

Please respond ONLY with a valid JSON object matching the specified  
Do not include any other text, markdown formatting, or code blocks.

"""

continue

else:

```
raise LLMProviderError(f"Failed to parse JSON after {max_retries} a
```

```
except Exception as e:
```

```
if attempt < max_retries - 1:  
    continue  
else:  
    raise LLMProviderError(f"Error generating structured data from Ollama a
```

## 5. Create Base Resume Template Engine

Finally, let's implement a flexible resume template engine using ReportLab.

### Resume Template Engine ( `app/services/resume_builder/template.py` )

python

 Copy

```
import os  
import json  
from datetime import datetime  
from typing import Dict, Any, List, Optional  
  
from reportlab.lib.pagesizes import LETTER  
from reportlab.lib import colors  
from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle  
from reportlab.platypus import SimpleDocTemplate, Paragraph, Spacer, Table, TableStyle  
from reportlab.lib.enums import TA_LEFT, TA_CENTER  
from reportlab.lib.units import inch  
from reportlab.pdfbase import pdfmetrics  
from reportlab.pdfbase.ttfonts import TTFont  
  
from app.core.exceptions import TemplateEngineError  
  
class ResumeTemplateEngine:  
    """Engine for generating résumés with consistent formatting"""  
  
    def __init__(  
        self,  
        template_dir: str = "./templates",  
        output_dir: str = "./resumes",  
        default_template: str = "standard"  
    ):  
        self.template_dir = template_dir  
        self.output_dir = output_dir  
        self.default_template = default_template  
  
        # Ensure directories exist  
        os.makedirs(self.template_dir, exist_ok=True)  
        os.makedirs(self.output_dir, exist_ok=True)
```

```
# Register fonts
self._register_fonts()

# Load styles
self.styles = getSampleStyleSheet()
self._setup_styles()

def _register_fonts(self):
    """Register fonts for use in PDFs"""
    # Define font paths
    font_dir = os.path.join(self.template_dir, "fonts")
    os.makedirs(font_dir, exist_ok=True)

    calibri_path = os.path.join(font_dir, "calibri.ttf")
    calibri_bold_path = os.path.join(font_dir, "calibrib.ttf")

    # Register fonts if they exist
    try:
        if os.path.exists(calibri_path):
            pdfmetrics.registerFont(TTFont('Calibri', calibri_path))
        if os.path.exists(calibri_bold_path):
            pdfmetrics.registerFont(TTFont('CalibriB', calibri_bold_path))
    except Exception as e:
        # Fall back to standard fonts
        print(f"Error registering fonts: {e}. Using standard fonts.")

def _setup_styles(self):
    """Set up paragraph styles for the resume"""
    # Name style
    self.styles.add(ParagraphStyle(
        name='Name',
        fontName='CalibriB' if 'CalibriB' in pdfmetrics.getRegisteredFontNames() else 'Times New Roman',
        fontSize=14,
        leading=16,
        alignment=TA_LEFT,
        spaceAfter=6
    ))

    # Job title style
    self.styles.add(ParagraphStyle(
        name='JobTitle',
        fontName='CalibriB' if 'CalibriB' in pdfmetrics.getRegisteredFontNames() else 'Times New Roman',
        fontSize=11,
        leading=13,
        alignment=TA_LEFT,
```

```
spaceAfter=6
))

# Contact info style
self.styles.add(ParagraphStyle(
    name='ContactInfo',
    fontName='Calibri' if 'Calibri' in pdfmetrics.getRegisteredFontNames() else 'He
    fontSize=10,
    leading=12,
    alignment=TA_LEFT,
    spaceAfter=16
))

# Section heading style
self.styles.add(ParagraphStyle(
    name='SectionHeading',
    fontName='CalibriB' if 'Calibrib' in pdfmetrics.getRegisteredFontNames() else 'He
    fontSize=12,
    leading=14,
    alignment=TA_LEFT,
    spaceAfter=8,
    spaceBefore=12
))

# Body text style
self.styles.add(ParagraphStyle(
    name='BodyText',
    fontName='Calibri' if 'Calibri' in pdfmetrics.getRegisteredFontNames() else 'He
    fontSize=10,
    leading=12,
    alignment=TA_LEFT,
    spaceAfter=6
))

# Bullet style
self.styles.add(ParagraphStyle(
    name='Bullet',
    fontName='Calibri' if 'Calibri' in pdfmetrics.getRegisteredFontNames() else 'He
    fontSize=10,
    leading=14,
    leftIndent=18,
    firstLineIndent=-12,
    bulletIndent=0,
    spaceAfter=3
))
```

```
# Job position style
self.styles.add(ParagraphStyle(
    name='JobPosition',
    fontName='CalibriB' if 'CalibriB' in pdfmetrics.getRegisteredFontNames() else 'Times New Roman',
    fontSize=11,
    leading=13,
    alignment=TA_LEFT,
    spaceAfter=0
))

# Company style
self.styles.add(ParagraphStyle(
    name='Company',
    fontName='Calibri' if 'Calibri' in pdfmetrics.getRegisteredFontNames() else 'Times New Roman',
    fontSize=10,
    leading=12,
    alignment=TA_LEFT,
    spaceAfter=0
))

# Date range style
self.styles.add(ParagraphStyle(
    name='DateRange',
    fontName='CalibriB' if 'CalibriB' in pdfmetrics.getRegisteredFontNames() else 'Times New Roman',
    fontSize=10,
    leading=12,
    alignment=TA_LEFT,
    spaceAfter=6
))

def create_resume(
    self,
    resume_data: Dict[str, Any],
    output_path: Optional[str] = None,
    template_name: str = None
) -> str:
    """Create a résumé PDF using the provided data and template"""
    try:
        # Use default template if none specified
        template_name = template_name or self.default_template

        # Generate output path if not provided
        if not output_path:
            timestamp = datetime.now().strftime("%Y%m%d%H%M%S")
            company_name = resume_data.get("company_name", "company").replace(" ", "_")
            output_path = os.path.join(
                os.path.dirname(os.path.abspath(__file__)),
                f'{company_name}_{timestamp}.pdf'
            )
```
```

