Complete Candidate Evaluation System Files (Part 2)

14. src/mcp/tools.py

python

```python
from typing import Dict, Any, List
import json
import numpy as np

from src.utils.logger import setup_logger

logger = setup_logger(__name__)

class MCPTools:
    """MCP standardized tools"""

    def __init__(self):
        self.logger = logger

    async def analyze_skills_match(self, evaluation_data: Dict[str, Any]) -> Dict[str, Any]:
        """Analyze skills matching details"""

        try:
            matched_skills = evaluation_data.get("matched_skills", [])
            missing_skills = evaluation_data.get("missing_skills", [])

            # Calculate detailed metrics
            total_skills = len(matched_skills) + len(missing_skills)
            match_ratio = len(matched_skills) / total_skills if total_skills > 0 else 0

            # Categorize skills
            skill_categories = self._categorize_skills(matched_skills)

            return {
                "total_required_skills": total_skills,
                "matched_count": len(matched_skills),
                "missing_count": len(missing_skills),
                "match_ratio": round(match_ratio, 2),
                "skill_categories": skill_categories,
                "critical_missing": self._identify_critical_skills(missing_skills)
            }

        except Exception as e:
            self.logger.error(f"Skills analysis failed: {str(e)}")
            return {}

    def _categorize_skills(self, skills: List[str]) -> Dict[str, List[str]]:
        """Categorize skills by type"""
```

```python
    categories = {
        "languages": [],
        "frameworks": [],
        "databases": [],
        "tools": [],
        "cloud": [],
        "other": []
    }

    # Simple categorization logic
    for skill in skills:
        skill_lower = skill.lower()

        if skill_lower in ["python", "java", "javascript", "typescript", "go", "rust"]:
            categories["languages"].append(skill)
        elif skill_lower in ["fastapi", "django", "react", "vue", "angular", "spring"]:
            categories["frameworks"].append(skill)
        elif skill_lower in ["postgresql", "mysql", "mongodb", "redis", "elasticsearch"]:
            categories["databases"].append(skill)
        elif skill_lower in ["docker", "kubernetes", "git", "jenkins", "terraform"]:
            categories["tools"].append(skill)
        elif skill_lower in ["aws", "gcp", "azure"]:
            categories["cloud"].append(skill)
        else:
            categories["other"].append(skill)

    return categories

def _identify_critical_skills(self, missing_skills: List[str]) -> List[str]:
    """Identify critical missing skills"""

    critical = ["Python", "FastAPI", "PostgreSQL"]
    return [skill for skill in missing_skills if skill in critical]

async def search_candidates(self, query_embedding: List[float]) -> List[Dict[str, Any]]:
    """Search for similar candidates (placeholder)"""

    # This would connect to FAISS in production
    return []
```

---

15. src/llm/gpt_client.py

python

```python
import openai
import json
import asyncio
from typing import Dict, Any, Optional
from tenacity import retry, stop_after_attempt, wait_exponential

from config.settings import settings
from src.utils.logger import setup_logger

logger = setup_logger(__name__)

class GPTClient:
    """Client for GPT-4o API calls"""

    def __init__(self):
        self.client = openai.AsyncOpenAI(api_key=settings.openai_api_key)
        self.model = settings.openai_model
        self.temperature = settings.openai_temperature
        self.max_tokens = settings.openai_max_tokens
        self.logger = logger

    @retry(
        stop=stop_after_attempt(3),
        wait=wait_exponential(multiplier=1, min=2, max=10)
    )
    async def evaluate(self, prompt: str) -> Dict[str, Any]:
        """Call GPT-4o for evaluation"""

        try:
            self.logger.info("Calling GPT-4o for evaluation")

            response = await self.client.chat.completions.create(
                model=self.model,
                messages=[
                    {
                        "role": "system",
                        "content": "You are an expert technical recruiter. Always respond in valid JSON format."
                    },
                    {
                        "role": "user",
                        "content": prompt
                    }
                ],
                temperature=self.temperature,
```

```
            max_tokens=self.max_tokens,
            response_format={"type": "json_object"}
        )

        # Parse response
        content = response.choices[0].message.content
        result = json.loads(content)

        self.logger.info("GPT-4o evaluation successful")
        return result

    except json.JSONDecodeError as e:
        self.logger.error(f"Failed to parse GPT response: {str(e)}")
        raise
    except Exception as e:
        self.logger.error(f"GPT-4o API call failed: {str(e)}")
        raise

async def generate_questions(self, job_profile: Dict[str, Any]) -> List[str]:
    """Generate interview questions based on job profile"""

    prompt = f"""
    Generate 5 technical interview questions for the following position:
    Title: {job_profile.get('title')}
    Required Skills: {', '.join(job_profile.get('required_skills', []))}

    Format as JSON with a 'questions' array.
    """

    try:
        response = await self.evaluate(prompt)
        return response.get("questions", [])
    except:
        # Fallback questions
        return [
            "Can you explain your experience with Python?",
            "How do you approach API design?",
            "Describe a challenging problem you solved.",
            "What's your experience with databases?",
            "How do you ensure code quality?"
        ]
```

