Statement of Work (SOW)

Voice-Based Greeting Agent with Facial Recognition

Two Architecture Approaches: Cloud API vs Self-Hosted (Free)

Project Name: AI-Powered Employee Greeting System
Version: 3.0 (Comprehensive - Both Approaches)
Date: September 29, 2025
Technology Stack: Python FastAPI + React Native + AI (Cloud or Self-Hosted)

---

1. Project Overview

We're building an intelligent greeting system that recognizes employees through facial detection, automatically registers new employees, and interacts with them using voice. The system is **dynamic** - when a new employee uses it for the first time, they can instantly register themselves without any admin intervention.

This document presents TWO architectural approaches:

Approach A: Cloud API-Based (Face++, Clarifai, Kairos)

- Quick setup and deployment
- Highly accurate (99.5%+)
- Scalable with minimal infrastructure
- Pay-per-use pricing
- Suitable for: Quick prototypes, small-to-medium deployments

Approach B: Self-Hosted Open Source (DeepFace)

- 100% free (no API costs)
- Complete control and privacy
- Works offline
- One-time setup effort
- Suitable for: Budget-conscious, privacy-focused, large-scale deployments

Both approaches support:

- Dynamic auto-registration of new employees
- Multi-language voice interaction
- Personalized greetings based on scenarios

- Cross-platform mobile app (iOS & Android)

---

2. Comparison: Approach A vs Approach B

Feature	Approach A: Cloud API	Approach B: Self-Hosted
Setup Time	⭐⭐⭐⭐⭐ (Fast - 2 days)	⭐⭐⭐ (Moderate - 4 days)
Accuracy	⭐⭐⭐⭐⭐ (99.5%)	⭐⭐⭐⭐ (97-99%)
Cost (Dev)	$0 (free tier)	$0
Cost (Prod 100 employees)	$10-30/month	$0-10/month (hosting only)
Cost (Prod 1000 employees)	$100-300/month	$10-20/month (hosting only)
Internet Required	✅ Yes	❌ No (works offline)
Privacy	⭐⭐⭐ (data sent to third party)	⭐⭐⭐⭐⭐ (100% local)
Scalability	⭐⭐⭐⭐ (API handles load)	⭐⭐⭐⭐ (depends on hardware)
Maintenance	⭐⭐⭐⭐⭐ (minimal)	⭐⭐⭐ (moderate)
Speed	⭐⭐⭐⭐ (1-2 sec)	⭐⭐⭐⭐⭐ (0.5-1 sec local)
Vendor Lock-in	⭐⭐ (dependent on API)	⭐⭐⭐⭐⭐ (independent)

---

3. Detailed Approach A: Cloud API-Based Architecture

3.1 Technology Stack (Approach A)

Face Recognition APIs

Primary: Face++ (Recommended) ⭐

- **Accuracy**: 99.5%
- **Free Tier**: 10,000 API calls/month
- **Pricing After Free Tier**:
  - Detect API: $0.0005/call
  - Compare API: $0.0005/call
  - Search API: $0.001/call
- **Features**:
  - Face detection and attributes
  - Face comparison (1:1)
  - Face search (1:N)
  - FaceSet management (store up to 10,000 faces per set)
- **Best For**: Production deployments with high accuracy requirements

Backup: Clarifai

- **Accuracy**: 98-99%
- **Free Tier**: 5,000 operations/month
- **Pricing After Free Tier**:
  - $1.20 per 1,000 operations
  - Volume discounts available
- **Features**:
  - Face detection
  - Face recognition
  - Custom model training
- **Best For**: Backup solution or if Face++ quota exceeded

Alternative: Kairos

- **Accuracy**: 97-98%
- **Free Tier**: Available (limited)
- **Pricing**: Contact for pricing
- **Features**:
  - Face recognition
  - Emotion detection
  - Demographics analysis
- **Best For**: Additional features like emotion detection

How Approach A Works

```
REGISTRATION FLOW:
1. Employee uploads photo via mobile app (one-time)
2. Backend receives image
3. Send image to Face++ Detect API → Get face_token
4. Send face_token to Face++ FaceSet Add API → Add to company faceset
5. Store face_token + employee_id mapping in database
6. Total: 2 API calls per employee registration

RECOGNITION FLOW:
1. Employee scans face
2. Backend sends image to Face++ Detect API → Get face_token
3. Send face_token to Face++ Search API with company faceset
4. Face++ returns matched employee_id (if found)
5. If match: Fetch employee details and show greeting
6. If no match: Prompt for registration
7. Total: 2 API calls per recognition
```

Cost Analysis (Approach A)

Development Phase:

```
Users: 10 test employees
Daily scans: 20 scans/day
Monthly API calls:
- Recognition: 2 calls × 20 scans × 30 days = 1,200 calls
- Registration: 2 calls × 10 employees = 20 calls (one-time)
- Total: 1,220 calls/month

Cost: $0 (well within 10,000 free tier)
```

Production - Small Office (100 employees, 50 scans/day):

```
Monthly API calls:
- Recognition: 2 calls × 50 scans × 30 days = 3,000 calls
- New registrations: ~5 new employees/month × 2 = 10 calls
- Total: 3,010 calls/month

Cost: $0 (within 10,000 free tier)
Additional costs:
- Database hosting: $5-10/month
- Backend hosting: $5-10/month
Total: $10-20/month
```

Production - Medium Office (500 employees, 200 scans/day):

```
Monthly API calls:
- Recognition: 2 calls × 200 scans × 30 days = 12,000 calls
- New registrations: ~10 new employees/month × 2 = 20 calls
- Total: 12,020 calls/month

Exceeds free tier by: 2,020 calls
API cost: 2,020 × $0.0005 = $1.01/month

Additional costs:
- Database hosting: $10/month
- Backend hosting: $10-20/month
Total: $21-31/month
```

Production - Large Office (1000 employees, 500 scans/day):

```
Monthly API calls:
- Recognition: 2 calls × 500 scans × 30 days = 30,000 calls
- Total: 30,000 calls/month

Exceeds free tier by: 20,000 calls
API cost: 20,000 × $0.0005 = $10/month

Additional costs:
- Database hosting: $15-20/month
- Backend hosting: $20-30/month
Total: $45-60/month
```

Implementation (Approach A)

```python
```

```
# app/services/face_service_api.py

import requests
from app.config import settings

class FaceRecognitionAPIService:
    def __init__(self):
        self.api_key = settings.FACEPP_API_KEY
        self.api_secret = settings.FACEPP_API_SECRET
        self.faceset_token = settings.FACEPP_FACESET_TOKEN

        # Face++ API endpoints
        self.detect_url = "https://api-us.faceplusplus.com/facepp/v3/detect"
        self.search_url = "https://api-us.faceplusplus.com/facepp/v3/search"
        self.faceset_add_url = "https://api-us.faceplusplus.com/facepp/v3/faceset/addface"

    async def detect_face(self, image_data: bytes) -> Optional[str]:
        """
        Detect face and return face_token
        Cost: $0.0005 per call (or free if within quota)
        """
        try:
            response = requests.post(
                self.detect_url,
                data={
                    'api_key': self.api_key,
                    'api_secret': self.api_secret
                },
                files={'image_file': image_data},
                timeout=10
            )

            result = response.json()

            if 'faces' in result and len(result['faces']) > 0:
                return result['faces'][0]['face_token']

            return None

        except Exception as e:
            print(f"Face detection error: {str(e)}")
            # Fallback to Clarifai if Face++ fails
            return await self.detect_face_clarifai(image_data)

    async def search_face(self, face_token: str) -> Optional[Dict]:
        """
        Search for matching face in FaceSet
        Cost: $0.001 per call (or free if within quota)
        """
        try:
            response = requests.post(
                self.search_url,
                data={
                    'api_key': self.api_key,
                    'api_secret': self.api_secret,
                    'face_token': face_token,
                    'faceset_token': self.faceset_token
                },
                timeout=10
            )

            result = response.json()

            if 'results' in result and len(result['results']) > 0:
                match = result['results'][0]
                if match['confidence'] > 75:  # 75% threshold
                    return {
                        'employee_id': match['user_id'],
                        'confidence': match['confidence']
                    }

            return None

        except Exception as e:
            print(f"Face search error: {str(e)}")
            return None

    async def add_face_to_faceset(self, face_token: str, employee_id: str):
        """
        Add face to company FaceSet
        Cost: $0.0005 per call (or free if within quota)
        """
        try:
            response = requests.post(
                self.faceset_add_url,
                data={
                    'api_key': self.api_key,
                    'api_secret': self.api_secret,
                    'faceset_token': self.faceset_token,
                    'face_tokens': face_token,
                    'user_id': employee_id  # Link face to employee
                },
                timeout=10
            )

            result = response.json()
            return result.get('face_added', 0) > 0

        except Exception as e:
            print(f"Add face error: {str(e)}")
            return False

    async def identify_or_register(self, image_data: bytes) -> Dict:
        """
        Main function: Identify employee or prompt registration
        """
        # Step 1: Detect face (1 API call)
        face_token = await self.detect_face(image_data)

        if not face_token:
            return {
                'status': 'error',
                'message': 'No face detected'
            }

        # Step 2: Search for match (1 API call)
        match = await self.search_face(face_token)

        if match:
            # Employee found!
            return {
                'status': 'recognized',
                'employee_id': match['employee_id'],
                'confidence': match['confidence']
            }
        else:
            # New employee - store face_token for registration
            return {
                'status': 'new_employee',
                'face_token': face_token,
                'needs_registration': True
            }
```

