Customization Guide - Automated Phone Answering System

Table of Contents

Voice Response Customization

1. Response Templates

Edit Response Templates in `app.py`

```
# Customize greeting messages
GREETING_MESSAGES = [
    "Hello! Thank you for calling [YOUR BUSINESS NAME]. I'm here to help you with
court rentals, pricing, and availability. How can I assist you today?",
    "Hi! Welcome to [YOUR BUSINESS NAME]. I can help you check pricing, availability,
or make a booking. What would you like to know?",
    "Good [morning/afternoon/evening]! This is [YOUR BUSINESS NAME]. I'm your virtual
assistant. How may I help you with your sports facility needs?"
]

# Customize after-hours messages
AFTER_HOURS_MESSAGES = [
    "Thank you for calling [YOUR BUSINESS NAME]. We're currently closed. Our business
hours are [START_TIME] to [END_TIME], [DAYS]. Please call back during business
hours.",
    "Hi! You've reached [YOUR BUSINESS NAME] after hours. We're open [SCHEDULE]. For
urgent matters, please press 1 to leave a message."
]

# Customize error messages
ERROR_MESSAGES = [
    "I'm sorry, I didn't quite catch that. Could you please repeat your request?",
    "I apologize, but I'm having trouble understanding. Can you try rephrasing that?",
    "Let me make sure I understand correctly. Could you repeat what you're looking
for?"
]
```

Dynamic Response Generation

```python
def create_personalized_greeting():
    """Generate time-appropriate greeting"""
    current_hour = datetime.now().hour

    if 5 <= current_hour < 12:
        time_greeting = "Good morning"
    elif 12 <= current_hour < 17:
        time_greeting = "Good afternoon"
    else:
        time_greeting = "Good evening"

    business_name = os.getenv('BUSINESS_NAME', 'City Sports Center')

    return f"{time_greeting}! Thank you for calling {business_name}. I'm here to help
you with court rentals and bookings. How can I assist you today?"

def create_weather_aware_response():
    """Include weather information in responses"""
    # Integrate with weather API
    weather_info = get_current_weather()  # Implement this function

    if weather_info['condition'] == 'rain':
        return "Given today's rainy weather, our indoor courts are perfect for your
game! "
    elif weather_info['temperature'] > 85:
        return "It's quite hot outside today - our air-conditioned facility will keep
you comfortable! "

    return ""
```

2. Voice Characteristics Customization

TTS Voice Settings

```python
# In app.py, customize TTS parameters
def create_tts_action(text, voice_config=None):
    """Create TTS action with custom voice settings"""

    default_config = {
        'language': 'en-US',
        'style': 'neural',
        'voice_name': 'Amy',  # Options: Amy, Emma, Brian, Arthur, etc.
        'speed': 0,           # -10 to 10 (slower to faster)
        'volume': 0,          # -10 to 10 (quieter to louder)
        'pitch': 0            # -50 to 50 (lower to higher)
    }

    if voice_config:
        default_config.update(voice_config)

    return {
        'action': 'talk',
        'text': text,
        'language': default_config['language'],
        'style': default_config['style'],
        'voiceName': default_config['voice_name']
    }

# Usage examples
def create_energetic_response(text):
    """High-energy voice for promotions"""
    return create_tts_action(text, {
        'speed': 2,
        'pitch': 5,
        'volume': 2
    })

def create_calm_response(text):
    """Calm voice for complex information"""
    return create_tts_action(text, {
        'speed': -1,
        'pitch': -2,
        'volume': 0
    })
```

SSML for Advanced Control

```python
def create_ssml_response(content_type, **kwargs):
    """Generate SSML for advanced voice control"""

    ssml_templates = {
        'pricing': '''
            <speak>
                <prosody rate="medium" pitch="medium">
                    For <emphasis level="strong">{service_type}</emphasis> rentals,
                    <break time="0.3s"/>
                    our rate is <prosody rate="slow" pitch="high">
                        ${rate} per hour
                    </prosody>
                    <break time="0.5s"/>
                    {additional_info}
                </prosody>
            </speak>
        ''',

        'availability': '''
            <speak>
                <prosody rate="medium">
                    Let me check availability for you.
                    <break time="1s"/>
                    <prosody pitch="high" rate="fast">
                        Great news!
                    </prosody>
                    <break time="0.3s"/>
                    {availability_info}
                </prosody>
            </speak>
        ''',

        'confirmation': '''
            <speak>
                <prosody rate="medium" pitch="medium">
                    Perfect! <break time="0.5s"/>
                    <emphasis level="strong">Confirmed:</emphasis>
                    <break time="0.3s"/>
                    {booking_details}
                    <break time="0.5s"/>
                    <prosody rate="slow">
                        Your confirmation number is
                        <say-as interpret-as="spell-out">{confirmation_number}</say-
as>
                    </prosody>
                </prosody>
            </speak>
        '''
    }

    return ssml_templates[content_type].format(**kwargs)
```