---

16. src/embeddings/manager.py

python

```python
import numpy as np
from typing import List, Optional, Dict, Any
from sentence_transformers import SentenceTransformer
import pickle
from pathlib import Path

from config.settings import settings
from src.utils.logger import setup_logger

logger = setup_logger(__name__)

class EmbeddingManager:
    """Manages embedding generation and caching"""

    def __init__(self):
        self.model_name = settings.embedding_model
        self.dimension = settings.embedding_dimension
        self.model = None
        self.cache_dir = Path("data/embedding_cache")
        self.cache_dir.mkdir(parents=True, exist_ok=True)
        self.logger = logger

    def _load_model(self):
        """Lazy load the embedding model"""
        if self.model is None:
            self.logger.info(f"Loading embedding model: {self.model_name}")
            self.model = SentenceTransformer(self.model_name)
        return self.model

    async def generate_embedding(self, text: str) -> List[float]:
        """Generate embedding for text"""

        try:
            # Check cache first
            cached = self._get_cached_embedding(text)
            if cached is not None:
                return cached

            # Generate new embedding
            model = self._load_model()
            embedding = model.encode(text, convert_to_numpy=True)
            embedding_list = embedding.tolist()

            # Cache the result
            self._cache_embedding(text, embedding_list)
```

```
        self._cache_embedding(text, embedding_list)

        return embedding_list

    except Exception as e:
        self.logger.error(f"Embedding generation failed: {str(e)}")
        # Return zero vector as fallback
        return [0.0] * self.dimension

async def generate_batch_embeddings(self, texts: List[str]) -> List[List[float]]:
    """Generate embeddings for multiple texts"""

    try:
        model = self._load_model()
        embeddings = model.encode(texts, convert_to_numpy=True, show_progress_bar=False)
        return embeddings.tolist()
    except Exception as e:
        self.logger.error(f"Batch embedding generation failed: {str(e)}")
        return [[0.0] * self.dimension for _ in texts]

def _get_cache_key(self, text: str) -> str:
    """Generate cache key for text"""
    import hashlib
    return hashlib.md5(text.encode()).hexdigest()

def _get_cached_embedding(self, text: str) -> Optional[List[float]]:
    """Retrieve cached embedding if exists"""
    cache_key = self._get_cache_key(text)
    cache_file = self.cache_dir / f"{cache_key}.pkl"

    if cache_file.exists():
        try:
            with open(cache_file, 'rb') as f:
                embedding = pickle.load(f)
            self.logger.debug(f"Using cached embedding for: {text[:50]}...")
            return embedding
        except:
            pass
    return None

def _cache_embedding(self, text: str, embedding: List[float]):
    """Cache embedding for future use"""
    cache_key = self._get_cache_key(text)
    cache_file = self.cache_dir / f"{cache_key}.pkl"

    try:
        with open(cache_file, 'wb') as f:
```

```
            pickle.dump(embedding, f)
        except Exception as e:
            self.logger.warning(f"Failed to cache embedding: {str(e)}")
```