4. Detailed Approach B: Self-Hosted Open Source Architecture

4.1 Technology Stack (Approach B)

Face Recognition Library

Primary: DeepFace (Recommended) ⭐

- **Accuracy**: 97-99%
- **Cost**: 100% FREE
- **Models Available**:
  - VGG-Face: 98.95% accuracy, slower
  - Facenet: 99.20% accuracy, balanced (RECOMMENDED)
  - OpenFace: 93.80% accuracy, fastest
  - ArcFace: 99.40% accuracy, slowest but most accurate
- **Features**:
  - Face detection
  - Face recognition
  - Face verification
  - Age, gender, emotion detection
  - Works completely offline
- **Requirements**:
  - Python 3.7+
  - TensorFlow or PyTorch
  - 2-4GB RAM for models

Alternative: face_recognition

- **Accuracy**: 95-97%
- **Cost**: 100% FREE
- **Based on**: dlib library
- **Features**:
  - Simple API
  - Fast processing
  - Good for basic recognition
- **Requirements**:
  - Python 3.6+
  - dlib
  - Lower memory footprint

How Approach B Works

```
REGISTRATION FLOW:
1. Employee uploads photo via mobile app (one-time)
2. Backend receives image
3. DeepFace generates 128-dimensional face embedding (locally, free!)
4. Store embedding as binary (BYTEA) in PostgreSQL
5. Save profile image in local file system
6. Total: 0 API calls, 100% local processing

RECOGNITION FLOW:
1. Employee scans face
2. Backend receives image
3. DeepFace generates face embedding (locally, free!)
4. Compare with all stored embeddings in database
5. Find best match above threshold (60%+)
6. If match: Fetch employee details and show greeting
7. If no match: Prompt for registration
8. Total: 0 API calls, 100% local processing
```

Cost Analysis (Approach B)

Development Phase:

```
Users: 10 test employees
Daily scans: 20 scans/day

API Costs: $0 (no APIs used!)
Hosting: Local machine
Total: $0/month
```

Production - Small Office (100 employees, 50 scans/day):

```
API Costs: $0 (no APIs!)
Storage:
- Face embeddings: 100 × 512 bytes = 50 KB
- Profile images: 100 × 200 KB = 20 MB
- Database: < 100 MB

Server Requirements:
- CPU: 4 cores
- RAM: 8GB
- Storage: 50GB
- VPS Cost: $5-10/month (e.g., DigitalOcean, Linode)

Total: $5-10/month
```

Production - Medium Office (500 employees, 200 scans/day):

```
API Costs: $0
Storage:
- Face embeddings: 500 × 512 bytes = 250 KB
- Profile images: 500 × 200 KB = 100 MB
- Database: < 500 MB

Server Requirements:
- CPU: 6-8 cores
- RAM: 16GB
- Storage: 100GB
- VPS Cost: $10-20/month

Total: $10-20/month
```

Production - Large Office (1000 employees, 500 scans/day):

```
API Costs: $0
Storage:
- Face embeddings: 1000 × 512 bytes = 500 KB
- Profile images: 1000 × 200 KB = 200 MB
- Database: < 1 GB

Server Requirements:
- CPU: 8-12 cores
- RAM: 32GB
- Storage: 200GB
- GPU: Optional (speeds up 3-5x)
- VPS Cost: $20-40/month

Total: $20-40/month
```

Implementation (Approach B)

```
python
```

```
# app/services/face_service_deepface.py

from deepface import DeepFace
import numpy as np
from scipy.spatial.distance import cosine
import pickle
from typing import Optional, Dict, List, Tuple

class FaceRecognitionDeepFaceService:
    def __init__(self):
        # Use Facenet model (best balance of speed and accuracy)
        self.model_name = "Facenet"
        # Alternatives: "VGG-Face" (most accurate), "OpenFace" (fastest)

        self.threshold = 0.6  # 60% similarity threshold

    def generate_face_embedding(self, image_path: str) -> Optional[np.ndarray]:
        """
        Generate 128-dimensional face embedding locally
        Cost: $0 (completely free!)
        Time: ~0.5-1 second on CPU
        """
        try:
            embedding_objs = DeepFace.represent(
                img_path=image_path,
                model_name=self.model_name,
                enforce_detection=True,
                detector_backend='opencv'
            )

            if embedding_objs:
                # Returns 128-dimensional vector (512 bytes when stored)
                embedding = np.array(embedding_objs[0]["embedding"])
                return embedding

            return None

        except Exception as e:
            print(f"Face embedding error: {str(e)}")
            return None

    def compare_embeddings(
        self,
        embedding1: np.ndarray,
        embedding2: np.ndarray
    ) -> float:
```

```python
        """
        Compare two face embeddings using cosine similarity
        Returns: similarity score (0-1, higher = more similar)
        """
        distance = cosine(embedding1, embedding2)
        similarity = 1 - distance
        return similarity

    def find_matching_employee(
        self,
        captured_embedding: np.ndarray,
        stored_embeddings: List[Tuple[str, np.ndarray]]
    ) -> Optional[Tuple[str, float]]:
        """
        Find matching employee from stored embeddings
        Compares with all employees in database
        """
        best_match = None
        best_similarity = 0

        for employee_id, stored_embedding in stored_embeddings:
            similarity = self.compare_embeddings(
                captured_embedding,
                stored_embedding
            )

            if similarity > best_similarity and similarity > self.threshold:
                best_similarity = similarity
                best_match = employee_id

        if best_match:
            return best_match, best_similarity

        return None

    async def identify_or_register(
        self,
        image_path: str,
        all_embeddings: List[Tuple[str, np.ndarray]]
    ) -> Dict:
        """
        Main function: Identify employee or prompt registration
        100% local processing - no API costs!
        """
        # Step 1: Generate embedding locally (FREE!)
        captured_embedding = self.generate_face_embedding(image_path)
        if captured_embedding is None:
            return {
                'status': 'error',
                'message': 'No face detected'
            }

        # Step 2: Search for match locally (FREE!)
        match = self.find_matching_employee(captured_embedding, all_embeddings)

        if match:
            employee_id, confidence = match
            return {
                'status': 'recognized',
                'employee_id': employee_id,
                'confidence': float(confidence)
            }
        else:
            # New employee - store embedding temporarily
            temp_id = str(uuid.uuid4())
            # Cache embedding for registration
            cache_embedding(temp_id, captured_embedding)

            return {
                'status': 'new_employee',
                'temp_embedding_id': temp_id,
                'needs_registration': True
            }
```

---

5. Voice Services (Both Approaches)

5.1 Text-to-Speech Options

Option 1: Cloud-Based TTS (Approach A)

Google Cloud Text-to-Speech ⭐

- **Free Tier**: 1 million characters/month
- **Pricing**: $4 per 1 million characters after free tier
- **Voices**: 220+ voices in 40+ languages
- **Quality**: Excellent, natural-sounding
- **Languages**: English, Hindi, Marathi supported
- **Cost Estimate (100 employees, 50 greetings/day)**:
    - Average greeting: 100 characters

- Monthly: 50 × 30 × 100 = 150,000 characters
- Cost: $0 (within free tier)

Amazon Polly

- **Free Tier**: 5 million characters/month (first 12 months)
- **Pricing**: $4 per 1 million characters
- **Voices**: 60+ voices
- **Quality**: Excellent
- **Neural TTS**: Available for premium quality

Option 2: Self-Hosted TTS (Approach B)

pyttsx3 (Offline, Completely Free) ⭐

```python
pip install pyttsx3

import pyttsx3

def speak_greeting(text: str, language: str = 'en'):
    engine = pyttsx3.init()
    engine.setProperty('rate', 150)
    engine.setProperty('volume', 0.9)
    engine.say(text)
    engine.runAndWait()

# Cost: $0
# Internet: Not required
# Quality: Good (robotic but clear)
```

gTTS (Free, requires internet)

```python
```

```
pip install gtts

from gtts import gTTS

def generate_audio(text: str, language: str = 'en'):
    tts = gTTS(text=text, lang=language)
    tts.save('greeting.mp3')
    return 'greeting.mp3'

# Cost: $0
# Internet: Required
# Quality: Excellent (uses Google's engine)
```

AI4Bharat Indic TTS (For Indian Languages)

- Supports English, Hindi, Marathi
- Open-source and free
- Natural-sounding Indian voices
- Self-hosted

5.2 Speech-to-Text Options

Option 1: Cloud-Based STT (Approach A)

Google Cloud Speech-to-Text

- **Free Tier**: 60 minutes/month
- **Pricing**: $0.006 per 15 seconds after free tier
- **Languages**: 125+ languages including Hindi, Marathi
- **Accuracy**: Excellent for Indian accents
- **Cost Estimate (50 conversations/day, 30 sec each)**:
  - Monthly: 50 × 30 × 30 sec = 45,000 seconds = 750 minutes
  - Exceeds free tier by: 690 minutes
  - Cost: 690 min × 4 chunks × $0.006 = $16.56/month

