In [ ]:
```
pip install SpeechRecognition openai pydub google-cloud-speech pyaudio
```

In [ ]:
```
# Voice-Controlled CRM Commands System
# Supports both Google Speech-to-Text and OpenAI Whisper

import speech_recognition as sr
import openai
import json
import re
import logging
from typing import Dict, List, Optional, Tuple
from dataclasses import dataclass
from datetime import datetime
import pyaudio
import wave
import io

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@dataclass
class CRMCommand:
    """Data class to represent a parsed CRM command"""
    command_type: str
    entities: Dict
    confidence: float
    raw_text: str
    api_call: str
    payload: Dict

class VoiceToText:
    """Handles voice-to-text conversion using multiple engines"""

    def __init__(self, google_credentials_path: Optional[str] = None, openai_api_ke
        self.recognizer = sr.Recognizer()
        self.microphone = sr.Microphone()
        self.google_credentials = google_credentials_path
        self.openai_api_key = openai_api_key

        if openai_api_key:
            openai.api_key = openai_api_key

        # Calibrate microphone for ambient noise
        self._calibrate_microphone()

    def _calibrate_microphone(self):
        """Calibrate microphone for ambient noise"""
        try:
            with self.microphone as source:
                logger.info("Calibrating microphone for ambient noise...")
                self.recognizer.adjust_for_ambient_noise(source, duration=1)
                logger.info("Microphone calibrated!")
        except Exception as e:
```

```
            logger.error(f"Microphone calibration failed: {e}")

    def listen_for_speech(self, timeout: int = 5) -> Optional[sr.AudioData]:
        """Listen for speech input from microphone"""
        try:
            logger.info("Listening for speech... Speak now!")
            with self.microphone as source:
                # Listen for audio with timeout
                audio = self.recognizer.listen(source, timeout=timeout, phrase_time
                logger.info("Audio captured successfully!")
                return audio
        except sr.WaitTimeoutError:
            logger.warning("No speech detected within timeout period")
            return None
        except Exception as e:
            logger.error(f"Error capturing audio: {e}")
            return None

    def convert_with_google(self, audio_data: sr.AudioData) -> Optional[str]:
        """Convert speech to text using Google Speech-to-Text"""
        try:
            if self.google_credentials:
                # Use Google Cloud Speech-to-Text with credentials
                text = self.recognizer.recognize_google_cloud(
                    audio_data,
                    credentials_json=self.google_credentials,
                    language='en-US'
                )
            else:
                # Use free Google Speech Recognition
                text = self.recognizer.recognize_google(audio_data, language='en-US

            logger.info(f"Google STT Result: {text}")
            return text
        except sr.UnknownValueError:
            logger.error("Google Speech Recognition could not understand audio")
            return None
        except sr.RequestError as e:
            logger.error(f"Google Speech Recognition error: {e}")
            return None

    def convert_with_whisper(self, audio_data: sr.AudioData) -> Optional[str]:
        """Convert speech to text using OpenAI Whisper"""
        try:
            if not self.openai_api_key:
                logger.error("OpenAI API key not provided for Whisper")
                return None

            # Convert audio data to WAV format
            wav_data = io.BytesIO()
            wav_data.write(audio_data.get_wav_data())
            wav_data.seek(0)

            # Use OpenAI Whisper API
            transcript = openai.Audio.transcribe(
                model="whisper-1",
```

```
                file=wav_data,
                response_format="text"
            )

            logger.info(f"Whisper STT Result: {transcript}")
            return transcript.strip()
        except Exception as e:
            logger.error(f"Whisper Speech Recognition error: {e}")
            return None

class CRMNLPProcessor:
    """Natural Language Processing for CRM commands"""

    def __init__(self):
        self.command_patterns = {
            'CREATE_LEAD': {
                'patterns': [
                    r'add.*lead.*from\s+(.+)',
                    r'create.*lead.*for\s+(.+)',
                    r'new lead.*from\s+(.+)',
                    r'add.*new.*lead.*(.+)'
                ],
                'extract_entities': self._extract_lead_entities
            },
            'CREATE_CONTACT': {
                'patterns': [
                    r'add.*contact.*(.+)',
                    r'create.*contact.*for\s+(.+)',
                    r'new contact.*(.+)',
                    r'add.*person.*(.+)'
                ],
                'extract_entities': self._extract_contact_entities
            },
            'SCHEDULE_MEETING': {
                'patterns': [
                    r'schedule.*meeting.*with\s+(.+?)\s+(?:at|on|for)\s+(.+)',
                    r'book.*meeting.*with\s+(.+?)\s+(?:at|on|for)\s+(.+)',
                    r'set.*meeting.*(.+)',
                    r'arrange.*meeting.*(.+)'
                ],
                'extract_entities': self._extract_meeting_entities
            },
            'UPDATE_DEAL': {
                'patterns': [
                    r'update.*deal.*to\s+(.+)',
                    r'change.*deal.*status.*to\s+(.+)',
                    r'mark.*deal.*as\s+(.+)',
                    r'set.*deal.*(.+)'
                ],
                'extract_entities': self._extract_deal_entities
            },
            'SEARCH': {
                'patterns': [