---

17. src/embeddings/faiss_index.py

python

```python
import faiss
import numpy as np
import pickle
from pathlib import Path
from typing import List, Dict, Any, Tuple

from config.settings import settings
from src.utils.logger import setup_logger

logger = setup_logger(__name__)

class FAISSIndex:
    """FAISS index for similarity search"""

    def __init__(self):
        self.dimension = settings.embedding_dimension
        self.index_path = Path(settings.faiss_index_path)
        self.index_path.parent.mkdir(parents=True, exist_ok=True)

        self.index = None
        self.id_map = []  # Maps FAISS IDs to actual candidate IDs
        self.metadata = {}  # Stores candidate metadata
        self.logger = logger

        self._initialize_index()

    def _initialize_index(self):
        """Initialize or load FAISS index"""

        index_file = self.index_path.with_suffix('.index')
        metadata_file = self.index_path.with_suffix('.meta')

        if index_file.exists() and metadata_file.exists():
            # Load existing index
            try:
                self.index = faiss.read_index(str(index_file))
                with open(metadata_file, 'rb') as f:
                    data = pickle.load(f)
                    self.id_map = data['id_map']
                    self.metadata = data['metadata']
                self.logger.info(f"Loaded FAISS index with {self.index.ntotal} vectors")
            except Exception as e:
                self.logger.error(f"Failed to load index: {str(e)}")
                self._create_new_index()
        else:
```

```
    else:
        self._create_new_index()

def _create_new_index(self):
    """Create new FAISS index"""

    # Use Inner Product for normalized vectors (cosine similarity)
    self.index = faiss.IndexFlatIP(self.dimension)
    self.id_map = []
    self.metadata = {}
    self.logger.info("Created new FAISS index")

def add_candidate(self, candidate_id: str, embedding: List[float], metadata: Dict[str, Any]):
    """Add candidate to index"""

    try:
        # Convert to numpy array and normalize
        embedding_np = np.array([embedding], dtype='float32')
        faiss.normalize_L2(embedding_np)

        # Add to index
        self.index.add(embedding_np)

        # Store mapping and metadata
        faiss_id = self.index.ntotal - 1
        self.id_map.append(candidate_id)
        self.metadata[candidate_id] = metadata

        self.logger.debug(f"Added candidate {candidate_id} to index")

    except Exception as e:
        self.logger.error(f"Failed to add candidate: {str(e)}")

def search(self, query_embedding: List[float], k: int = 10) -> List[Dict[str, Any]]:
    """Search for similar candidates"""

    if self.index.ntotal == 0:
        self.logger.warning("Index is empty")
        return []

    try:
        # Normalize query
        query_np = np.array([query_embedding], dtype='float32')
        faiss.normalize_L2(query_np)

        # Search
        scores, indices = self.index.search(query_np, min(k, self.index.ntotal))
```

```python
        # Build results
        results = []
        for score, idx in zip(scores[0], indices[0]):
            if idx >= 0 and idx < len(self.id_map):
                candidate_id = self.id_map[idx]
                results.append({
                    'candidate_id': candidate_id,
                    'similarity_score': float(score),
                    'metadata': self.metadata.get(candidate_id, {})
                })

        return results

    except Exception as e:
        self.logger.error(f"Search failed: {str(e)}")
        return []

def save(self):
    """Save index to disk"""

    try:
        index_file = self.index_path.with_suffix('.index')
        metadata_file = self.index_path.with_suffix('.meta')

        # Save index
        faiss.write_index(self.index, str(index_file))

        # Save metadata
        with open(metadata_file, 'wb') as f:
            pickle.dump({
                'id_map': self.id_map,
                'metadata': self.metadata
            }, f)

        self.logger.info(f"Saved index with {self.index.ntotal} vectors")

    except Exception as e:
        self.logger.error(f"Failed to save index: {str(e)}")
```

---

18. src/utils/logger.py

python

```python
import logging
import json
from datetime import datetime
from pathlib import Path
from typing import Any, Dict
from pythonjsonlogger import jsonlogger

from config.settings import settings

# Create logs directory
log_dir = Path("logs")
log_dir.mkdir(exist_ok=True)

def setup_logger(name: str) -> logging.Logger:
    """Setup JSON logger"""

    logger = logging.getLogger(name)
    logger.setLevel(getattr(logging, settings.log_level.upper()))

    # Remove existing handlers
    logger.handlers.clear()

    # Console handler
    console_handler = logging.StreamHandler()
    console_formatter = jsonlogger.JsonFormatter(
        '%(timestamp)s %(level)s %(name)s %(message)s',
        rename_fields={'levelname': 'level', 'name': 'logger'}
    )
    console_handler.setFormatter(console_formatter)
    logger.addHandler(console_handler)

    # File handler
    file_handler = logging.FileHandler(
        log_dir / f"evaluation_{datetime.now().strftime('%Y%m%d')}.log"
    )
    file_formatter = jsonlogger.JsonFormatter(
        '%(timestamp)s %(level)s %(name)s %(message)s %(pathname)s %(lineno)d',
        rename_fields={'levelname': 'level', 'name': 'logger'}
    )
    file_handler.setFormatter(file_formatter)
    logger.addHandler(file_handler)

    # Add timestamp
    class TimestampFilter(logging.Filter):
        def filter(self, record):
```

```
        def filter(self, record):
            record.timestamp = datetime.utcnow().isoformat()
            return True

    logger.addFilter(TimestampFilter())

    return logger

class EvaluationLogger:
    """Logger specifically for evaluation results"""

    def __init__(self):
        self.log_dir = Path("logs")
        self.log_dir.mkdir(exist_ok=True)

    def log_evaluation(self, evaluation_data: Dict[str, Any]):
        """Log evaluation to JSON file"""

        timestamp = datetime.utcnow().strftime("%Y%m%d_%H%M%S_%f")[:-3]
        filename = self.log_dir / f"eval_{timestamp}.json"

        # Add metadata
        log_entry = {
            "timestamp": datetime.utcnow().isoformat(),
            "type": "evaluation",
            "data": evaluation_data
        }

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

        return filename
```