AssemblyAI

- **Free Tier**: 5 hours/month
- **Pricing**: $0.00025 per second
- **Quality**: Excellent
- **Features**: Punctuation, speaker detection

Option 2: Self-Hosted STT (Approach B)

Vosk (Offline, Completely Free) ⭐

```python
pip install vosk

# Download model (one-time, ~50MB)
# https://alphacephei.com/vosk/models

import vosk
import json

def speech_to_text(audio_file: str) -> str:
    model = vosk.Model("model_path")
    rec = vosk.KaldiRecognizer(model, 16000)

    with open(audio_file, "rb") as f:
        while True:
            data = f.read(4000)
            if len(data) == 0:
                break
            rec.AcceptWaveform(data)

    result = json.loads(rec.FinalResult())
    return result['text']

# Cost: $0
# Internet: Not required
# Quality: Good (90-95% accuracy)
# Models available: English, Hindi, Marathi
```

Whisper by OpenAI (Self-hosted)

- Open-source, completely free
- Very high accuracy
- Multilingual support
- Can run on CPU or GPU

6. Complete Cost Comparison Table

Development Phase (10 employees, 20 scans/day)

Component	Approach A (Cloud API)	Approach B (Self-Hosted)
Face Recognition	$0 (free tier)	$0
Image Storage	$0 (local/free tier)	$0 (local)
TTS	$0 (free tier)	$0 (pyttsx3)
STT	$0 (free tier)	$0 (Vosk)
Database	$0 (local)	$0 (local)
Hosting	$0 (local)	$0 (local)
TOTAL	**$0/month**	**$0/month**

Production - Small (100 employees, 50 scans/day)

Component	Approach A (Cloud API)	Approach B (Self-Hosted)
Face Recognition	$0 (within free tier)	$0
Image Storage	Cloudinary: $0	Local: $0
TTS	$0 (within free tier)	$0 (pyttsx3/gTTS)
STT	$0 (within free tier)	$0 (Vosk)
Database	PostgreSQL: $5-10	PostgreSQL: $5-10
Backend Hosting	$5-10	$5-10 (VPS)
TOTAL	**$10-20/month**	**$5-10/month**

Production - Medium (500 employees, 200 scans/day)

Component	Approach A (Cloud API)	Approach B (Self-Hosted)
Face Recognition	$1-5	$0
Image Storage	$0-5	$0
TTS	$0 (free tier)	$0
STT	$10-20	$0
Database	$10-15	$10-15
Backend Hosting	$10-20	$10-20 (VPS)
TOTAL	**$31-65/month**	**$10-20/month**

Production - Large (1000 employees, 500 scans/day)

Component	Approach A (Cloud API)	Approach B (Self-Hosted)
Face Recognition	$10-20	$0

Component	Approach A (Cloud API)	Approach B (Self-Hosted)
Image Storage	$5-10	$0
TTS	$0-5	$0
STT	$20-40	$0
Database	$15-25	$15-25
Backend Hosting	$20-40	$20-40 (VPS)
TOTAL	**$70-140/month**	**$20-40/month**

---

7. Recommendation: Which Approach to Choose?

Choose Approach A (Cloud API) If:

✅ You need fastest time-to-market (2-4 days setup)
✅ You want highest accuracy (99.5%)
✅ Your team has limited AI/ML experience
✅ You have budget for API costs ($10-140/month based on scale)
✅ You need minimal maintenance
✅ You prefer vendor-managed infrastructure
✅ Internet connectivity is reliable

Choose Approach B (Self-Hosted) If:

✅ You want zero API costs (100% free recognition)
✅ Privacy is critical (no data leaves your servers)
✅ You need offline capability
✅ You have large-scale deployment (1000+ employees)
✅ Your team can manage AI model deployment
✅ You want complete control over the system
✅ Long-term cost savings are priority

Hybrid Approach (Recommended for Most Cases) ⭐

Start with Approach A, Migrate to Approach B Later

Phase 1 (Months 1-3):

- Use Approach A (Cloud API) for rapid prototyping
- Validate business requirements
- Gather user feedback
- Stay within free tiers

Phase 2 (Months 4-6):

- Migrate to Approach B (Self-Hosted) if:
  - User base grows beyond free tier
  - Monthly costs exceed $50
  - Privacy concerns arise
- Keep Approach A as backup/fallback

This gives you:

- ✅ Fast initial deployment
- ✅ Low risk
- ✅ Easy testing
- ✅ Future cost optimization
- ✅ Best of both worlds

---

8. Core Features (Both Approaches)

8.1 Dynamic Facial Recognition System

- Real-time face detection through mobile camera
- **Automatic new employee detection** (if face not recognized, prompt registration)
- Face matching against stored employee database
- Retrieve employee metadata upon successful recognition
- Support for multiple face photos per employee
- Confidence scoring for recognition accuracy

8.2 Self-Service Employee Registration

- **New employees can register themselves immediately**
- Simple registration form (Name, Email, DOB, Department, Position)
- Automatic face encoding generation and storage
- No admin approval needed for basic registration
- Optional admin review workflow
- Email verification (optional)

8.3 Personalized Greeting Scenarios

- **Birthday Greetings**: Special wishes on employee birthdays
- **Work Anniversary**: Congratulations on joining date anniversaries

- **Daily Greetings**: Time-based greetings (Good morning, afternoon, evening)
- **First Day Welcome**: Special greeting for newly registered employees
- **Weekend Greetings**: Happy Friday or weekend messages
- **Random Casual Greetings**: Keep interactions fresh and engaging
- **Custom Events**: Company anniversaries, festivals, achievements

8.4 Voice Interaction

- Text-to-Speech for system responses
- Speech-to-Text for employee input
- Natural conversation capability for casual chat
- Multi-language support (English, Hindi, Marathi)
- Context-aware responses
- Conversation history

8.5 Mobile Application Features

- Clean, modern design with vibrant colors
- Real-time camera feed display
- Visual feedback during recognition
- Smooth animations and transitions
- Employee information display cards
- Registration form for new employees
- Chat interface for conversations
- Voice recording and playback
- Cross-platform support (iOS & Android)

---

9. Database Schema (Both Approaches)

```sql
```

```sql
-- Employees Table (Same for both approaches)
CREATE TABLE employees (
  employee_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name VARCHAR(255) NOT NULL,
  email VARCHAR(255) UNIQUE NOT NULL,
  position VARCHAR(100),
  department VARCHAR(100),
  date_of_birth DATE,
  joining_date DATE NOT NULL DEFAULT CURRENT_DATE,
  phone_number VARCHAR(20),

  -- For Approach A: Store Face++ face_token (TEXT, ~50 bytes)
  -- For Approach B: Store DeepFace embedding (BYTEA, 512 bytes)
  face_data BYTEA,  -- Flexible to store either face_token or embedding
  face_data_type VARCHAR(20),  -- 'face_token' or 'embedding'

  -- Local file path for profile image
  profile_image_url VARCHAR(500),

  -- Registration metadata
  is_active BOOLEAN DEFAULT TRUE,
  is_self_registered BOOLEAN DEFAULT FALSE,
  registration_date TIMESTAMP DEFAULT NOW(),
  last_seen TIMESTAMP,

  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_employees_email ON employees(email);
CREATE INDEX idx_employees_active ON employees(is_active);
CREATE INDEX idx_employees_dob ON employees(date_of_birth);
CREATE INDEX idx_employees_joining ON employees(joining_date);

-- Face Encodings Table (for multiple photos per employee)
CREATE TABLE face_encodings (
  encoding_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  employee_id UUID REFERENCES employees(employee_id) ON DELETE CASCADE,
  face_data BYTEA NOT NULL,
  face_data_type VARCHAR(20) NOT NULL,
  image_url VARCHAR(500),
  encoding_quality FLOAT,
  is_primary BOOLEAN DEFAULT FALSE,
  created_at TIMESTAMP DEFAULT NOW()
);
```

```sql
CREATE INDEX idx_face_encodings_employee ON face_encodings(employee_id);
CREATE INDEX idx_face_encodings_primary ON face_encodings(is_primary);

-- Greetings Log Table
CREATE TABLE greetings_log (
    log_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    employee_id UUID REFERENCES employees(employee_id) ON DELETE SET NULL,
    greeting_type VARCHAR(50),
    greeting_text TEXT,
    recognition_confidence FLOAT,
    recognition_method VARCHAR(20),  -- 'api' or 'local'
    timestamp TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_greetings_employee ON greetings_log(employee_id);
CREATE INDEX idx_greetings_timestamp ON greetings_log(timestamp);
CREATE INDEX idx_greetings_type ON greetings_log(greeting_type);

-- Conversation History Table
CREATE TABLE conversation_history (
    conversation_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    employee_id UUID REFERENCES employees(employee_id) ON DELETE SET NULL,
    session_id UUID NOT NULL,
    user_input TEXT,
    system_response TEXT,
    intent_detected VARCHAR(100),
    language_used VARCHAR(10) DEFAULT 'en',
    timestamp TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_conversation_session ON conversation_history(session_id);
CREATE INDEX idx_conversation_employee ON conversation_history(employee_id);
CREATE INDEX idx_conversation_timestamp ON conversation_history(timestamp);

-- Special Events Table
CREATE TABLE special_events (
    event_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    employee_id UUID REFERENCES employees(employee_id) ON DELETE CASCADE,
    event_type VARCHAR(50) NOT NULL,
    event_date DATE NOT NULL,
    event_description TEXT,
    greeting_template TEXT,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_events_date ON special_events(event_date);
CREATE INDEX idx_events_employee ON special_events(employee_id);
CREATE INDEX idx_events_type ON special_events(event_type);

-- System Configuration Table (for switching between approaches)
CREATE TABLE system_config (
  config_key VARCHAR(100) PRIMARY KEY,
  config_value TEXT,
  updated_at TIMESTAMP DEFAULT NOW()
);

-- Insert default configuration
INSERT INTO system_config (config_key, config_value) VALUES
('recognition_method', 'api'),  -- 'api' or 'local'
('api_provider', 'facepp'),    -- 'facepp', 'clarifai', 'kairos'
('tts_provider', 'google'),    -- 'google', 'pyttsx3', 'gtts'
('stt_provider', 'google');    -- 'google', 'vosk'

-- API Usage Tracking (for cost monitoring)
CREATE TABLE api_usage_log (
  usage_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  api_name VARCHAR(50) NOT NULL,
  endpoint VARCHAR(100),
  request_count INTEGER DEFAULT 1,
  cost_estimate DECIMAL(10, 4),
  timestamp TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_api_usage_timestamp ON api_usage_log(timestamp);
CREATE INDEX idx_api_usage_name ON api_usage_log(api_name);
```

