Complete Candidate Evaluation System

Project Structure

```
candidate-evaluation-system/
├── README.md
├── .env.example
├── .gitignore
├── requirements.txt
├── docker-compose.yml
├── main.py
├── config/
│   ├── settings.py
│   ├── job_profiles/
│   │   └── senior_python_developer.json
│   └── sample_resumes/
│       └── john_doe_resume.json
├── src/
│   ├── __init__.py
│   ├── agents/
│   │   ├── __init__.py
│   │   ├── base_agent.py
│   │   ├── resume_evaluator.py
│   │   └── orchestrator.py
│   ├── mcp/
│   │   ├── __init__.py
│   │   ├── server.py
│   │   └── tools.py
│   ├── embeddings/
│   │   ├── __init__.py
│   │   ├── manager.py
│   │   └── faiss_index.py
│   ├── llm/
│   │   ├── __init__.py
│   │   └── gpt_client.py
│   ├── models/
│   │   ├── __init__.py
│   │   └── schemas.py
│   └── utils/
│       ├── __init__.py
│       ├── logger.py
│       └── fallback.py
└── logs/
    └── .gitkeep
```

1. README.md

```markdown
# Candidate Evaluation System

AI-powered candidate evaluation using embeddings, GPT-4o, and MCP protocol with fallback mechanisms.

## Features
- Resume evaluation using GPT-4o with fallback to dummy responses
- FAISS-based semantic search for candidate matching
- MCP server for standardized tool access
- Local JSON logging (no external dependencies)
- Configurable job profiles and sample resumes

## Setup
1. Install dependencies: `pip install -r requirements.txt`
2. Copy `.env.example` to `.env` and add your OpenAI API key
3. Run: `python main.py`

## Configuration
- Job profiles: `config/job_profiles/`
- Sample resumes: `config/sample_resumes/`
- Logs output: `logs/` directory
```

2. .env.example

```bash
# OpenAI API Configuration
OPENAI_API_KEY=your_openai_api_key_here
OPENAI_MODEL=gpt-4o
OPENAI_TEMPERATURE=0.3
OPENAI_MAX_TOKENS=2000

# Fallback Configuration
USE_FALLBACK=true
FALLBACK_AFTER_SECONDS=5

# MCP Configuration
MCP_SERVER_HOST=localhost
MCP_SERVER_PORT=8080

# Logging
LOG_LEVEL=INFO
LOG_FORMAT=json

# Embedding Configuration
EMBEDDING_MODEL=all-MiniLM-L6-v2
EMBEDDING_DIMENSION=384
FAISS_INDEX_PATH=./data/faiss_index

# System Configuration
MAX_WORKERS=4
BATCH_SIZE=32
```

---

3. .gitignore

```gitignore
# Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
env/
venv/
ENV/
.venv

# IDE
.vscode/
.idea/
*.swp
*.swo
.DS_Store

# Project specific
.env
logs/*.json
logs/*.log
data/
*.faiss
*.pkl

# Testing
.pytest_cache/
.coverage
htmlcov/
*.cover

# Docker
*.pid
```

---

4. requirements.txt

```txt
# Core dependencies
fastapi==0.104.1
uvicorn==0.24.0
pydantic==2.5.0
python-dotenv==1.0.0

# AI/ML dependencies
openai==1.12.0
langchain==0.1.0
langgraph==0.0.26
sentence-transformers==2.3.1
faiss-cpu==1.7.4
numpy==1.24.3
tiktoken==0.5.2

# MCP dependencies
httpx==0.25.2
websockets==12.0

# Utilities
redis==5.0.1
python-json-logger==2.0.7
tenacity==8.2.3
```

---

5. docker-compose.yml

```yaml
version: '3.8'

services:
  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    command: redis-server --appendonly yes

  app:
    build: .
    ports:
      - "8000:8000"
    environment:
      - REDIS_URL=redis://redis:6379
    env_file:
      - .env
    volumes:
      - ./logs:/app/logs
      - ./config:/app/config
      - ./data:/app/data
    depends_on:
      - redis

volumes:
  redis_data:
```