3. Interactive Elements Customization

Custom Menu Systems

```python
def create_main_menu_ncco():
    """Create customizable main menu"""

    menu_options = {
        '1': 'Check pricing information',
        '2': 'Check availability and make booking',
        '3': 'Speak with a staff member',
        '4': 'Hear facility information',
        '9': 'Repeat this menu'
    }

    # Generate menu text
    menu_text = "Please select from the following options: "
    for key, description in menu_options.items():
        menu_text += f"Press {key} for {description}. "

    return [
        {
            'action': 'talk',
            'text': menu_text,
            'voiceName': 'Amy'
        },
        {
            'action': 'input',
            'eventUrl': f"{BASE_URL}/webhooks/menu-selection",
            'timeOut': 10,
            'maxDigits': 1,
            'submitOnHash': False
        }
    ]
```

Dynamic Response Flow

```python
class ResponseFlowManager:
    """Manage dynamic conversation flows"""

    def __init__(self):
        self.flows = {
            'pricing_inquiry': self._pricing_flow,
            'booking_flow': self._booking_flow,
            'availability_check': self._availability_flow
        }

    def _pricing_flow(self, session, step):
        """Multi-step pricing inquiry flow"""

        if step == 'initial':
            return self._ask_service_type(session)
        elif step == 'service_selected':
            return self._ask_time_period(session)
        elif step == 'time_selected':
            return self._provide_pricing(session)
        elif step == 'pricing_provided':
            return self._ask_next_action(session)

    def _ask_service_type(self, session):
        return [
            {
                'action': 'talk',
                'text': 'What type of rental are you interested in? Say "basketball
court", "party package", or "hourly rental".',
                'voiceName': 'Amy'
            },
            {
                'action': 'input',
                'eventUrl': f"{BASE_URL}/webhooks/service-type",
                'speech': {
                    'endOnSilence': 3,
                    'language': 'en-US',
                    'maxDuration': 10
                }
            }
        ]
```

Pricing Logic Customization

1. Custom Pricing Models

Create Custom Pricing Rules

Edit `pricing.py` to implement your business model:

```python
class CustomPricingEngine(PricingEngine):
    """Extended pricing engine with custom rules"""

    def __init__(self):
        super().__init__()
        self.custom_rules = self._load_custom_rules()

    def _load_custom_rules(self):
        """Load business-specific pricing rules"""
        return {
            'group_discounts': {
                'members': 0.15,        # 15% discount for members
                'students': 0.10,       # 10% discount for students
                'seniors': 0.20,        # 20% discount for seniors (65+)
                'bulk_booking': {       # Bulk booking discounts
                    '5_hours': 0.05,
                    '10_hours': 0.10,
                    '20_hours': 0.15
                }
            },
            'loyalty_program': {
                'bronze': 0.05,         # 5% after 10 bookings
                'silver': 0.10,         # 10% after 25 bookings
                'gold': 0.15            # 15% after 50 bookings
            },
            'seasonal_events': {
                'summer_camp': {
                    'dates': ['2025-06-15', '2025-08-15'],
                    'discount': 0.25
                },
                'holiday_special': {
                    'dates': ['2025-12-20', '2025-01-05'],
                    'premium': 0.10     # 10% premium during holidays
                }
            }
        }

    def calculate_custom_rate(self, base_rate, customer_info, booking_info):
        """Calculate rate with custom rules"""

        final_rate = base_rate
        applied_discounts = []

        # Member discounts
        if customer_info.get('membership_type'):
            discount = self.custom_rules['group_discounts'].get(
                customer_info['membership_type'], 0
            )
            final_rate *= (1 - discount)
            applied_discounts.append(f"{customer_info['membership_type']} discount")

        # Bulk booking discounts
        duration = booking_info.get('duration', 1)
        if duration >= 20:
            final_rate *= 0.85
            applied_discounts.append("bulk booking (20+ hours)")
        elif duration >= 10:
            final_rate *= 0.90
            applied_discounts.append("bulk booking (10+ hours)")
        elif duration >= 5:
            final_rate *= 0.95
            applied_discounts.append("bulk booking (5+ hours)")
        # Loyalty program
        booking_history = customer_info.get('total_bookings', 0)
        if booking_history >= 50:
            final_rate *= 0.85
            applied_discounts.append("gold loyalty member")
        elif booking_history >= 25:
            final_rate *= 0.90
            applied_discounts.append("silver loyalty member")
        elif booking_history >= 10:
            final_rate *= 0.95
            applied_discounts.append("bronze loyalty member")

        return {
            'rate': round(final_rate, 2),
            'base_rate': base_rate,
            'applied_discounts': applied_discounts,
            'savings': round(base_rate - final_rate, 2)
        }
```