10. Backend Structure (Unified for Both Approaches)

```
backend/
├── app/
│   ├── __init__.py
│   ├── main.py            # FastAPI app initialization
│   ├── config.py          # Environment variables, settings
│   ├── database.py          # Database connection
│   │
│   ├── models/            # SQLAlchemy models
│   │   ├── __init__.py
│   │   ├── employee.py
│   │   ├── face_encoding.py
│   │   ├── greeting_log.py
│   │   ├── conversation.py
│   │   ├── special_event.py
│   │   └── system_config.py
│   │
│   ├── schemas/           # Pydantic schemas
│   │   ├── __init__.py
│   │   ├── employee.py
│   │   ├── greeting.py
│   │   ├── conversation.py
│   │   └── recognition.py
│   │
│   ├── api/               # API routes
│   │   ├── __init__.py
│   │   ├── endpoints/
│   │   │   ├── recognition.py  # Face recognition & registration
│   │   │   ├── employees.py    # Employee management
│   │   │   ├── greeting.py     # Greeting generation
│   │   │   ├── conversation.py # Chat endpoints
│   │   │   ├── voice.py        # TTS/STT integration
│   │   │   └── admin.py        # Admin configuration
│   │
│   ├── services/          # Business logic
│   │   ├── __init__.py
│   │   ├── face_service_factory.py    # Factory to choose approach
│   │   ├── face_service_api.py        # Approach A (Face++)
│   │   ├── face_service_deepface.py   # Approach B (DeepFace)
│   │   ├── employee_service.py        # Employee CRUD
│   │   ├── greeting_service.py        # Greeting logic
│   │   ├── nlp_service.py             # Conversation handling
│   │   ├── voice_service.py           # TTS/STT abstraction
│   │   └── api_cost_tracker.py        # Track API usage costs
│   │
│   ├── utils/             # Helper functions
│   │   ├── __init__.py
│   │   ├── image_processing.py
│   │   ├── date_utils.py
│   │   └── response_templates.py
│   │
│   ├── core/              # Core functionality
│   │   ├── __init__.py
│   │   ├── security.py        # JWT, authentication
│   │   └── exceptions.py      # Custom exceptions
│   │
│   ├── uploads/           # Local image storage
│   │   ├── employee_faces/
│   │   └── temp/
│   │
│   └── cache/              # Temporary data
│       ├── temp_embeddings/
│       └── audio_cache/
│
├── alembic/                # Database migrations
├── tests/                  # Unit tests
├── requirements.txt        # Python dependencies
├── requirements-api.txt    # Additional for Approach A
├── requirements-local.txt  # Additional for Approach B
├── Dockerfile
└── .env                    # Environment variables
```

---

11. Implementation Roadmap (20 Days)

Phase 1: Core Backend Development (Days 1-5)

Day 1: Environment & Database Setup

- Set up Python virtual environment
- Install FastAPI and core dependencies
- Create PostgreSQL database
- Set up SQLAlchemy models
- Configure Alembic migrations
- Run initial migrations
- Set up environment variables

Day 2: Approach A Implementation (Cloud API)

- Register for Face++ API account
- Obtain API keys
- Implement Face++ integration service
- Create FaceSet in Face++
- Test face detection and search APIs
- Implement fallback to Clarifai
- Test with sample photos

Day 3: Approach B Implementation (Self-Hosted)

- Install DeepFace and dependencies
- Download and configure models (Facenet)

- Implement DeepFace integration service
- Test face embedding generation
- Test face comparison locally
- Optimize for performance

Day 4: Service Factory & Dynamic Registration

- Create face service factory (switch between approaches)
- Implement employee service
- Build "identify or register" logic
- Implement auto-registration workflow
- Test both approaches
- Add configuration switching

Day 5: Greeting & Scenario Logic

- Build greeting generation service
- Implement all greeting scenarios
- Create greeting templates
- Add randomization
- Test scenario detection
- Implement API endpoints

Deliverable: Functional backend with both face recognition approaches

---

Phase 2: Voice Services (Days 6-8)

Day 6: TTS Implementation

- Integrate Google Cloud TTS (Approach A)
- Implement pyttsx3 (Approach B)
- Add gTTS as alternative
- Create TTS service abstraction
- Test with multiple languages
- Implement audio caching

Day 7: STT Implementation

- Integrate Google Speech-to-Text (Approach A)

- Install and configure Vosk (Approach B)
- Download Vosk models for English, Hindi, Marathi
- Create STT service abstraction
- Test voice recognition accuracy

Day 8: NLP & Conversation

- Implement rule-based conversation system
- Create intent detection
- Build response templates
- Add conversation history storage
- Test end-to-end voice interaction
- Optimize response times

Deliverable: Complete voice interaction system with both approaches

---

Phase 3: Mobile App Development (Days 9-14)

Day 9: React Native Setup & Navigation

- Initialize React Native project
- Set up navigation structure
- Configure required libraries
- Set up state management (Redux/Zustand)
- Create app theme and styling
- Configure environment variables

Day 10: Camera & Permissions

- Implement camera screen with react-native-vision-camera
- Configure camera permissions (iOS & Android)
- Add face detection overlay
- Implement image capture
- Test camera functionality on devices

Day 11: Face Recognition Integration

- Connect camera to backend API
- Implement image upload

- Handle recognition responses
- Show loading states
- Handle errors gracefully
- Test with real backend

Day 12: Registration Flow

- Build registration form UI
- Implement form validation
- Add date pickers
- Connect to backend registration API
- Handle registration success/error
- Test complete registration flow

Day 13: Greeting & Employee Display

- Build greeting display screen
- Add animations for greetings
- Implement employee card component
- Integrate TTS for audio playback
- Add visual feedback
- Test greeting scenarios

Day 14: Chat Interface

- Create chat UI with message bubbles
- Implement voice recording
- Integrate STT for voice input
- Add text input as alternative
- Connect to conversation API
- Add typing indicators

Deliverable: Complete mobile app with all features

Phase 4: Testing & Optimization (Days 15-17)

Day 15: Integration Testing

- Test complete user flows:
    - New employee registration

  - Existing employee recognition
  - Voice greetings
  - Conversations
- Test on Android devices
- Test on iOS devices (if applicable)
- Verify both recognition approaches
- Test switching between approaches

Day 16: Performance Optimization

- Optimize face recognition speed
- Reduce image upload sizes
- Implement caching strategies
- Optimize database queries
- Improve API response times
- Test under load (simulate 50+ concurrent users)

Day 17: Bug Fixes & Polish

- Fix identified bugs
- Improve UI animations
- Enhance error messages
- Add offline detection
- Implement retry logic
- Test edge cases
- User acceptance testing with 10-20 users

Deliverable: Production-ready, tested application

Phase 5: Deployment & Documentation (Days 18-20)

Day 18: Backend Deployment

- Choose hosting provider based on approach:
  - Approach A: Railway, Render, or AWS
  - Approach B: VPS (DigitalOcean, Linode) with more resources
- Set up production server
- Configure PostgreSQL database

- Deploy backend application
- Set up SSL certificate
- Configure firewall and security
- Test production endpoints

Day 19: Mobile App Build & Distribution

- Build Android APK (release mode)
- Sign Android app
- Build iOS app (if applicable)
- Test builds on physical devices
- Set up Firebase App Distribution
- Create app store assets (icons, screenshots)
- Distribute to test users

Day 20: Documentation & Training

- Create comprehensive API documentation
- Write deployment guide
- Document system configuration
- Create user manual with screenshots
- Write admin guide
- Document cost monitoring procedures
- Create training materials for employees
- Prepare handover documentation