```
self.output_dir,
f"{resume_data['personal_info']['name'].replace(' ', '_')}{company_name}"
)

# Create the document
doc = SimpleDocTemplate(
    output_path,
    pagesize=LETTER,
    leftMargin=0.75*inch,
    rightMargin=0.75*inch,
    topMargin=0.75*inch,
    bottomMargin=0.75*inch
)

# Build the content
elements = []

# Add header section
self._add_header(elements, resume_data["personal_info"])

# Add professional summary
self._add_section(elements, "Professional Summary")
elements.append(Paragraph(resume_data["professional_summary"], self.styles["Body"]))

# Add skills section
if resume_data.get("skills"):
    self._add_section(elements, "Skills")
    self._add_skills(elements, resume_data["skills"])

# Add professional experience
if resume_data.get("experiences"):
    self._add_section(elements, "Professional Experience")
    self._add_experiences(elements, resume_data["experiences"])

# Add education
if resume_data.get("education"):
    self._add_section(elements, "Education")
    self._add_education(elements, resume_data["education"])

# Add certifications
if resume_data.get("certifications"):
    self._add_section(elements, "Certifications")
    self._add_certifications(elements, resume_data["certifications"])

# Build the PDF
doc.build(elements)
```

```
        return output_path

    except Exception as e:
        raise TemplateEngineError(f"Error creating resume: {str(e)}")

def _add_header(self, elements: List, personal_info: Dict[str, Any]):
    """Add the header section with name and contact info"""
    # Add name
    elements.append(Paragraph(personal_info["name"], self.styles["Name"]))

    # Add job title if provided
    if personal_info.get("job_title"):
        elements.append(Paragraph(personal_info["job_title"], self.styles["JobTitle"]))