Dynamic Pricing Based on Demand

```
def calculate_demand_based_pricing(self, base_rate, date_time, duration):
    """Adjust pricing based on demand patterns"""

    # Get historical booking data
    demand_level = self._analyze_demand(date_time, duration)

    pricing_multipliers = {
        'very_low': 0.80,    # 20% discount during low demand
        'low': 0.90,         # 10% discount
        'normal': 1.00,      # Base rate
        'high': 1.15,        # 15% premium
        'very_high': 1.30    # 30% premium
    }

    multiplier = pricing_multipliers.get(demand_level, 1.00)
    adjusted_rate = base_rate * multiplier

    return {
        'rate': adjusted_rate,
        'demand_level': demand_level,
        'multiplier': multiplier,
        'explanation': self._get_demand_explanation(demand_level)
    }

def _analyze_demand(self, date_time, duration):
    """Analyze booking patterns to determine demand"""

    # Check historical bookings for similar time slots
    similar_bookings = self._get_historical_bookings(date_time, duration)

    # Calculate demand score
    avg_bookings = len(similar_bookings) / 52  # Average per week

    if avg_bookings >= 4:
        return 'very_high'
    elif avg_bookings >= 3:
        return 'high'
    elif avg_bookings >= 1:
        return 'normal'
    elif avg_bookings >= 0.5:
        return 'low'
    else:
        return 'very_low'
```

2. Package and Promotion System

Custom Package Definitions

```python
# In pricing.py
CUSTOM_PACKAGES = {
    'birthday_party_basic': {
        'name': 'Basic Birthday Package',
        'duration': 2,
        'max_guests': 15,
        'included': [
            'Court rental (2 hours)',
            'Basic decorations',
            'Party host assistance'
        ],
        'base_price': 120.00,
        'additional_guest_fee': 5.00,
        'max_additional_guests': 10
    },

    'birthday_party_deluxe': {
        'name': 'Deluxe Birthday Package',
        'duration': 3,
        'max_guests': 25,
        'included': [
            'Court rental (3 hours)',
            'Premium decorations',
            'Party host assistance',
            'Refreshment setup',
            'Cleanup service'
        ],
        'base_price': 200.00,
        'additional_guest_fee': 7.00,
        'max_additional_guests': 15
    },

    'team_training': {
        'name': 'Team Training Package',
        'duration': 10,  # 10 hours total
        'sessions': 5,   # Spread across 5 sessions
        'max_players': 12,
        'included': [
            'Court rental (10 hours)',
            'Equipment usage',
            'Optional coach referral'
        ],
        'base_price': 300.00,
        'discount_vs_hourly': 0.25  # 25% off individual hourly rate
    }
}

def calculate_package_pricing(self, package_type, customizations):
    """Calculate pricing for custom packages"""

    if package_type not in CUSTOM_PACKAGES:
        raise ValueError(f"Unknown package type: {package_type}")

    package = CUSTOM_PACKAGES[package_type]
    total_price = package['base_price']

    # Additional guests
    extra_guests = max(0, customizations.get('guest_count', 0) - package['max_guests']
)
    if extra_guests > 0:
        if extra_guests <= package.get('max_additional_guests', 0):
            total_price += extra_guests * package['additional_guest_fee']
        else:
            raise ValueError("Too many guests for this package")

    # Add-ons
    addons = customizations.get('addons', [])
    addon_costs = {
        'photographer': 75.00,
        'catered_meal': 12.00,  # per person
        'extended_time': 35.00,  # per hour
        'premium_equipment': 25.00
    }

    for addon in addons:
        if addon == 'catered_meal':
            guest_count = customizations.get('guest_count', package['max_guests'])
            total_price += addon_costs[addon] * guest_count
        else:
            total_price += addon_costs.get(addon, 0)

    return {
        'package': package,
        'total_price': total_price,
        'breakdown': self._generate_price_breakdown(package, customizations,
addon_costs)
    }
```