                    r'search.*for\s+(.+)',
                    r'find.*(.+)',
                    r'show.*me.*(.+)',
```

```
                r'list.*(.+)',
                r'get.*(.+)'
            ],
            'extract_entities': self._extract_search_entities
        },
        'DELETE': {
            'patterns': [
                r'delete.*contact.*(.+)',
                r'remove.*(.+)',
                r'delete.*(.+)',
                r'eliminate.*(.+)'
            ],
            'extract_entities': self._extract_delete_entities
        }
    }

def parse_command(self, text: str) -> CRMCommand:
    """Parse natural language text into structured CRM command"""
    text_lower = text.lower().strip()

    for command_type, config in self.command_patterns.items():
        for pattern in config['patterns']:
            match = re.search(pattern, text_lower, re.IGNORECASE)
            if match:
                entities = config['extract_entities'](match, text)
                api_call, payload = self._generate_api_call(command_type, entit

                return CRMCommand(
                    command_type=command_type,
                    entities=entities,
                    confidence=0.9,
                    raw_text=text,
                    api_call=api_call,
                    payload=payload
                )

    # Unknown command
    return CRMCommand(
        command_type='UNKNOWN',
        entities={'original_text': text},
        confidence=0.1,
        raw_text=text,
        api_call='',
        payload={}
    )

def _extract_lead_entities(self, match, original_text: str) -> Dict:
    """Extract entities for lead creation"""
    company = match.group(1).strip()
    return {
        'company': company,
        'source': 'voice_input',
        'created_at': datetime.now().isoformat()
    }

def _extract_contact_entities(self, match, original_text: str) -> Dict:
```

```
        """Extract entities for contact creation"""
        name_info = match.group(1).strip()
        # Try to extract company if mentioned
        company_match = re.search(r'at\s+(.+)', name_info)
        if company_match:
            name = name_info.replace(company_match.group(0), '').strip()
            company = company_match.group(1).strip()
        else:
            name = name_info
            company = None

        return {
            'name': name,
            'company': company,
            'source': 'voice_input',
            'created_at': datetime.now().isoformat()
        }

    def _extract_meeting_entities(self, match, original_text: str) -> Dict:
        """Extract entities for meeting scheduling"""
        if len(match.groups()) >= 2:
            person = match.group(1).strip()
            time_info = match.group(2).strip()
        else:
            person = match.group(1).strip()
            time_info = "not specified"

        return {
            'attendee': person,
            'datetime': time_info,
            'created_by': 'voice_input',
            'created_at': datetime.now().isoformat()
        }

    def _extract_deal_entities(self, match, original_text: str) -> Dict:
        """Extract entities for deal updates"""
        status = match.group(1).strip()
        return {
            'status': status,
            'updated_at': datetime.now().isoformat(),
            'updated_by': 'voice_input'
        }

    def _extract_search_entities(self, match, original_text: str) -> Dict:
        """Extract entities for search queries"""
        query = match.group(1).strip()
        return {
            'query': query,
            'search_type': 'general',
            'timestamp': datetime.now().isoformat()
        }

    def _extract_delete_entities(self, match, original_text: str) -> Dict:
        """Extract entities for delete operations"""
        target = match.group(1).strip()
        return {
```

```
            'target': target,
            'operation': 'delete',
            'timestamp': datetime.now().isoformat()
        }

    def _generate_api_call(self, command_type: str, entities: Dict) -> Tuple[str, D
        """Generate API call and payload for the command"""
        api_mappings = {
            'CREATE_LEAD': ('POST /api/leads', {
                'company': entities.get('company'),
                'source': entities.get('source'),
                'status': 'new'
            }),
            'CREATE_CONTACT': ('POST /api/contacts', {
                'name': entities.get('name'),
                'company': entities.get('company'),
                'source': entities.get('source')
            }),
            'SCHEDULE_MEETING': ('POST /api/meetings', {
                'attendee': entities.get('attendee'),
                'datetime': entities.get('datetime'),
                'type': 'voice_scheduled'
            }),
            'UPDATE_DEAL': ('PUT /api/deals/{deal_id}', {
                'status': entities.get('status'),
                'updated_by': 'voice_system'
            }),
            'SEARCH': ('GET /api/search', {
                'q': entities.get('query'),
                'type': 'voice_search'
            }),
            'DELETE': ('DELETE /api/contacts/{contact_id}', {
                'target': entities.get('target'),
                'confirmed': False
            })
        }

        return api_mappings.get(command_type, ('', {}))

class VoiceCRMSystem:
    """Main Voice-Controlled CRM System"""

    def __init__(self, google_credentials: Optional[str] = None, openai_api_key: Op
        self.voice_to_text = VoiceToText(google_credentials, openai_api_key)
        self.nlp_processor = CRMNLPProcessor()
        self.command_history = []

    def process_voice_command(self, use_whisper: bool = False) -> Optional[CRMComma
        """Complete pipeline: Voice -> Text -> NLP -> CRM Command"""

        # Step 1: Capture audio
        print("🎤 Voice CRM System Ready!")