    # Add contact info
    contact_parts = []
    if personal_info.get("email"):
        contact_parts.append(personal_info["email"])
    if personal_info.get("phone"):
        contact_parts.append(personal_info["phone"])
    if personal_info.get("linkedin"):
        contact_parts.append(personal_info["linkedin"])
    if personal_info.get("location"):
        contact_parts.append(personal_info["location"])

    contact_info = " | ".join(contact_parts)
    elements.append(Paragraph(contact_info, self.styles["ContactInfo"]))

def _add_section(self, elements: List, title: str):
    """Add a section heading"""
    elements.append(Paragraph(title, self.styles["SectionHeading"]))

def _add_skills(self, elements: List, skills: List[str]):
    """Add skills section"""
    # Group skills into chunks of 4-5 for better layout
    skill_chunks = []
    chunk_size = 5
    for i in range(0, len(skills), chunk_size):
        skill_chunks.append(skills[i:i + chunk_size])

    # Create a table for skills
    skill_data = []
    for chunk in skill_chunks:
        skill_row = []
        for skill in chunk:
```

```
bullet_text = f"• {skill}"
skill_row.append(Paragraph(bullet_text, self.styles["BodyText"]))

# Pad the row if needed
while len(skill_row) < chunk_size:
    skill_row.append("")

skill_data.append(skill_row)

# Create the table
if skill_data:
    skill_table = Table(skill_data, colWidths=[1.5*inch] * chunk_size)
    skill_table.setStyle(TableStyle([
        ('VALIGN', (0, 0), (-1, -1), 'TOP'),
        ('LEFTPADDING', (0, 0), (-1, -1), 0),
        ('RIGHTPADDING', (0, 0), (-1, -1), 0),
        ('BOTTOMPADDING', (0, 0), (-1, -1), 3),
        ('TOPPADDING', (0, 0), (-1, -1), 3),
    ]))

    elements.append(skill_table)
    elements.append(Spacer(1, 6))

def _add_experiences(self, elements: List, experiences: List[Dict[str, Any]]):
    """Add professional experiences"""
    for experience in experiences:
        # Add position
        elements.append(Paragraph(experience["position"], self.styles["JobPosition"]))

        # Add company and date on the same line
        company = experience.get("company", "")
        date_range = experience.get("date_range", "")

        elements.append(Paragraph(company, self.styles["Company"]))
        elements.append(Paragraph(date_range, self.styles["DateRange"]))

        # Add bullet points
        for bullet in experience.get("bullet_points", []):
            bullet_text = f"• {bullet}"
            elements.append(Paragraph(bullet_text, self.styles["Bullet"]))

        # Add space after each experience
        elements.append(Spacer(1, 6))

def _add_education(self, elements: List, education: List[Dict[str, Any]]):
    """Add education section"""


```

```

for edu in education:
    # Format degree info
    degree_text = f'{edu.get('degree', '')} in {edu.get('field', '')}'
    elements.append(Paragraph(degree_text, self.styles["JobPosition"]))

    # Format institution and date
    institution = edu.get("institution", "")
    location = edu.get("location", "")
    date_range = edu.get("date_range", "")

    institution_text = f'{institution}, {location}'
    if date_range:
        institution_text += f' | {date_range}'

    elements.append(Paragraph(institution_text, self.styles["Company"]))
    elements.append(Spacer(1, 12))

def _add_certifications(self, elements: List, certifications: List[Dict[str, Any]]):
    """Add certifications section"""
    for cert in certifications:
        cert_text = f'* {cert.get('name', '')}'
        elements.append(Paragraph(cert_text, self.styles["Bullet"]))

```

These five foundational components establish a solid, maintainable architecture for the job application automation system. Each component is designed to be modular, testable, and extensible.

To get started with development:

## 1. Set up the project structure and virtual environment:

bash

 Copy

```

mkdir job-application-automation
cd job-application-automation
python -m venv venv
source venv/bin/activate # or venv\Scripts\activate on Windows
pip install -r requirements.txt

```

## 2. Create the initial configuration files:

bash

 Copy

```

# Create .env file with necessary configuration
cp .env.example .env

```

```
# Edit .env with your specific settings
```

### 3. Set up MongoDB:

bash

 Copy

```
# If using Docker
docker-compose up -d mongodb
# Run setup script
python scripts/setup_db.py
```

### 4. Run initial tests:

bash

 Copy

```
pytest tests/
```

This foundational architecture provides a solid base for the complete system. Each component is designed with clear responsibilities and interfaces, making it easier to extend and maintain as requirements evolve.