Business Rules Configuration

1. Booking Policies Customization

Custom Booking Rules

```
# In calendar_helper.py
class CustomBookingRules:
    """Define custom business rules for bookings"""

    def __init__(self):
        self.rules = {
            'advance_booking': {
                'min_hours': 2,        # Minimum 2 hours advance notice
                'max_days': 90,        # Maximum 90 days in advance
                'peak_hours_min': 24   # 24 hours notice for peak times
            },
            'duration_limits': {
                'min_duration': 1,     # Minimum 1 hour
                'max_duration': 8,     # Maximum 8 hours per booking
                'max_daily_hours': 12  # Maximum 12 hours per day per customer
            },
            'group_size_limits': {
                'basketball': {'min': 2, 'max': 20},
                'birthday_party': {'min': 5, 'max': 30},
                'team_practice': {'min': 8, 'max': 15}
            },
            'cancellation_policy': {
                'free_cancellation_hours': 24,    # Free cancellation 24+ hours
                'partial_refund_hours': 12,       # 50% refund 12-24 hours
                'no_refund_hours': 12             # No refund <12 hours
            }
        }

    def validate_booking_request(self, booking_request):
        """Validate booking against business rules"""

        errors = []
        warnings = []

        # Check advance booking requirements
        advance_hours = self._calculate_advance_hours(booking_request['start_time'])
        min_required = self.rules['advance_booking']['min_hours']

        if advance_hours < min_required:
            errors.append(f"Bookings require {min_required} hours advance notice")

        # Check duration limits
        duration = booking_request['duration']
        if duration < self.rules['duration_limits']['min_duration']:
            errors.append(f"Minimum booking duration is {self.rules['duration_limits']
['min_duration']} hour(s)")

        if duration > self.rules['duration_limits']['max_duration']:
            errors.append(f"Maximum booking duration is {self.rules['duration_limits']
['max_duration']} hours")

        # Check group size
        service_type = booking_request.get('service_type', 'basketball')
        group_size = booking_request.get('group_size', 1)

        if service_type in self.rules['group_size_limits']:
            limits = self.rules['group_size_limits'][service_type]
            if group_size < limits['min']:
                warnings.append(f"Recommended minimum group size for
{service_type} is {limits['min']}")
            if group_size > limits['max']:
                errors.append(f"Maximum group size for {service_type} is
{limits['max']}")

        return {
            'valid': len(errors) == 0,
            'errors': errors,
            'warnings': warnings
        }
```

2. Seasonal and Holiday Rules

Custom Holiday Management

```
class HolidayManager:
    """Manage holiday schedules and pricing"""

    def __init__(self):
        self.holidays = {
            '2025-01-01': {'name': 'New Year\'s Day', 'type': 'major', 'hours': 'close
d'},
            '2025-07-04': {'name': 'Independence Day', 'type': 'major', 'hours': '10:0
0-18:00'},
            '2025-12-25': {'name': 'Christmas Day', 'type': 'major', 'hours':
'closed'},
            '2025-11-28': {'name': 'Thanksgiving', 'type': 'major', 'hours': '12:00-17
:00'},
            '2025-05-26': {'name': 'Memorial Day', 'type': 'minor', 'hours': '9:00-21:
00'},
            '2025-09-01': {'name': 'Labor Day', 'type': 'minor', 'hours':
'9:00-21:00'}
        }

        self.seasonal_adjustments = {
            'summer': {
                'months': [6, 7, 8],
                'rate_multiplier': 1.15,
                'extended_hours': {'open': 7, 'close': 22}
            },
            'winter': {
                'months': [12, 1, 2],
                'rate_multiplier': 0.90,
                'reduced_hours': {'open': 9, 'close': 20}
            }
        }

    def get_holiday_info(self, date):
        """Get holiday information for specific date"""
        date_str = date.strftime('%Y-%m-%d')
        return self.holidays.get(date_str)

    def apply_seasonal_pricing(self, base_rate, date):
        """Apply seasonal pricing adjustments"""
        month = date.month

        for season, config in self.seasonal_adjustments.items():
            if month in config['months']:
                multiplier = config.get('rate_multiplier', 1.0)
                return {
                    'rate': base_rate * multiplier,
                    'season': season,
                    'adjustment': f"{((multiplier - 1) * 100):+.0f}%"
                }

        return {'rate': base_rate, 'season': 'standard', 'adjustment': '0%'}
```