---

6. main.py

python

```python
#!/usr/bin/env python3
"""
Main entry point for the Candidate Evaluation System
"""

import asyncio
import json
from pathlib import Path
from datetime import datetime
from typing import Dict, Any

from dotenv import load_dotenv

from src.agents.orchestrator import EvaluationOrchestrator
from src.utils.logger import setup_logger
from config.settings import Settings

# Load environment variables
load_dotenv()

# Setup logger
logger = setup_logger(__name__)

class CandidateEvaluationSystem:
    def __init__(self):
        self.settings = Settings()
        self.orchestrator = EvaluationOrchestrator()
        self.logs_dir = Path("logs")
        self.logs_dir.mkdir(exist_ok=True)

    async def load_job_profile(self, profile_name: str = "senior_python_developer") -> Dict[str, Any]:
        """Load job profile from config"""
        profile_path = Path(f"config/job_profiles/{profile_name}.json")

        if not profile_path.exists():
            logger.warning(f"Job profile not found: {profile_path}")
            return self._get_default_job_profile()

        with open(profile_path, 'r') as f:
            return json.load(f)

    async def load_sample_resume(self, resume_name: str = "john_doe_resume") -> Dict[str, Any]:
        """Load sample resume from config"""
        resume_path = Path(f"config/sample_resumes/{resume_name}.json")
        if not resume_path.exists():
            logger.warning(f"Resume not found: {resume_path}")
            return self._get_default_resume()

        with open(resume_path, 'r') as f:
            return json.load(f)

    def _get_default_job_profile(self) -> Dict[str, Any]:
        """Default job profile for testing"""
        return {
            "title": "Senior Python Developer",
            "required_skills": ["Python", "FastAPI", "PostgreSQL", "Docker"],
            "preferred_skills": ["AWS", "Kubernetes", "React"],
            "min_experience": 5,
            "description": "Looking for a senior Python developer with strong backend skills"
        }

    def _get_default_resume(self) -> Dict[str, Any]:
        """Default resume for testing"""
        return {
            "name": "John Doe",
            "email": "john.doe@example.com",
            "skills": ["Python", "Django", "PostgreSQL", "Docker", "AWS"],
            "experience_years": 7,
            "summary": "Experienced Python developer with 7 years in web development",
            "experience": [
                {
                    "company": "Tech Corp",
                    "role": "Senior Developer",
                    "duration": "3 years",
                    "description": "Led backend development using Python and FastAPI"
                }
            ]
        }

    async def evaluate_candidate(self, resume: Dict[str, Any], job_profile: Dict[str, Any]) -> Dict[str, Any]:
        """Main evaluation pipeline"""
        logger.info("Starting candidate evaluation")

        try:
            # Run evaluation through orchestrator
            evaluation_result = await self.orchestrator.evaluate(
                resume=resume,
                job_profile=job_profile
            )
            # Add metadata
            evaluation_result["timestamp"] = datetime.utcnow().isoformat()
            evaluation_result["status"] = "completed"

            # Log results to file
            await self._log_results(evaluation_result)

            return evaluation_result

        except Exception as e:
            logger.error(f"Evaluation failed: {str(e)}", exc_info=True)

            # Fallback response
            fallback_result = self._get_fallback_evaluation(resume, job_profile)
            await self._log_results(fallback_result)

            return fallback_result

    def _get_fallback_evaluation(self, resume: Dict[str, Any], job_profile: Dict[str, Any]) -> Dict[str, Any]:
        """Generate fallback evaluation when API fails"""
        logger.info("Using fallback evaluation")

        # Simple rule-based matching
        required_skills = set(job_profile.get("required_skills", []))
        candidate_skills = set(resume.get("skills", []))

        matched_skills = required_skills.intersection(candidate_skills)
        match_percentage = len(matched_skills) / len(required_skills) * 100 if required_skills else 0

        return {
            "timestamp": datetime.utcnow().isoformat(),
            "status": "fallback",
            "candidate_name": resume.get("name", "Unknown"),
            "match_score": round(match_percentage, 2),
            "matched_skills": list(matched_skills),
            "missing_skills": list(required_skills - candidate_skills),
            "recommendation": "CONSIDER" if match_percentage >= 60 else "REJECT",
            "evaluation_method": "fallback_rules",
            "details": {
                "experience_match": resume.get("experience_years", 0) >= job_profile.get("min_experience", 0),
                "skills_match_percentage": match_percentage,
                "used_fallback": True,
                "fallback_reason": "API unavailable or timeout"
            }
        }
```

```python
    async def _log_results(self, results: Dict[str, Any]):
        """Log evaluation results to JSON file"""
        timestamp = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
        log_file = self.logs_dir / f"evaluation_{timestamp}.json"

        with open(log_file, 'w') as f:
            json.dump(results, f, indent=2, default=str)

        logger.info(f"Results logged to: {log_file}")

    async def run(self):
        """Main execution"""
        logger.info("=== Candidate Evaluation System Started ===")

        # Load job profile and resume
        job_profile = await self.load_job_profile()
        resume = await self.load_sample_resume()

        logger.info(f"Loaded job profile: {job_profile.get('title')}")
        logger.info(f"Loaded resume: {resume.get('name')}")

        # Run evaluation
        result = await self.evaluate_candidate(resume, job_profile)

        # Print summary
        print("\n" + "="*50)
        print("EVALUATION SUMMARY")
        print("="*50)
        print(f"Candidate: {result.get('candidate_name')}")
        print(f"Match Score: {result.get('match_score')}%")
        print(f"Recommendation: {result.get('recommendation')}")
        print(f"Evaluation Method: {result.get('evaluation_method', 'AI')}")
        print(f"Log file: logs/evaluation_*.json")
        print("="*50)

        return result

async def main():
    """Entry point"""
    system = CandidateEvaluationSystem()
    await system.run()

if __name__ == "__main__":
    asyncio.run(main())
```