Deliverable: Deployed application with complete documentation

---

12. API Endpoints Reference

Recognition Endpoints

```
POST   /api/v1/recognition/scan-face
    Description: Main endpoint - Scan face and recognize or register
    Request: multipart/form-data (image file)
    Response: {
      status: 'recognized' | 'new_employee' | 'error',
      employee: {...} (if recognized),
      temp_id: '...' (if new employee),
      confidence: 0.95 (if recognized)
    }

POST   /api/v1/recognition/register
    Description: Register new employee with face
    Request: {
     temp_id: string,
     name: string,
     email: string,
     position?: string,
     department?: string,
     date_of_birth?: string,
     image: file
    }
    Response: {
     status: 'success',
     employee_id: string
    }

POST   /api/v1/recognition/add-face
    Description: Add additional face photo for existing employee
    Request: {
     employee_id: string,
     image: file
    }
```

Employee Endpoints

```
GET    /api/v1/employees
    Description: List all active employees
    Query Params: ?page=1&limit=50&search=john

GET    /api/v1/employees/{id}
    Description: Get employee details

PUT    /api/v1/employees/{id}
    Description: Update employee information

DELETE /api/v1/employees/{id}
    Description: Deactivate employee
```

Greeting Endpoints

```
GET    /api/v1/greetings/generate/{employee_id}
    Description: Generate personalized greeting
```

```
GET   /api/v1/greetings/history
    Description: Get greeting history
    Query Params: ?employee_id=xxx&from=2025-01-01
```

Conversation Endpoints

```
POST   /api/v1/conversation/chat
    Description: Send text message
    Request: {
      employee_id: string,
      message: string,
      session_id: string
    }

POST   /api/v1/conversation/voice
    Description: Send voice message
    Request: {
      employee_id: string,
      audio: file,
      session_id: string
    }

GET   /api/v1/conversation/history/{session_id}
    Description: Get conversation history
```

Voice Endpoints

```
POST   /api/v1/voice/tts
    Description: Convert text to speech
    Request: {
      text: string,
      language: 'en' | 'hi' | 'mr'
    }
    Response: {
      audio_url: string
    }

POST   /api/v1/voice/stt
    Description: Convert speech to text
    Request: multipart/form-data (audio file)
    Response: {
      text: string,
      confidence: 0.95
    }
```

Admin Endpoints

```
GET    /api/v1/admin/config
       Description: Get system configuration

PUT    /api/v1/admin/config
       Description: Update system configuration
       Request: {
         recognition_method: 'api' | 'local',
         api_provider: 'facepp' | 'clarifai',
         tts_provider: 'google' | 'pyttsx3',
         stt_provider: 'google' | 'vosk'
       }

GET    /api/v1/admin/usage-stats
       Description: Get API usage statistics and costs

GET    /api/v1/admin/dashboard
       Description: Get dashboard metrics
```

---

13. Environment Variables Configuration

```
bash
```

```
# .env file

# Database
DATABASE_URL=postgresql://user:password@localhost:5432/greeting_db
DB_POOL_SIZE=20

# JWT Authentication
SECRET_KEY=your-secret-key-here
ALGORITHM=HS256
ACCESS_TOKEN_EXPIRE_MINUTES=30

# System Configuration
RECOGNITION_METHOD=api  # or 'local'
API_PROVIDER=facepp     # or 'clarifai', 'kairos'

# Face++ API (Approach A)
FACEPP_API_KEY=your-facepp-api-key
FACEPP_API_SECRET=your-facepp-api-secret
FACEPP_FACESET_TOKEN=your-faceset-token

# Clarifai API (Approach A - Backup)
CLARIFAI_API_KEY=your-clarifai-api-key

# Google Cloud APIs (Approach A)
GOOGLE_TTS_API_KEY=your-google-tts-key
GOOGLE_STT_API_KEY=your-google-stt-key
GOOGLE_APPLICATION_CREDENTIALS=path/to/credentials.json

# DeepFace Configuration (Approach B)
DEEPFACE_MODEL=Facenet  # or 'VGG-Face', 'ArcFace'
FACE_SIMILARITY_THRESHOLD=0.6

# Voice Configuration
TTS_PROVIDER=google     # or 'pyttsx3', 'gtts'
STT_PROVIDER=google     # or 'vosk'
VOSK_MODEL_PATH=models/vosk-model-small-en-us-0.15

# File Storage
UPLOAD_DIR=app/uploads
TEMP_DIR=app/temp
CACHE_DIR=app/cache

# Server
HOST=0.0.0.0
PORT=8000
ENVIRONMENT=development  # or 'production'
```

```
DEBUG=True

# CORS
ALLOWED_ORIGINS=http://localhost:3000,http://localhost:19006

# Cost Tracking
TRACK_API_COSTS=True
MONTHLY_BUDGET_ALERT=50  # USD
```

---

14. Monitoring & Cost Management

14.1 API Cost Tracking Service

```
python
```

```
# app/services/api_cost_tracker.py

from datetime import datetime, timedelta
from sqlalchemy.orm import Session
from app.models.api_usage_log import APIUsageLog

class APICostTracker:
    # Cost per API call (in USD)
    COSTS = {
        'facepp_detect': 0.0005,
        'facepp_search': 0.001,
        'facepp_compare': 0.0005,
        'clarifai_detect': 0.0012,
        'google_tts': 0.000004,  # per character
        'google_stt': 0.0004,    # per 15 seconds
    }

    def __init__(self, db: Session):
        self.db = db

    def log_api_call(self, api_name: str, endpoint: str):
        """Log API usage for cost tracking"""
        cost = self.COSTS.get(api_name, 0)

        usage = APIUsageLog(
            api_name=api_name,
            endpoint=endpoint,
            request_count=1,
            cost_estimate=cost,
            timestamp=datetime.now()
        )

        self.db.add(usage)
        self.db.commit()

    def get_monthly_cost(self) -> Dict:
        """Get current month's API costs"""
        start_date = datetime.now().replace(day=1, hour=0, minute=0, second=0)

        logs = self.db.query(APIUsageLog).filter(
            APIUsageLog.timestamp >= start_date
        ).all()

        total_cost = sum(log.cost_estimate for log in logs)
        breakdown = {}
        for log in logs:
            if log.api_name not in breakdown:
                breakdown[log.api_name] = {
                    'calls': 0,
                    'cost': 0
                }
            breakdown[log.api_name]['calls'] += log.request_count
            breakdown[log.api_name]['cost'] += log.cost_estimate

        return {
            'total_cost': round(total_cost, 2),
            'breakdown': breakdown,
            'period': 'current_month'
        }

    def check_budget_alert(self, budget_limit: float = 50):
        """Check if monthly cost exceeds budget"""
        cost_data = self.get_monthly_cost()

        if cost_data['total_cost'] > budget_limit:
            # Send alert (email, SMS, etc.)
            return {
                'alert': True,
                'message': f"Monthly cost ${cost_data['total_cost']} exceeds budget ${budget_limit}",
                'recommendation': 'Consider switching to self-hosted approach (Approach B)'
            }

        return {'alert': False}
```

14.2 Admin Dashboard Metrics

```python
```

```
# app/api/endpoints/admin.py

@router.get("/dashboard")
async def get_dashboard_metrics(db: Session = Depends(get_db)):
    """Get system metrics for admin dashboard"""

    # Employee metrics
    total_employees = db.query(Employee).filter(Employee.is_active == True).count()
    new_this_month = db.query(Employee).filter(
        Employee.registration_date >= datetime.now().replace(day=1)
    ).count()

    # Recognition metrics
    today = datetime.now().date()
    recognitions_today = db.query(GreetingLog).filter(
        GreetingLog.timestamp >= today
    ).count()

    # Cost metrics
    cost_tracker = APICostTracker(db)
    monthly_cost = cost_tracker.get_monthly_cost()

    # System configuration
    config = db.query(SystemConfig).all()
    system_config = {c.config_key: c.config_value for c in config}

    return {
        'employees': {
            'total': total_employees,
            'new_this_month': new_this_month
        },
        'usage': {
            'recognitions_today': recognitions_today,
            'greetings_this_month': db.query(GreetingLog).filter(
                GreetingLog.timestamp >= datetime.now().replace(day=1)
            ).count()
        },
        'costs': monthly_cost,
        'system': system_config
    }
```

15. Migration Guide: Switching Between Approaches

15.1 From Approach A to Approach B

Reasons to switch:

- Monthly API costs exceeding $50
- Privacy/compliance requirements
- Need for offline capability
- Scaling to 500+ employees

Migration steps:

1. **Install DeepFace dependencies**

```bash
pip install deepface tensorflow opencv-python
```

2. **Generate embeddings for existing employees**

```python
# Migration script
from app.services.face_service_deepface import FaceRecognitionDeepFaceService

face_service = FaceRecognitionDeepFaceService()

# For each employee with face_token
for employee in employees:
    # Retrieve original image
    image_path = employee.profile_image_url

    # Generate DeepFace embedding
    embedding = face_service.generate_face_embedding(image_path)

    # Update database
    employee.face_data = pickle.dumps(embedding)
    employee.face_data_type = 'embedding'
    db.commit()
```

3. **Update system configuration**

```sql
```

```
UPDATE system_config
SET config_value = 'local'
WHERE config_key = 'recognition_method';
```

4. **Test recognition**

```python
# Test with sample employees
# Verify accuracy matches or exceeds previous approach
```