NLU Enhancement

1. Adding Custom Intents

Extend Intent Recognition

```python
# In nlu.py
class EnhancedNLU(SportsRentalNLU):
    """Enhanced NLU with custom intents"""

    def __init__(self):
        super().__init__()
        self._add_custom_intents()

    def _add_custom_intents(self):
        """Add business-specific intents"""

        # Membership inquiries
        self.intent_patterns['membership'] = [
            r'\b(member|membership|join|sign up|register)\b',
            r'\b(member.*benefits|member.*pricing|member.*discount)\b',
            r'\b(how to.*member|become.*member|membership.*fee)\b'
        ]

        # Equipment inquiries
        self.intent_patterns['equipment'] = [
            r'\b(equipment|gear|balls|nets|shoes|rentals)\b',
            r'\b(do you have|can I rent|what equipment)\b',
            r'\b(bring.*own|provide.*equipment)\b'
        ]

        # Facility information
        self.intent_patterns['facility_info'] = [
            r'\b(location|address|directions|parking|how to get)\b',
            r'\b(facilities|amenities|locker|shower|restroom)\b',
            r'\b(where.*located|how.*find|GPS coordinates)\b'
        ]

        # Policy questions
        self.intent_patterns['policy'] = [
            r'\b(policy|rule|regulation|allowed|permitted)\b',
            r'\b(cancel|refund|reschedule|change.*booking)\b',
            r'\b(food.*drink|outside.*food|what.*not.*allowed)\b'
        ]

        # Complaint/feedback
        self.intent_patterns['complaint'] = [
            r'\b(complaint|complain|problem|issue|dissatisfied)\b',
            r'\b(dirty|broken|not.*working|poor.*condition)\b',
            r'\b(manager|supervisor|speak.*someone)\b'
        ]

    def process_custom_intent(self, intent, entities, context):
        """Process custom intents with specific logic"""

        handlers = {
            'membership': self._handle_membership_inquiry,
            'equipment': self._handle_equipment_inquiry,
            'facility_info': self._handle_facility_info,
            'policy': self._handle_policy_question,
            'complaint': self._handle_complaint
        }

        handler = handlers.get(intent)
        if handler:
            return handler(entities, context)
        else:
            return self._handle_unknown_intent(entities, context)
    def _handle_membership_inquiry(self, entities, context):
        """Handle membership-related questions"""

        membership_info = {
            'types': {
                'basic': {'monthly_fee': 29.99, 'court_discount': 0.10},
                'premium': {'monthly_fee': 49.99, 'court_discount': 0.20},
                'family': {'monthly_fee': 79.99, 'court_discount': 0.15}
            },
            'benefits': [
                'Priority booking access',
                'Member-only events',
                'Equipment rental discounts',
                'Guest pass privileges'
            ]
        }

        return {
            'intent': 'membership',
            'response_type': 'detailed_info',
            'data': membership_info
        }
```

2. Advanced Entity Extraction

Custom Entity Recognition

```
class AdvancedEntityExtractor:
    """Enhanced entity extraction with custom patterns"""

    def __init__(self):
        self.entity_patterns = {
            'phone_number': r'\b(\+?1[-.\s]?)?\(?([0-9]{3})\)?[-.\s]?([0-9]{3})[-.\s]?
([0-9]{4})\b',
            'email': r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b',
            'date_complex': r'\b(next|this)\s+(monday|tuesday|wednesday|thursday|
friday|saturday|sunday|week|month)\b',
            'duration_complex': r'\b(half|quarter|\d+\.?\d*)\s*(hour|hours|hr|hrs)\b',
            'group_size': r'\b(for|party\s+of|group\s+of|\d+)\s+(\d+)\s+(people|
persons|kids|children|adults)\b',
            'budget': r'\$(\d+(?:,\d{3})*(?:\.\d{2})?)',
            'skill_level': r'\b(beginner|intermediate|advanced|expert|professional|
recreational)\b'
        }

        self.location_patterns = {
            'court_preference': r'\b(main\s+court|court\s+1|court\s+2|center\s+court|
side\s+court)\b',
            'parking': r'\b(parking|park|lot|garage|street\s+parking)\b',
            'entrance': r'\b(main\s+entrance|back\s+door|side\s+entrance)\b'
        }

    def extract_complex_entities(self, text, context=None):
        """Extract complex entities from speech input"""

        entities = {}

        # Phone number extraction
        phone_match = re.search(self.entity_patterns['phone_number'], text, re.IGNORE-
CASE)
        if phone_match:
            entities['phone_number'] = ''.join(phone_match.groups()[1:])

        # Budget extraction
        budget_match = re.search(self.entity_patterns['budget'], text)
        if budget_match:
            entities['budget'] = float(budget_match.group(1).replace(',', ''))

        # Skill level
        skill_match = re.search(self.entity_patterns['skill_level'], text, re.IGNORE-
CASE)
        if skill_match:
            entities['skill_level'] = skill_match.group(0).lower()

        # Complex date parsing
        entities.update(self._parse_complex_dates(text))

        # Group composition
        entities.update(self._parse_group_composition(text))

        return entities

    def _parse_complex_dates(self, text):
        """Parse complex date expressions"""

        date_entities = {}
        current_date = datetime.now().date()

        # Handle relative dates
        if 'next week' in text.lower():
            next_monday = current_date + timedelta(days=(7 - current_date.weekday()))
            date_entities['suggested_dates'] = [
                next_monday + timedelta(days=i) for i in range(7)
            ]

        # Handle specific day mentions
        weekday_match = re.search(r'\b(monday|tuesday|wednesday|thursday|friday|
saturday|sunday)\b', text, re.IGNORECASE)
        if weekday_match:
            weekday_name = weekday_match.group(0).lower()
            date_entities['preferred_weekday'] = weekday_name

        return date_entities
```