7. config/settings.py

```python
from pydantic import BaseSettings, Field
from typing import Optional
import os

class Settings(BaseSettings):
    """Application settings"""

    # OpenAI Configuration
    openai_api_key: str = Field(..., env="OPENAI_API_KEY")
    openai_model: str = Field("gpt-4o", env="OPENAI_MODEL")
    openai_temperature: float = Field(0.3, env="OPENAI_TEMPERATURE")
    openai_max_tokens: int = Field(2000, env="OPENAI_MAX_TOKENS")

    # Fallback Configuration
    use_fallback: bool = Field(True, env="USE_FALLBACK")
    fallback_timeout: int = Field(5, env="FALLBACK_AFTER_SECONDS")

    # MCP Configuration
    mcp_server_host: str = Field("localhost", env="MCP_SERVER_HOST")
    mcp_server_port: int = Field(8080, env="MCP_SERVER_PORT")

    # Embedding Configuration
    embedding_model: str = Field("all-MiniLM-L6-v2", env="EMBEDDING_MODEL")
    embedding_dimension: int = Field(384, env="EMBEDDING_DIMENSION")
    faiss_index_path: str = Field("./data/faiss_index", env="FAISS_INDEX_PATH")

    # System Configuration
    max_workers: int = Field(4, env="MAX_WORKERS")
    batch_size: int = Field(32, env="BATCH_SIZE")
    log_level: str = Field("INFO", env="LOG_LEVEL")

    class Config:
        env_file = ".env"
        case_sensitive = False

# Global settings instance
settings = Settings()
```

---

8. config/job_profiles/senior_python_developer.json

json

```json
{
  "id": "job_001",
  "title": "Senior Python Developer",
  "department": "Engineering",
  "location": "Remote",
  "type": "Full-time",
  "required_skills": [
    "Python",
    "FastAPI",
    "PostgreSQL",
    "Docker",
    "REST APIs",
    "Git"
  ],
  "preferred_skills": [
    "AWS",
    "Kubernetes",
    "Redis",
    "Elasticsearch",
    "React",
    "TypeScript"
  ],
  "min_experience": 5,
  "max_experience": 10,
  "education": {
    "required": "Bachelor's in Computer Science or related field",
    "preferred": "Master's degree"
  },
  "responsibilities": [
    "Design and develop scalable backend services",
    "Implement RESTful APIs using FastAPI",
    "Optimize database queries and performance",
    "Collaborate with frontend team",
    "Code review and mentoring junior developers"
  ],
  "nice_to_have": [
    "Experience with AI/ML frameworks",
    "Open source contributions",
    "System design experience"
  ],
  "evaluation_criteria": {
    "technical_weight": 0.4,
    "experience_weight": 0.3,
    "skills_weight": 0.3
  }
}
```