---

19. src/utils/fallback.py

python

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

from src.utils.logger import setup_logger

logger = setup_logger(__name__)

class FallbackEvaluator:
    """Fallback evaluation when API is unavailable"""

    def __init__(self):
        self.logger = logger

    def evaluate(self, resume: Dict[str, Any], job_profile: Dict[str, Any]) -> Dict[str, Any]:
        """Rule-based fallback evaluation"""

        self.logger.info("Using fallback evaluation")

        # Extract data
        candidate_skills = set(resume.get("skills", []))
        required_skills = set(job_profile.get("required_skills", []))
        preferred_skills = set(job_profile.get("preferred_skills", []))

        candidate_experience = resume.get("experience_years", 0)
        min_experience = job_profile.get("min_experience", 0)

        # Calculate matches
        matched_required = candidate_skills.intersection(required_skills)
        matched_preferred = candidate_skills.intersection(preferred_skills)
        missing_required = required_skills - candidate_skills

        # Calculate scores
        required_score = len(matched_required) / len(required_skills) * 100 if required_skills else 0
        preferred_score = len(matched_preferred) / len(preferred_skills) * 50 if preferred_skills else 0
        experience_score = 100 if candidate_experience >= min_experience else (candidate_experience / min_experience * 100)

        # Overall score (weighted)
        overall_score = (required_score * 0.5 + preferred_score * 0.2 + experience_score * 0.3)

        # Determine recommendation
        if overall_score >= 80 and len(missing_required) <= 1:
            recommendation = "STRONG_YES"
        elif overall_score >= 60 and len(missing_required) <= 2:
            recommendation = "YES"
        elif overall_score >= 40
```

```
    elif overall_score >= 40:
        recommendation = "MAYBE"
    else:
        recommendation = "NO"

    return {
        "evaluation_method": "fallback_rules",
        "gpt_analysis": {
            "match_percentage": round(overall_score, 2),
            "matched_skills": list(matched_required.union(matched_preferred)),
            "missing_skills": list(missing_required),
            "recommendation": recommendation,
            "key_strengths": self._identify_strengths(resume, matched_required),
            "areas_of_concern": self._identify_concerns(missing_required, candidate_experience, min_experience)
        },
        "fallback_details": {
            "reason": "API unavailable or timeout",
            "required_skills_match": round(required_score, 2),
            "preferred_skills_match": round(preferred_score, 2),
            "experience_match": round(experience_score, 2)
        },
        "timestamp": datetime.utcnow().isoformat()
    }

def _identify_strengths(self, resume: Dict[str, Any], matched_skills: set) -> List[str]:
    """Identify candidate strengths"""

    strengths = []

    if resume.get("experience_years", 0) >= 5:
        strengths.append(f"{resume.get('experience_years')} years of experience")

    if len(matched_skills) >= 3:
        strengths.append(f"Strong technical skills match ({len(matched_skills)} skills)")

    if resume.get("education"):
        strengths.append("Relevant educational background")

    return strengths

def _identify_concerns(self, missing_skills: set, candidate_exp: int, required_exp: int) -> List[str]:
    """Identify areas of concern"""

    concerns = []

    if len(missing_skills) > 2:
        concerns.append(f"Missing {len(missing_skills)} required skills")
```