Calendar Integration Customization

1. Custom Event Templates

Advanced Event Creation

```python
# In calendar_helper.py
class CustomEventTemplates:
    """Custom event templates for different booking types"""

    def __init__(self):
        self.templates = {
            'basketball_rental': {
                'summary': '🏀 Basketball Court - {customer_name}',
                'description': self._get_basketball_description(),
                'colorId': '9',  # Blue
                'location': 'Main Basketball Court'
            },

            'birthday_party': {
                'summary': '🎉 Birthday Party - {customer_name}',
                'description': self._get_party_description(),
                'colorId': '5',  # Yellow
                'location': 'Party Area + Basketball Court'
            },

            'team_practice': {
                'summary': '🏆 Team Practice - {team_name}',
                'description': self._get_practice_description(),
                'colorId': '10', # Green
                'location': 'Main Basketball Court'
            },

            'tournament': {
                'summary': '🏆 Tournament - {tournament_name}',
                'description': self._get_tournament_description(),
                'colorId': '11', # Red
                'location': 'All Courts'
            }
        }

    def create_custom_event(self, event_type, booking_data):
        """Create event with custom template"""

        if event_type not in self.templates:
            event_type = 'basketball_rental'  # Default

        template = self.templates[event_type]

        event = {
            'summary': template['summary'].format(**booking_data),
            'description': template['description'].format(**booking_data),
            'start': {
                'dateTime': booking_data['start_time'].isoformat(),
                'timeZone': booking_data['timezone']
            },
            'end': {
                'dateTime': booking_data['end_time'].isoformat(),
                'timeZone': booking_data['timezone']
            },
            'location': template['location'],
            'colorId': template['colorId']
        }

        # Add custom reminders based on event type
        event['reminders'] = self._get_custom_reminders(event_type, booking_data)

        # Add attendees if provided
        if booking_data.get('attendees'):
            event['attendees'] = [
                {'email': email, 'responseStatus': 'needsAction'}
                for email in booking_data['attendees']
            ]

        return event

    def _get_custom_reminders(self, event_type, booking_data):
        """Get custom reminders based on event type"""

        reminder_configs = {
            'basketball_rental': [
                {'method': 'email', 'minutes': 24 * 60},  # 1 day
                {'method': 'popup', 'minutes': 60}        # 1 hour
            ],
            'birthday_party': [
                {'method': 'email', 'minutes': 3 * 24 * 60},  # 3 days
                {'method': 'email', 'minutes': 24 * 60},      # 1 day
                {'method': 'popup', 'minutes': 2 * 60}        # 2 hours
            ],
            'team_practice': [
                {'method': 'email', 'minutes': 24 * 60},  # 1 day
                {'method': 'popup', 'minutes': 30}        # 30 minutes
            ]
        }

        return {
            'useDefault': False,
            'overrides': reminder_configs.get(event_type, reminder_configs['basket-
ball_rental'])
        }
```

2. Multi-Calendar Management

Manage Multiple Calendars

```python
class MultiCalendarManager:
    """Manage bookings across multiple calendars"""

    def __init__(self):
        self.calendars = {
            'main_court': {
                'calendar_id': os.getenv('MAIN_COURT_CALENDAR_ID'),
                'name': 'Main Basketball Court',
                'capacity': 20,
                'hourly_rate': 35.00
            },
            'practice_court': {
                'calendar_id': os.getenv('PRACTICE_COURT_CALENDAR_ID'),
                'name': 'Practice Court',
                'capacity': 12,
                'hourly_rate': 25.00
            },
            'party_area': {
                'calendar_id': os.getenv('PARTY_AREA_CALENDAR_ID'),
                'name': 'Party Area',
                'capacity': 30,
                'hourly_rate': 45.00
            }
        }

    def find_available_court(self, start_time, duration, group_size):
        """Find best available court for requirements"""

        suitable_courts = []

        for court_key, court_info in self.calendars.items():
            # Check capacity
            if court_info['capacity'] >= group_size:
                # Check availability
                if self.is_court_available(court_info['calendar_id'], start_time, dur-
ation):
                    suitable_courts.append({
                        'key': court_key,
                        'info': court_info,
                        'efficiency': group_size / court_info['capacity']  # Utiliza-
tion efficiency
                    })

        # Sort by efficiency (prefer courts that match group size better)
        suitable_courts.sort(key=lambda x: x['efficiency'], reverse=True)

        return suitable_courts[0] if suitable_courts else None

    def create_booking_across_calendars(self, booking_data):
        """Create booking with automatic court selection"""

        optimal_court = self.find_available_court(
            booking_data['start_time'],
            booking_data['duration'],
            booking_data['group_size']
        )

        if not optimal_court:
            return {'success': False, 'error': 'No suitable courts available'}

        # Create event in optimal court calendar
        court_info = optimal_court['info']
        booking_data['calendar_id'] = court_info['calendar_id']
        booking_data['location'] = court_info['name']
        booking_data['hourly_rate'] = court_info['hourly_rate']

        return self.create_calendar_event(booking_data)
```