9. config/sample_resumes/john_doe_resume.json

json

```json
{
  "id": "resume_001",
  "name": "John Doe",
  "email": "john.doe@example.com",
  "phone": "+1-555-0123",
  "location": "San Francisco, CA",
  "linkedin": "linkedin.com/in/johndoe",
  "github": "github.com/johndoe",

  "summary": "Experienced Senior Python Developer with 7+ years of expertise in building scalable web applications and RE

  "skills": [
    "Python",
    "FastAPI",
    "Django",
    "PostgreSQL",
    "MongoDB",
    "Docker",
    "Kubernetes",
    "AWS",
    "Redis",
    "Celery",
    "REST APIs",
    "GraphQL",
    "Git",
    "CI/CD",
    "Microservices"
  ],

  "experience_years": 7,

  "experience": [
    {
      "company": "Tech Solutions Inc.",
      "position": "Senior Python Developer",
      "location": "San Francisco, CA",
      "start_date": "2021-01",
      "end_date": "present",
      "duration": "3 years",
      "description": "Lead backend developer for a high-traffic e-commerce platform",
      "achievements": [
        "Designed and implemented microservices architecture using FastAPI, reducing response time by 40%",
        "Optimized PostgreSQL queries resulting in 60% performance improvement",
        "Led team of 4 developers, conducting code reviews and mentoring",
        "Implemented CI/CD pipelines using GitHub Actions and AWS"
        "Implemented CI/CD pipeline using GitHub Actions and AWS"
      ],
      "technologies": ["Python", "FastAPI", "PostgreSQL", "Redis", "Docker", "AWS", "Kubernetes"]
    },
    {
      "company": "StartupXYZ",
      "position": "Python Developer",
      "location": "Remote",
      "start_date": "2018-06",
      "end_date": "2020-12",
      "duration": "2.5 years",
      "description": "Full-stack developer for a SaaS analytics platform",
      "achievements": [
        "Built RESTful APIs serving 100K+ daily requests",
        "Implemented real-time data processing pipeline using Celery and Redis",
        "Reduced database costs by 30% through query optimization"
      ],
      "technologies": ["Python", "Django", "PostgreSQL", "Celery", "Redis", "Docker"]
    },
    {
      "company": "Digital Agency Co.",
      "position": "Junior Python Developer",
      "location": "New York, NY",
      "start_date": "2017-01",
      "end_date": "2018-05",
      "duration": "1.5 years",
      "description": "Backend developer for various client projects",
      "achievements": [
        "Developed REST APIs for 10+ client projects",
        "Automated deployment process using Docker",
        "Participated in agile development process"
      ],
      "technologies": ["Python", "Flask", "MySQL", "Docker", "Git"]
    }
  ],

  "education": [
    {
      "degree": "Bachelor of Science in Computer Science",
      "university": "University of California, Berkeley",
      "graduation_year": 2016,
      "gpa": 3.8
    }
  ],

  "certifications": [
    {
      "name": "AWS Certified Developer - Associate",
      "issuer": "Amazon Web Services",
      "date": "2022-03"
    },
    {
      "name": "Docker Certified Associate",
      "issuer": "Docker",
      "date": "2021-08"
    }
  ],

  "projects": [
    {
      "name": "Open Source API Framework",
      "description": "Contributed to popular Python API framework",
      "url": "github.com/framework/repo",
      "contributions": "Added async support and improved documentation"
    }
  ]
}
```

---

10. src/agents/base_agent.py

python

```python
from abc import ABC, abstractmethod
from typing import Dict, Any, Optional
import asyncio
from tenacity import retry, stop_after_attempt, wait_exponential

from src.utils.logger import setup_logger

logger = setup_logger(__name__)

class BaseAgent(ABC):
    """Base class for all agents"""

    def __init__(self, name: str):
        self.name = name
        self.logger = setup_logger(f"agent.{name}")

    @abstractmethod
    async def process(self, input_data: Dict[str, Any]) -> Dict[str, Any]:
        """Process input and return results"""
        pass

    @retry(
        stop=stop_after_attempt(3),
        wait=wait_exponential(multiplier=1, min=4, max=10)
    )
    async def execute_with_retry(self, func, *args, **kwargs):
        """Execute function with retry logic"""
        try:
            return await func(*args, **kwargs)
        except Exception as e:
            self.logger.error(f"Error in {self.name}: {str(e)}")
            raise

    async def validate_input(self, input_data: Dict[str, Any]) -> bool:
        """Validate input data"""
        if not input_data:
            self.logger.error("Empty input data")
            return False
        return True

    def log_processing(self, input_data: Dict[str, Any], result: Dict[str, Any]):
        """Log processing details"""
        self.logger.info(f"Processing completed by {self.name}")
        self.logger.debug(f"Input: {input_data}")
        self.logger.debug(f"Result: {result}")
    self.logger.debug(f"Result: {result}")
```