5. **Monitor and optimize**

```python
# Check recognition speed
# Optimize threshold if needed
# Add more face photos per employee if accuracy is low
```

15.2 From Approach B to Approach A

Reasons to switch:

- Need higher accuracy
- Limited server resources
- Want to reduce maintenance
- Prefer managed solution

Migration steps:

1. **Set up Face++ account**

- Create account and get API keys
- Create FaceSet

2. **Upload existing embeddings to Face++**

```python
```

```
# Migration script
for employee in employees:
    # Get original image
    image_path = employee.profile_image_url

    with open(image_path, 'rb') as f:
        image_data = f.read()

    # Send to Face++
    face_token = face_service_api.detect_face(image_data)
    face_service_api.add_face_to_faceset(face_token, employee.employee_id)

    # Update database
    employee.face_data = face_token.encode()
    employee.face_data_type = 'face_token'
    db.commit()
```

3. **Update configuration**

```sql
sql

UPDATE system_config
SET config_value = 'api'
WHERE config_key = 'recognition_method';
```

---

16. Testing Strategy

16.1 Unit Tests

```python
python
```

```
# tests/test_face_recognition.py

import pytest
from app.services.face_service_deepface import FaceRecognitionDeepFaceService

def test_face_embedding_generation():
    service = FaceRecognitionDeepFaceService()
    embedding = service.generate_face_embedding('tests/fixtures/face1.jpg')

    assert embedding is not None
    assert len(embedding) == 128  # Facenet produces 128-dim vectors
    assert embedding.dtype == np.float64

def test_face_comparison():
    service = FaceRecognitionDeepFaceService()
    embedding1 = service.generate_face_embedding('tests/fixtures/face1.jpg')
    embedding2 = service.generate_face_embedding('tests/fixtures/face1_different_angle.jpg')

    similarity = service.compare_embeddings(embedding1, embedding2)
    assert similarity > 0.7  # Same person should have high similarity

def test_different_faces():
    service = FaceRecognitionDeepFaceService()
    embedding1 = service.generate_face_embedding('tests/fixtures/person1.jpg')
    embedding2 = service.generate_face_embedding('tests/fixtures/person2.jpg')

    similarity = service.compare_embeddings(embedding1, embedding2)
    assert similarity < 0.5  # Different people should have low similarity
```

16.2 Integration Tests

python

```python
# tests/test_api_endpoints.py

import pytest
from fastapi.testclient import TestClient
from app.main import app

client = TestClient(app)

def test_scan_face_new_employee():
    with open('tests/fixtures/new_face.jpg', 'rb') as f:
        response = client.post(
            '/api/v1/recognition/scan-face',
            files={'image': f}
        )

    assert response.status_code == 200
    data = response.json()
    assert data['status'] == 'new_employee'
    assert 'temp_id' in data

def test_register_employee():
    # First scan
    with open('tests/fixtures/new_face.jpg', 'rb') as f:
        scan_response = client.post(
            '/api/v1/recognition/scan-face',
            files={'image': f}
        )

    temp_id = scan_response.json()['temp_id']

    # Then register
    with open('tests/fixtures/new_face.jpg', 'rb') as f:
        register_response = client.post(
            '/api/v1/recognition/register',
            data={
                'temp_id': temp_id,
                'name': 'Test Employee',
                'email': 'test@company.com'
            },
            files={'image': f}
        )

    assert register_response.status_code == 200
    data = register_response.json()
    assert data['status'] == 'success'
```

16.3 Performance Tests

```python
# tests/test_performance.py

import time
import pytest

def test_recognition_speed_local():
    """Test that local recognition completes in under 1 second"""
    service = FaceRecognitionDeepFaceService()

    start = time.time()
    embedding = service.generate_face_embedding('tests/fixtures/face1.jpg')
    end = time.time()

    assert (end - start) < 1.0  # Should complete in under 1 second

def test_recognition_speed_api():
    """Test that API recognition completes in under 2 seconds"""
    service = FaceRecognitionAPIService()

    with open('tests/fixtures/face1.jpg', 'rb') as f:
        image_data = f.read()

    start = time.time()
    face_token = service.detect_face(image_data)
    end = time.time()

    assert (end - start) < 2.0  # Should complete in under 2 seconds
```

---

17. Security Considerations

17.1 Data Protection

Face Data Security:

- Encrypt face embeddings at rest
- Use HTTPS for all API communications
- Implement rate limiting on recognition endpoints
- Log all recognition attempts with timestamps
- Auto-expire temporary registration data after 1 hour

Access Control:

```python
# Implement JWT authentication for sensitive endpoints
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials

security = HTTPBearer()

@router.delete("/employees/{employee_id}")
async def delete_employee(
    employee_id: str,
    credentials: HTTPAuthorizationCredentials = Depends(security),
    db: Session = Depends(get_db)
):
    # Verify JWT token
    # Only admins can delete employees
    pass
```

17.2 Privacy Compliance

GDPR Compliance:

- Obtain explicit consent before storing face data
- Provide data export functionality
- Implement right to be forgotten (delete employee data)
- Log all data access
- Data retention policies

```python
```

```
@router.post("/employees/{employee_id}/request-data-deletion")
async def request_data_deletion(employee_id: str, db: Session = Depends(get_db)):
    """GDPR: Right to be forgotten"""
    employee = db.query(Employee).filter_by(employee_id=employee_id).first()

    if employee:
        # Delete face data
        employee.face_data = None
        employee.is_active = False
        employee.deleted_at = datetime.now()

        # Delete profile image
        if employee.profile_image_url:
            os.remove(employee.profile_image_url)

        # Anonymize personal data
        employee.email = f"deleted_{employee_id}@removed.com"
        employee.phone_number = None

        db.commit()

        return {"status": "success", "message": "Data deletion completed"}
```

---

18. Deployment Guide

18.1 Approach A Deployment (Cloud API)

Requirements:

- Linux server (Ubuntu 20.04+)
- 2 CPU cores, 4GB RAM
- Python 3.9+
- PostgreSQL 14+
- Nginx (reverse proxy)

Step-by-step:

```bash
```

```
# 1. Update system
sudo apt update && sudo apt upgrade -y

# 2. Install dependencies
sudo apt install python3-pip postgresql postgresql-contrib nginx -y

# 3. Create database
sudo -u postgres psql
CREATE DATABASE greeting_db;
CREATE USER greeting_user WITH PASSWORD 'secure_password';
GRANT ALL PRIVILEGES ON DATABASE greeting_db TO greeting_user;
\q

# 4. Clone repository
git clone <your-repo-url>
cd backend

# 5. Create virtual environment
python3 -m venv venv
source venv/bin/activate

# 6. Install dependencies
pip install -r requirements.txt
pip install -r requirements-api.txt

# 7. Configure environment
cp .env.example .env
nano .env
# Set all required variables

# 8. Run migrations
alembic upgrade head

# 9. Start with systemd
sudo nano /etc/systemd/system/greeting-api.service
```

Systemd service file:

```
ini
```

```
[Unit]
Description=Greeting Agent API
After=network.target

[Service]
User=www-data
Group=www-data
WorkingDirectory=/var/www/greeting-backend
Environment="PATH=/var/www/greeting-backend/venv/bin"
ExecStart=/var/www/greeting-backend/venv/bin/uvicorn app.main:app --host 0.0.0.0 --port 8000

[Install]
WantedBy=multi-user.target
```

```bash
# 10. Enable and start service
sudo systemctl enable greeting-api
sudo systemctl start greeting-api

# 11. Configure Nginx
sudo nano /etc/nginx/sites-available/greeting-api
```

Nginx configuration:

```nginx
server {
    listen 80;
    server_name api.yourcompany.com;

    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }

    location /uploads {
        alias /var/www/greeting-backend/app/uploads;
    }
}
```

```bash
```

```
# 12. Enable Nginx site
sudo ln -s /etc/nginx/sites-available/greeting-api /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl restart nginx

# 13. Setup SSL with Let's Encrypt
sudo apt install certbot python3-certbot-nginx -y
sudo certbot --nginx -d api.yourcompany.com

# 14. Setup automated backups
sudo crontab -e
# Add: 0 2 * * * pg_dump greeting_db > /backup/db_$(date +\%Y\%m\%d).sql
```

18.2 Approach B Deployment (Self-Hosted)

Requirements:

- Linux server (Ubuntu 20.04+)
- 8 CPU cores, 16GB RAM (for optimal performance)
- 100GB storage
- Python 3.9+
- PostgreSQL 14+
- Nginx
- Optional: NVIDIA GPU (3-5x faster)

Additional steps for Approach B:

```
bash
```

```
# 1-10: Same as Approach A

# 11. Install AI/ML dependencies
pip install -r requirements-local.txt

# 12. Download DeepFace models (one-time, ~200MB)
python -c "from deepface import DeepFace; DeepFace.build_model('Facenet')"

# 13. (Optional) Install CUDA for GPU acceleration
# Follow NVIDIA CUDA installation guide for your GPU

# 14. (Optional) Download Vosk models for offline STT
cd app/models
wget https://alphacephei.com/vosk/models/vosk-model-small-en-us-0.15.zip
unzip vosk-model-small-en-us-0.15.zip
wget https://alphacephei.com/vosk/models/vosk-model-small-hi-0.22.zip
unzip vosk-model-small-hi-0.22.zip

# 15. Optimize for production
# Update .env
RECOGNITION_METHOD=local
DEEPFACE_MODEL=Facenet
TTS_PROVIDER=pyttsx3
STT_PROVIDER=vosk

# 16. Continue with steps 12-14 from Approach A
```