        print("Say your command (e.g., 'Add a new lead from ABC Corp')...")

        audio_data = self.voice_to_text.listen_for_speech()
        if not audio_data:
```

```
            print("❌ No audio captured")
            return None

        # Step 2: Convert to text
        if use_whisper:
            print("🔄 Converting speech to text using Whisper...")
            text = self.voice_to_text.convert_with_whisper(audio_data)
        else:
            print("🔄 Converting speech to text using Google...")
            text = self.voice_to_text.convert_with_google(audio_data)

        if not text:
            print("❌ Could not transcribe audio")
            return None

        print(f"📝 Transcribed Text: '{text}'")

        # Step 3: Process with NLP
        print("🧠 Processing command with NLP...")
        command = self.nlp_processor.parse_command(text)

        # Step 4: Display results
        self._display_command_result(command)

        # Step 5: Store in history
        self.command_history.append(command)

        return command

    def _display_command_result(self, command: CRMCommand):
        """Display the processed command result"""
        print("\n" + "="*50)
        print("🎯 COMMAND ANALYSIS RESULT")
        print("="*50)
        print(f"Command Type: {command.command_type}")
        print(f"Confidence: {command.confidence:.1%}")
        print(f"Raw Text: '{command.raw_text}'")
        print(f"API Call: {command.api_call}")
        print(f"Payload: {json.dumps(command.payload, indent=2)}")
        print(f"Entities: {json.dumps(command.entities, indent=2)}")
        print("="*50 + "\n")

    def run_interactive_mode(self):
        """Run the system in interactive mode"""
        print("🚀 Voice-Controlled CRM System Started!")
        print("Commands you can try:")
        print("- 'Add a new lead from ABC Corporation'")
        print("- 'Create contact for John Smith at Microsoft'")
        print("- 'Schedule meeting with Sarah tomorrow at 2 PM'")
        print("- 'Update deal status to closed won'")
        print("- 'Search for leads from Google'")
        print("- 'Delete contact named Mike Wilson'")
        print("\nPress Ctrl+C to exit\n")

        try:
            while True:
```

```
                print("\n🎤 Ready for your voice command...")
                choice = input("Press Enter to start listening, 'w' for Whisper, or

                if choice == 'q':
                    break
                elif choice == 'w':
                    self.process_voice_command(use_whisper=True)
                else:
                    self.process_voice_command(use_whisper=False)

            except KeyboardInterrupt:
                print("\n👋 Voice CRM System stopped!")

    def get_command_history(self) -> List[CRMCommand]:
        """Get command history"""
        return self.command_history

# Example usage and testing
def main():
    """Main function to run the Voice CRM System"""

    # Initialize the system
    # Add your API keys/credentials here:
    GOOGLE_CREDENTIALS_PATH = None  # Path to Google Cloud credentials JSON
    OPENAI_API_KEY = None           # Your OpenAI API key

    system = VoiceCRMSystem(
        google_credentials=GOOGLE_CREDENTIALS_PATH,
        openai_api_key=OPENAI_API_KEY
    )

    # Run in interactive mode
    system.run_interactive_mode()

if __name__ == "__main__":
    main()
```