11. src/agents/resume_evaluator.py

python

```python
from typing import Dict, Any, List, Optional
import asyncio
from datetime import datetime

from src.agents.base_agent import BaseAgent
from src.llm.gpt_client import GPTClient
from src.embeddings.manager import EmbeddingManager
from src.utils.fallback import FallbackEvaluator

class ResumeEvaluatorAgent(BaseAgent):
    """Agent for evaluating resumes against job profiles"""

    def __init__(self):
        super().__init__("resume_evaluator")
        self.gpt_client = GPTClient()
        self.embedding_manager = EmbeddingManager()
        self.fallback_evaluator = FallbackEvaluator()

    async def process(self, input_data: Dict[str, Any]) -> Dict[str, Any]:
        """Evaluate resume against job profile"""

        resume = input_data.get("resume")
        job_profile = input_data.get("job_profile")

        if not await self.validate_input(input_data):
            return self.fallback_evaluator.evaluate(resume, job_profile)

        try:
            # Try GPT-4o evaluation
            gpt_evaluation = await self._evaluate_with_gpt(resume, job_profile)

            # Generate embeddings for similarity search
            embeddings = await self._generate_embeddings(resume, job_profile)

            # Combine results
            result = {
                "candidate_name": resume.get("name"),
                "evaluation_method": "gpt-4o",
                "gpt_analysis": gpt_evaluation,
                "embeddings": embeddings,
                "timestamp": datetime.utcnow().isoformat()
            }

            self.log_processing(input_data, result)
            return result

        except Exception as e:
            self.logger.error(f"GPT evaluation failed: {str(e)}")
            return self.fallback_evaluator.evaluate(resume, job_profile)

    async def _evaluate_with_gpt(self, resume: Dict[str, Any], job_profile: Dict[str, Any]) -> Dict[str, Any]:
        """Evaluate using GPT-4o"""

        prompt = self._build_evaluation_prompt(resume, job_profile)

        try:
            response = await asyncio.wait_for(
                self.gpt_client.evaluate(prompt),
                timeout=5.0  # 5 second timeout
            )
            return response
        except asyncio.TimeoutError:
            self.logger.warning("GPT-4o timeout, using fallback")
            raise

    def _build_evaluation_prompt(self, resume: Dict[str, Any], job_profile: Dict[str, Any]) -> str:
        """Build evaluation prompt for GPT-4o"""

        return f"""
        Evaluate the following resume against the job profile:

        JOB PROFILE:
        Title: {job_profile.get('title')}
        Required Skills: {', '.join(job_profile.get('required_skills', []))}
        Preferred Skills: {', '.join(job_profile.get('preferred_skills', []))}
        Minimum Experience: {job_profile.get('min_experience')} years

        RESUME:
        Name: {resume.get('name')}
        Skills: {', '.join(resume.get('skills', []))}
        Experience: {resume.get('experience_years')} years
        Summary: {resume.get('summary')}

        Please provide:
        1. Match percentage (0-100)
        2. Matched skills
        3. Missing skills
        4. Overall recommendation (STRONG_YES/YES/MAYBE/NO)
        5. Key strengths
        6. Areas of concern
        Format the response as JSON.
        """

    async def _generate_embeddings(self, resume: Dict[str, Any], job_profile: Dict[str, Any]) -> Dict[str, Any]:
        """Generate embeddings for semantic matching"""

        # Combine resume text
        resume_text = f"{resume.get('summary', '')} {' '.join(resume.get('skills', []))}"

        # Generate embedding
        resume_embedding = await self.embedding_manager.generate_embedding(resume_text)

        return {
            "resume_embedding_generated": True,
            "embedding_dimension": len(resume_embedding) if resume_embedding else 0
        }
```