18.3 Docker Deployment (Both Approaches)

Dockerfile:

```
dockerfile
```

```
FROM python:3.9-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    gcc \
    g++ \
    libpq-dev \
    libsm6 \
    libxext6 \
    libxrender-dev \
    libgomp1 \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements
COPY requirements.txt requirements-api.txt requirements-local.txt ./

# Install Python dependencies (choose based on approach)
RUN pip install --no-cache-dir -r requirements.txt
# For Approach A: RUN pip install --no-cache-dir -r requirements-api.txt
# For Approach B: RUN pip install --no-cache-dir -r requirements-local.txt

# Copy application
COPY . .

# Create necessary directories
RUN mkdir -p app/uploads app/cache app/temp

# Expose port
EXPOSE 8000

# Run application
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

docker-compose.yml:

```
yaml
```

```
version: '3.8'

services:
  db:
    image: postgres:14
    environment:
      POSTGRES_DB: greeting_db
      POSTGRES_USER: greeting_user
      POSTGRES_PASSWORD: secure_password
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

  api:
    build: .
    ports:
      - "8000:8000"
    depends_on:
      - db
    environment:
      DATABASE_URL: postgresql://greeting_user:secure_password@db:5432/greeting_db
      RECOGNITION_METHOD: api  # or 'local'
    env_file:
      - .env
    volumes:
      - ./app/uploads:/app/app/uploads
      - ./app/cache:/app/app/cache

volumes:
  postgres_data:
```

Deploy with Docker:

```
bash
```

```
# Build and start
docker-compose up -d

# Run migrations
docker-compose exec api alembic upgrade head

# View logs
docker-compose logs -f api

# Stop
docker-compose down
```

---

19. Mobile App Deployment

19.1 Android Build

```bash
# 1. Navigate to mobile app directory
cd mobile-app

# 2. Install dependencies
npm install

# 3. Configure environment
cp .env.example .env
nano .env
# Set API_BASE_URL to production backend

# 4. Generate Android bundle
cd android
./gradlew bundleRelease

# Output: android/app/build/outputs/bundle/release/app-release.aab

# 5. Generate APK (for direct installation)
./gradlew assembleRelease

# Output: android/app/build/outputs/apk/release/app-release.apk

# 6. Sign the APK
jarsigner -verbose -sigalg SHA256withRSA -digestalg SHA-256 \
  -keystore my-release-key.keystore \
  app-release.apk alias_name
```

19.2 iOS Build (macOS only)

```bash
# 1. Install CocoaPods dependencies
cd ios
pod install
cd ..

# 2. Open Xcode
open ios/YourAppName.xcworkspace

# 3. Configure signing in Xcode
# - Select project in navigator
# - Select target
# - Go to "Signing & Capabilities"
# - Select your team and provisioning profile

# 4. Build for release
# Product > Archive
# Distribute App > App Store Connect

# 5. Upload to TestFlight
# Follow Xcode prompts
```

19.3 Internal Distribution (Firebase App Distribution)

```bash
# 1. Install Firebase CLI
npm install -g firebase-tools

# 2. Login to Firebase
firebase login

# 3. Initialize Firebase in project
firebase init

# 4. Upload APK to Firebase
firebase appdistribution:distribute \
  android/app/build/outputs/apk/release/app-release.apk \
  --app YOUR_FIREBASE_APP_ID \
  --groups "testers" \
  --release-notes "Initial release with face recognition and voice greetings"

# 5. Testers receive email to download app
```

20. Maintenance & Monitoring

20.1 Regular Maintenance Tasks

Daily:

- Check error logs
- Monitor API usage and costs
- Review recognition accuracy metrics
- Check system uptime

Weekly:

- Database backup verification
- Security updates
- Performance optimization review
- User feedback collection

Monthly:

- Cost analysis and optimization
- Feature usage analytics
- Update dependencies
- Comprehensive testing

20.2 Monitoring Setup

Backend Monitoring with Prometheus & Grafana:

python

```
# Install prometheus client
pip install prometheus-client

# Add to FastAPI app
from prometheus_client import Counter, Histogram, make_asgi_app

# Metrics
recognition_requests = Counter('recognition_requests_total', 'Total recognition requests')
recognition_duration = Histogram('recognition_duration_seconds', 'Recognition request duration')
api_cost = Counter('api_cost_total', 'Total API cost in USD')

# Mount metrics endpoint
metrics_app = make_asgi_app()
app.mount("/metrics", metrics_app)

# Use in endpoints
@router.post("/scan-face")
async def scan_face():
    recognition_requests.inc()

    with recognition_duration.time():
        # Process recognition
        pass
```

Log Aggregation:

```python
# Configure structured logging
import logging
from pythonjsonlogger import jsonlogger

logger = logging.getLogger()
logHandler = logging.StreamHandler()
formatter = jsonlogger.JsonFormatter()
logHandler.setFormatter(formatter)
logger.addHandler(logHandler)

# Log important events
logger.info("Face recognition successful", extra={
    "employee_id": employee_id,
    "confidence": confidence_score,
    "method": "api",  # or "local"
    "timestamp": datetime.now().isoformat()
})
```

20.3 Alerting

Set up alerts for:

1. **High API Costs**

```python
if monthly_cost > BUDGET_THRESHOLD:
    send_alert("API costs exceed budget!")
```

2. **Low Recognition Accuracy**

```python
if avg_confidence < 0.75:
    send_alert("Recognition accuracy degraded!")
```

3. **System Downtime**

```bash
# Setup uptime monitoring with UptimeRobot or Pingdom
```

4. **Database Issues**

```python
if db_connection_failures > 5:
    send_alert("Database connectivity issues!")
```

---

21. Troubleshooting Guide

21.1 Common Issues

Issue 1: Face Not Detected

Symptoms: API returns "No face detected"

Causes:

- Poor lighting
- Face too far or too close
- Multiple faces in frame
- Low image quality

- Face partially covered

Solutions:

```python
# Add better error messages
if not face_detected:
    return {
        "status": "error",
        "message": "No face detected",
        "tips": [
            "Ensure good lighting",
            "Look directly at camera",
            "Remove glasses if possible",
            "Move closer to camera"
        ]
    }

# Implement image preprocessing
from PIL import Image, ImageEnhance

def enhance_image(image_path):
    img = Image.open(image_path)

    # Enhance brightness
    enhancer = ImageEnhance.Brightness(img)
    img = enhancer.enhance(1.2)

    # Enhance contrast
    enhancer = ImageEnhance.Contrast(img)
    img = enhancer.enhance(1.3)

    return img
```

Issue 2: Slow Recognition (Approach B)

Symptoms: Recognition takes >3 seconds

Causes:

- CPU bottleneck
- Large database of employees
- Unoptimized model

Solutions:

python

```python
# 1. Use faster model
DEEPFACE_MODEL = "OpenFace"  # Faster than Facenet

# 2. Implement caching
from functools import lru_cache

@lru_cache(maxsize=1000)
def get_employee_embeddings():
   # Cache embeddings in memory
   return load_all_embeddings()

# 3. Use GPU acceleration
# Install tensorflow-gpu
# Embeddings generation 3-5x faster

# 4. Optimize database queries
# Add indexes on frequently queried fields
CREATE INDEX idx_employees_active_face ON employees(is_active)
WHERE face_data IS NOT NULL;
```

Issue 3: High API Costs (Approach A)

Symptoms: Monthly costs exceed budget

Causes:

- Too many API calls
- Inefficient implementation
- Not using caching

Solutions:

python

```python
# 1. Implement result caching
from functools import lru_cache
import hashlib

def get_image_hash(image_data):
    return hashlib.md5(image_data).hexdigest()

# Cache recognition results for 5 minutes
recognition_cache = {}

def recognize_with_cache(image_data):
    image_hash = get_image_hash(image_data)

    if image_hash in recognition_cache:
        cached_time = recognition_cache[image_hash]['timestamp']
        if (datetime.now() - cached_time).seconds < 300:  # 5 minutes
            return recognition_cache[image_hash]['result']

    # Call API
    result = face_service.identify(image_data)

    # Cache result
    recognition_cache[image_hash] = {
        'result': result,
        'timestamp': datetime.now()
    }

    return result

# 2. Switch to Approach B for cost savings
# See migration guide in Section 15

# 3. Use local face detection before API call
# Only send to API if face is detected locally
import cv2

def has_face_locally(image_path):
    face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
    img = cv2.imread(image_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray, 1.1, 4)
    return len(faces) > 0

# Only call expensive API if local detection succeeds
```

```
if has_face_locally(image_path):
    result = face_api.detect(image_path)
```

Issue 4: Incorrect Recognition

Symptoms: Wrong employee recognized

Causes:

- Similar-looking employees
- Low-quality photos
- Single photo per employee
- Threshold too low

Solutions:

python

```
# 1. Add multiple photos per employee
# Store 3-5 photos from different angles