Escalation Logic Customization

1. Intelligent Escalation Rules

Custom Escalation Triggers

```python
# In escalation.py
class IntelligentEscalationHandler(EscalationHandler):
    """Enhanced escalation with intelligent routing"""

    def __init__(self):
        super().__init__()
        self.escalation_rules = {
            'payment_issues': {
                'triggers': ['payment', 'charge', 'refund', 'billing', 'credit card'],
                'priority': 'high',
                'route_to': 'billing_specialist',
                'max_wait_time': 30
            },
            'technical_problems': {
                'triggers': ['broken', 'not working', 'equipment', 'facility issue'],
                'priority': 'medium',
                'route_to': 'facility_manager',
                'max_wait_time': 60
            },
            'complex_bookings': {
                'triggers': ['multiple courts', 'tournament', 'corporate event'],
                'priority': 'medium',
                'route_to': 'event_coordinator',
                'max_wait_time': 120
            },
            'complaints': {
                'triggers': ['complaint', 'dissatisfied', 'manager', 'supervisor'],
                'priority': 'high',
                'route_to': 'customer_service_manager',
                'max_wait_time': 45
            }
        }

        self.staff_availability = {
            'billing_specialist': '+15551234567',
            'facility_manager': '+15551234568',
            'event_coordinator': '+15551234569',
            'customer_service_manager': '+15551234570',
            'general_staff': '+15551234571'
        }

    def determine_escalation_type(self, speech_input, context):
        """Intelligently determine escalation type"""

        speech_lower = speech_input.lower()

        for escalation_type, config in self.escalation_rules.items():
            for trigger in config['triggers']:
                if trigger in speech_lower:
                    return {
                        'type': escalation_type,
                        'priority': config['priority'],
                        'route_to': config['route_to'],
                        'max_wait_time': config['max_wait_time']
                    }

        return {
            'type': 'general_inquiry',
            'priority': 'low',
            'route_to': 'general_staff',
            'max_wait_time': 180
        }
    def create_intelligent_escalation_ncco(self, escalation_info, context):
        """Create escalation NCCO with intelligent routing"""

        # Check staff availability
        target_staff = escalation_info['route_to']
        staff_phone = self.staff_availability.get(target_staff)

        if not staff_phone:
            staff_phone = self.staff_availability['general_staff']

        # Create personalized hold message
        hold_message = self._create_personalized_hold_message(escalation_info)

        ncco = [
            {
                'action': 'talk',
                'text': hold_message,
                'voiceName': 'Amy'
            }
        ]

        # Add hold music if available
        hold_music_url = os.getenv('HOLD_MUSIC_URL')
        if hold_music_url:
            ncco.append({
                'action': 'stream',
                'streamUrl': [hold_music_url],
                'loop': 0  # Loop until answered
            })

        # Connect to staff
        ncco.append({
            'action': 'connect',
            'endpoint': [{
                'type': 'phone',
                'number': staff_phone
            }],
            'timeOut': escalation_info['max_wait_time']
        })

        return ncco
```

2. Callback Management

Advanced Callback System

```
class CallbackManager:
    """Manage callback requests and scheduling"""

    def __init__(self):
        self.callback_queue = []
        self.business_hours = self._get_business_hours()

    def schedule_callback(self, caller_info, preferred_time=None):
        """Schedule callback with intelligent timing"""

        callback_request = {
            'id': self._generate_callback_id(),
            'caller_number': caller_info['number'],
            'caller_name': caller_info.get('name', 'Unknown'),
            'request_time': datetime.now(),
            'preferred_time': preferred_time,
            'priority': self._calculate_priority(caller_info),
            'context': caller_info.get('context', {}),
            'status': 'pending'
        }

        # Determine optimal callback time
        optimal_time = self._calculate_optimal_callback_time(
            preferred_time,
            callback_request['priority']
        )

        callback_request['scheduled_time'] = optimal_time

        # Add to queue
        self.callback_queue.append(callback_request)
        self.callback_queue.sort(key=lambda x: (x['priority'], x['scheduled_time']))

        # Log callback request
        self._log_callback_request(callback_request)

        return {
            'callback_id': callback_request['id'],
            'scheduled_time': optimal_time,
            'confirmation_message': self._generate_confirmation_message(optimal_time)
        }

    def _calculate_optimal_callback_time(self, preferred_time, priority):
        """Calculate optimal callback time"""

        now = datetime.now()

        # High priority callbacks within 30 minutes
        if priority == 'high':
            return now + timedelta(minutes=30)

        # Medium priority within 2 hours
        if priority == 'medium':
            return now + timedelta(hours=2)

        # Low priority next business day if after hours
        if preferred_time:
            return preferred_time

        # Default to next available business hour
        return self._next_business_hour(now)
```