12. src/agents/orchestrator.py

python

```python
from typing import Dict, Any, List
import asyncio
from datetime import datetime

from src.agents.base_agent import BaseAgent
from src.agents.resume_evaluator import ResumeEvaluatorAgent
from src.mcp.server import MCPServer
from src.utils.logger import setup_logger

logger = setup_logger(__name__)

class EvaluationOrchestrator:
    """Orchestrates the evaluation pipeline"""

    def __init__(self):
        self.resume_evaluator = ResumeEvaluatorAgent()
        self.mcp_server = MCPServer()
        self.logger = logger

    async def evaluate(self, resume: Dict[str, Any], job_profile: Dict[str, Any]) -> Dict[str, Any]:
        """Main evaluation pipeline"""

        self.logger.info("Starting evaluation pipeline")

        # Start MCP server
        await self.mcp_server.start()

        try:
            # Step 1: Resume evaluation
            evaluation_result = await self.resume_evaluator.process({
                "resume": resume,
                "job_profile": job_profile
            })

            # Step 2: Use MCP tools for additional analysis
            mcp_analysis = await self._run_mcp_analysis(evaluation_result)

            # Step 3: Compile final results
            final_result = self._compile_results(evaluation_result, mcp_analysis)

            return final_result

        except Exception as e:
            self.logger.error(f"Orchestration failed: {str(e)}")
            raise
        finally:
            await self.mcp_server.stop()

    async def _run_mcp_analysis(self, evaluation_result: Dict[str, Any]) -> Dict[str, Any]:
        """Run additional analysis using MCP tools"""

        try:
            # Call MCP tools
            skills_analysis = await self.mcp_server.analyze_skills(
                evaluation_result.get("gpt_analysis", {})
            )

            return {
                "mcp_analysis_completed": True,
                "skills_analysis": skills_analysis
            }
        except Exception as e:
            self.logger.warning(f"MCP analysis failed: {str(e)}")
            return {"mcp_analysis_completed": False}

    def _compile_results(self, evaluation: Dict[str, Any], mcp_analysis: Dict[str, Any]) -> Dict[str, Any]:
        """Compile final evaluation results"""

        # Extract key information
        gpt_analysis = evaluation.get("gpt_analysis", {})

        # Determine final recommendation
        recommendation = self._determine_recommendation(gpt_analysis)

        return {
            "candidate_name": evaluation.get("candidate_name"),
            "evaluation_method": evaluation.get("evaluation_method"),
            "match_score": gpt_analysis.get("match_percentage", 0),
            "matched_skills": gpt_analysis.get("matched_skills", []),
            "missing_skills": gpt_analysis.get("missing_skills", []),
            "recommendation": recommendation,
            "strengths": gpt_analysis.get("key_strengths", []),
            "concerns": gpt_analysis.get("areas_of_concern", []),
            "mcp_analysis": mcp_analysis,
            "timestamp": datetime.utcnow().isoformat()
        }

    def _determine_recommendation(self, analysis: Dict[str, Any]) -> str:
        """Determine final recommendation"""

        if not analysis:
```

```
        return "UNABLE_TO_EVALUATE"

    recommendation = analysis.get("recommendation", "")
    match_score = analysis.get("match_percentage", 0)

    if recommendation:
        return recommendation

    # Fallback logic based on match score
    if match_score >= 80:
        return "STRONG_YES"
    elif match_score >= 60:
        return "YES"
    elif match_score >= 40:
        return "MAYBE"
    else:
        return "NO"
```

---

13. src/mcp/server.py

python

```python
import asyncio
from typing import Dict, Any, Optional, List
import json

from src.mcp.tools import MCPTools
from src.utils.logger import setup_logger

logger = setup_logger(__name__)

class MCPServer:
    """MCP Server for standardized tool access"""

    def __init__(self, host: str = "localhost", port: int = 8080):
        self.host = host
        self.port = port
        self.tools = MCPTools()
        self.is_running = False
        self.logger = logger

    async def start(self):
        """Start MCP server"""
        self.is_running = True
        self.logger.info(f"MCP Server started on {self.host}:{self.port}")

    async def stop(self):
        """Stop MCP server"""
        self.is_running = False
        self.logger.info("MCP Server stopped")

    async def analyze_skills(self, evaluation_data: Dict[str, Any]) -> Dict[str, Any]:
        """Analyze skills using MCP tools"""

        if not self.is_running:
            self.logger.error("MCP Server is not running")
            return {}

        try:
            # Use MCP tools for skills analysis
            result = await self.tools.analyze_skills_match(evaluation_data)
            return result

        except Exception as e:
            self.logger.error(f"MCP skills analysis failed: {str(e)}")
            return {}
    async def search_similar_candidates(self,
```