# 2. Increase similarity threshold
FACE_SIMILARITY_THRESHOLD = 0.7  # Up from 0.6

# 3. Implement verification step
if confidence < 0.85:
    return {
        "status": "verification_required",
        "employee": employee_data,
        "message": "Please confirm: Is this you?",
        "confidence": confidence
    }

# 4. Manual verification option
# Add "Not me" button in app
```

Issue 5: Voice Recognition Failures

Symptoms: STT returns incorrect text or nothing

Causes:

- Background noise
- Low audio quality
- Wrong language selected

- Unclear speech

Solutions:

python

```python
# 1. Add noise reduction
import noisereduce as nr
import librosa

def clean_audio(audio_file):
    data, rate = librosa.load(audio_file)
    reduced_noise = nr.reduce_noise(y=data, sr=rate)
    return reduced_noise

# 2. Improve error messages
if not transcribed_text:
    return {
        "status": "error",
        "message": "Could not understand. Please speak clearly and try again.",
        "tips": [
            "Move to quieter location",
            "Speak closer to microphone",
            "Speak slowly and clearly"
        ]
    }

# 3. Add fallback to text input
# Always show text input option in UI
```

---

22. Success Metrics & KPIs

22.1 Technical Metrics

Metric	Target	Measurement
Face Recognition Accuracy	>95%	(Correct recognitions / Total recognitions) × 100
API Response Time	<2 seconds	Average time from scan to result
System Uptime	>99.5%	(Total time - Downtime) / Total time × 100
False Positive Rate	<2%	(Wrong recognitions / Total recognitions) × 100
Registration Completion Rate	>90%	(Completed registrations / Started registrations) × 100

22.2 Business Metrics

Metric	Target	Measurement
Daily Active Users	80% of employees	Unique employees using system daily
Average Greetings per Day	1.5 per employee	Total greetings / Total employees
Employee Satisfaction	>4/5	Survey rating
Conversation Engagement	>30%	Employees using chat feature
Time Saved	5 min/employee/day	vs manual attendance

22.3 Cost Metrics

Metric	Target	Measurement
Cost per Recognition (Approach A)	<$0.002	Monthly API cost / Total recognitions
Cost per Employee (Approach A)	<$0.30/month	Total monthly cost / Total employees
Cost per Recognition (Approach B)	$0	No API costs
Total Monthly Cost	<$50	All hosting + API costs

---

23. Future Enhancements (Post-MVP)

Phase 2 Features

Advanced Recognition:

- Multi-face detection (greet multiple employees simultaneously)
- Emotion detection (adjust greeting based on mood)
- Age and gender estimation
- Mask detection (COVID-19 safety)

Enhanced Interactions:

- Natural language understanding (more complex conversations)
- Personality customization (formal vs casual greetings)
- Voice cloning for personalized voices
- Multi-lingual switching mid-conversation

Admin Features:

- Web-based admin dashboard
- Analytics and insights
- Bulk employee import/export

- Custom greeting templates
- Role-based access control

Integration:

- HR system integration (auto-sync employee data)
- Calendar integration (meeting reminders)
- Slack/Teams notifications
- Time tracking integration
- Payroll system integration

Mobile Enhancements:

- Offline mode with local storage
- Push notifications for special events
- AR greeting experience
- Widget for quick access
- Apple Watch / WearOS support

Phase 3 Features

AI/ML Enhancements:

- Continuous learning (improve accuracy over time)
- Anomaly detection (unusual behavior alerts)
- Predictive analytics (attendance patterns)
- Sentiment analysis (employee mood tracking)

Gamification:

- Points for daily check-ins
- Leaderboards
- Achievement badges
- Team challenges

Advanced Analytics:

- Attendance trends
- Department-wise insights
- Peak usage times
- Recognition accuracy by demographics

24. Training & Support

24.1 User Training

For Employees:

1. **Quick Start Guide** (2 minutes)
   - Download app
   - First-time registration
   - Daily usage
2. **Video Tutorial** (5 minutes)
   - Face scanning best practices
   - Using voice features
   - Troubleshooting common issues
3. **FAQ Document**
   - Common questions
   - Privacy concerns
   - Technical issues

24.2 Admin Training

For System Administrators:

1. **Setup Guide** (30 minutes)
   - Installation
   - Configuration
   - Testing
2. **Management Guide** (20 minutes)
   - Adding/removing employees
   - Viewing analytics
   - Cost monitoring
   - Switching between approaches
3. **Troubleshooting Guide** (15 minutes)
   - Common issues
   - Log analysis
   - Performance optimization

24.3 Support Structure

Tier 1: Self-Service

- FAQ documentation
- Video tutorials
- In-app help

Tier 2: Email Support

- Response time: 24 hours
- tech-support@yourcompany.com

Tier 3: Phone Support

- Critical issues only
- Business hours: 9 AM - 6 PM
- Emergency hotline for system downtime

---

25. Conclusion & Recommendations

25.1 Summary

This SOW presents a comprehensive solution for building a voice-based greeting agent with dynamic facial recognition capabilities. We've outlined two distinct approaches:

Approach A (Cloud API-Based):

- ✅ Best for: Quick deployment, high accuracy requirements
- ✅ Pros: Easy setup, managed infrastructure, 99.5% accuracy
- ⚠️ Cons: Recurring costs, internet dependency

Approach B (Self-Hosted):

- ✅ Best for: Cost-conscious, privacy-focused, large-scale
- ✅ Pros: Zero API costs, complete control, offline capability
- ⚠️ Cons: Higher setup effort, requires technical expertise

25.2 Final Recommendations

For Most Organizations: Start with **Approach A** for first 3 months:

- Validate business requirements quickly
- Stay within free tiers ($0 cost)

- Gather user feedback
- Prove ROI

Then evaluate:

- If monthly costs < $30 → Continue with Approach A
- If monthly costs > $50 → Migrate to Approach B
- If privacy is critical → Migrate to Approach B

For Budget-Conscious Organizations: Go directly with **Approach B**:

- Zero ongoing costs
- Complete data control
- Scalable to any size
- One-time setup effort

25.3 Expected Outcomes

Week 1-2:

- System deployed and tested
- 10-20 employees registered
- Initial feedback collected

Month 1:

- 80%+ employee adoption
- Recognition accuracy >95%
- Average response time <2 seconds
- Cost within budget

Month 3:

- 90%+ daily active usage
- High employee satisfaction
- Proven time savings
- Decision point: continue Approach A or migrate to B

Month 6:

- Optimized system performance
- Advanced features implemented
- Integration with other systems

- Full ROI achieved

25.4 Success Factors

1. **Executive Sponsorship** - Management buy-in and support
2. **Clear Communication** - Transparency about privacy and data usage
3. **Gradual Rollout** - Start with small groups, expand gradually
4. **Continuous Improvement** - Regular updates based on feedback
5. **Technical Excellence** - Proper testing and monitoring

---

26. Appendices

Appendix A: API Pricing Comparison Table

API Provider	Free Tier	Pay-as-you-go	Monthly Plans
Face++	10,000 calls/month	$0.0005/call	Custom pricing available
Clarifai	5,000 ops/month	$1.20/1000 ops	$30/month (30K ops)
Google TTS	1M chars/month	$4/1M chars	N/A
Google STT	60 min/month	$0.006/15 sec	N/A
Amazon Polly	5M chars/month (12mo)	$4/1M chars	N/A

Appendix B: Hardware Requirements Comparison

Scale	Approach A Requirements	Approach B Requirements
Dev (10 employees)	2 CPU, 4GB RAM	4 CPU, 8GB RAM
Small (100 employees)	2 CPU, 4GB RAM	4-6 CPU, 8-16GB RAM
Medium (500 employees)	4 CPU, 8GB RAM	8 CPU, 16-32GB RAM
Large (1000+ employees)	4-6 CPU, 8GB RAM	12+ CPU, 32-64GB RAM, GPU optional

Appendix C: Sample Configuration Files

backend/.env (Approach A):

```bash
```

```
DATABASE_URL=postgresql://user:pass@localhost:5432/greeting_db
RECOGNITION_METHOD=api
API_PROVIDER=facepp
FACEPP_API_KEY=your_key_here
FACEPP_API_SECRET=your_secret_here
TTS_PROVIDER=google
STT_PROVIDER=google
GOOGLE_APPLICATION_CREDENTIALS=./credentials.json
```

backend/.env (Approach B):

```bash
DATABASE_URL=postgresql://user:pass@localhost:5432/greeting_db
RECOGNITION_METHOD=local
DEEPFACE_MODEL=Facenet
FACE_SIMILARITY_THRESHOLD=0.6
TTS_PROVIDER=pyttsx3
STT_PROVIDER=vosk
VOSK_MODEL_PATH=./models/vosk-model-en
```

END OF DOCUMENT

Document Information:

- **Total Pages:** Comprehensive SOW
- **Version:** 3.0 (Dual Approach)
- **Date:** September 29, 2025
- **Prepared For:** MD Review
- **Prepared By:** Development Team

Approval Signatures:

Managing Director

Technical Lead

Project Manager

Next Steps:

1. Review and approve this SOW
2. Choose initial approach (A or B)
3. Allocate resources and budget
4. Schedule kickoff meeting
5. Begin Phase 1 development