```
    if candidate_exp < required_exp:
        concerns.append(f"Experience below requirement ({candidate_exp} < {required_exp} years)")

    if not concerns:
        concerns.append("No major concerns identified")

    return concerns
```

---

20. src/models/schemas.py

python

```python
from pydantic import BaseModel, Field, EmailStr
from typing import List, Optional, Dict, Any
from datetime import datetime
from enum import Enum

class RecommendationLevel(str, Enum):
    STRONG_YES = "STRONG_YES"
    YES = "YES"
    MAYBE = "MAYBE"
    NO = "NO"
    UNABLE_TO_EVALUATE = "UNABLE_TO_EVALUATE"

class Skill(BaseModel):
    name: str
    category: Optional[str] = None
    years_experience: Optional[int] = None

class Experience(BaseModel):
    company: str
    position: str
    duration: str
    description: str
    technologies: List[str] = []

class Education(BaseModel):
    degree: str
    university: str
    graduation_year: int
    gpa: Optional[float] = None

class Resume(BaseModel):
    id: str
    name: str
    email: EmailStr
    phone: Optional[str] = None
    location: Optional[str] = None
    summary: str
    skills: List[str]
    experience_years: int
    experience: List[Experience]
    education: List[Education]

class JobProfile(BaseModel):
    id: str
    title: str
```

```
    title: str
    department: str
    location: str
    required_skills: List[str]
    preferred_skills: List[str] = []
    min_experience: int
    max_experience: Optional[int] = None

class EvaluationResult(BaseModel):
    candidate_name: str
    evaluation_method: str
    match_score: float
    matched_skills: List[str]
    missing_skills: List[str]
    recommendation: RecommendationLevel
    strengths: List[str]
    concerns: List[str]
    timestamp: datetime

class EvaluationLog(BaseModel):
    evaluation_id: str
    timestamp: datetime
    input_data: Dict[str, Any]
    result: EvaluationResult
    processing_time_ms: int
    used_fallback: bool
```

---

21. Dockerfile

```dockerfile
FROM python:3.9-slim

WORKDIR /app

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

# Copy requirements first for better caching
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Create necessary directories
RUN mkdir -p logs data/faiss_index data/embedding_cache

# Run the application
CMD ["python", "main.py"]
```

---

22. Makefile

makefile

```makefile
.PHONY: help install run test clean docker-build docker-run

help:
  @echo "Available commands:"
  @echo "  install      Install dependencies"
  @echo "  run          Run the application"
  @echo "  test         Run tests"
  @echo "  clean        Clean temporary files"
  @echo "  docker-build Build Docker image"
  @echo "  docker-run   Run with Docker Compose"

install:
  pip install -r requirements.txt

run:
  python main.py

test:
  pytest tests/ -v

clean:
  find . -type d -name __pycache__ -exec rm -rf {} +
  find . -type f -name "*.pyc" -delete
  rm -rf logs/*.json logs/*.log

docker-build:
  docker-compose build

docker-run:
  docker-compose up

docker-down:
  docker-compose down

setup:
  cp .env.example .env
  mkdir -p logs data/faiss_index data/embedding_cache config/job_profiles config/sample_resumes
  @echo "Setup complete! Edit .env file with your API keys"
```

---

23. setup.py

```python
from setuptools import setup, find_packages

setup(
    name="candidate-evaluation-system",
    version="1.0.0",
    packages=find_packages(where="src"),
    package_dir={"": "src"},
    install_requires=[
        "fastapi>=0.104.1",
        "uvicorn>=0.24.0",
        "pydantic>=2.5.0",
        "python-dotenv>=1.0.0",
        "openai>=1.12.0",
        "langchain>=0.1.0",
        "sentence-transformers>=2.3.1",
        "faiss-cpu>=1.7.4",
        "numpy>=1.24.3",
    ],
    python_requires=">=3.9",
)
```

Instructions to Create ZIP File:

1. Create the directory structure:

```bash
mkdir -p candidate-evaluation-system/{src,config,logs,data}
mkdir -p candidate-evaluation-system/src/{agents,mcp,embeddings,llm,models,utils}
mkdir -p candidate-evaluation-system/config/{job_profiles,sample_resumes}
```

2. Copy all the files to their respective locations as shown in the structure
3. Create the ZIP file:

```bash
zip -r candidate-evaluation-system.zip candidate-evaluation-system/
```

This system provides:

- ✅ GPT-4o integration with automatic fallback to dummy responses
- ✅ FAISS-based embeddings for similarity search
- ✅ MCP server for standardized tool access
- ✅ Local JSON logging (no external dependencies)
- ✅ Configuration-based job profiles and sample resumes
- ✅ Complete error handling and retry logic
- ✅ Docker support for easy deployment

To run the system:

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

The system will evaluate candidates and save results in the `logs/` folder as JSON files.