Multi-Language Support

1. Language Detection and Switching

Automatic Language Detection

```
class MultiLanguageNLU:
    """Multi-language NLU support"""

    def __init__(self):
        self.supported_languages = {
            'en': {'name': 'English', 'voice': 'Amy'},
            'es': {'name': 'Spanish', 'voice': 'Penelope'},
            'fr': {'name': 'French', 'voice': 'Celine'},
            'de': {'name': 'German', 'voice': 'Marlene'}
        }

        self.language_patterns = {
            'es': [
                r'\b(hola|buenos días|buenas tardes|gracias|por favor)\b',
                r'\b(precio|costo|disponible|reservar|cancha)\b'
            ],
            'fr': [
                r'\b(bonjour|bonsoir|merci|s\'il vous plaît)\b',
                r'\b(prix|coût|disponible|réserver|terrain)\b'
            ],
            'de': [
                r'\b(hallo|guten tag|danke|bitte)\b',
                r'\b(preis|kosten|verfügbar|reservieren|platz)\b'
            ]
        }

    def detect_language(self, speech_input):
        """Detect language from speech input"""

        speech_lower = speech_input.lower()

        for lang_code, patterns in self.language_patterns.items():
            for pattern in patterns:
                if re.search(pattern, speech_lower, re.IGNORECASE):
                    return lang_code

        return 'en'  # Default to English

    def get_localized_response(self, response_key, language, **kwargs):
        """Get response in specified language"""

        responses = {
            'greeting': {
                'en':
"Hello! Thank you for calling City Sports Center. How can I help you today?",
                'es': "¡Hola! Gracias por llamar al Centro Deportivo de la Ciudad.
¿Cómo puedo ayudarte hoy?",
                'fr': "Bonjour! Merci d'avoir appelé le Centre Sportif de la Ville.
Comment puis-je vous aider aujourd'hui?",
                'de':
"Hallo! Vielen Dank, dass Sie das Stadtsportzentrum anrufen. Wie kann ich Ihnen heute
helfen?"
            },
            'pricing_response': {
                'en': "Our court rental is ${rate} per hour. Would you like to check
availability?",
                'es': "El alquiler de nuestra cancha es $
{rate} por hora. ¿Te gustaría verificar la disponibilidad?",
                'fr': "La location de notre terrain est ${rate} par heure. Souhaitez-
vous vérifier la disponibilité?",
                'de': "Unsere Platzvermietung kostet ${rate} pro Stunde. Möchten Sie
die Verfügbarkeit prüfen?"
```

```
            }
        }

        template = responses.get(response_key, {}).get(language, re-
sponses[response_key]['en'])
        return template.format(**kwargs)
```

Advanced Integrations

1. CRM Integration

Customer Data Management

```python
class CRMIntegration:
    """Integrate with CRM systems for customer data"""

    def __init__(self):
        self.crm_config = {
            'api_url': os.getenv('CRM_API_URL'),
            'api_key': os.getenv('CRM_API_KEY'),
            'timeout': 5
        }

    def lookup_customer(self, phone_number):
        """Look up customer information"""

        try:
            response = requests.get(
                f"{self.crm_config['api_url']}/customers/search",
                params={'phone': phone_number},
                headers={'Authorization': f"Bearer {self.crm_config['api_key']}"},
                timeout=self.crm_config['timeout']
            )

            if response.status_code == 200:
                customer_data = response.json()
                return {
                    'found': True,
                    'name': customer_data.get('name'),
                    'email': customer_data.get('email'),
                    'membership_level': customer_data.get('membership_level'),
                    'total_bookings': customer_data.get('total_bookings', 0),
                    'preferred_court': customer_data.get('preferred_court'),
                    'notes': customer_data.get('notes', '')
                }

        except Exception as e:
            logging.warning(f"CRM lookup failed: {e}")

        return {'found': False}

    def create_or_update_customer(self, customer_info):
        """Create or update customer record"""

        try:
            response = requests.post(
                f"{self.crm_config['api_url']}/customers",
                json=customer_info,
                headers={'Authorization': f"Bearer {self.crm_config['api_key']}"},
                timeout=self.crm_config['timeout']
            )

            return response.status_code == 201

        except Exception as e:
            logging.error(f"CRM update failed: {e}")
            return False
```

2. Payment Processing Integration

Automated Payment Collection

```python
class PaymentIntegration:
    """Integrate payment processing for bookings"""

    def __init__(self):
        self.payment_config = {
            'stripe_key': os.getenv('STRIPE_SECRET_KEY'),
            'min_amount': 25.00,  # Minimum payment amount
            'currency': 'usd'
        }

    def create_payment_intent(self, amount, customer_info, booking_info):
        """Create Stripe payment intent"""

        try:
            intent = stripe.PaymentIntent.create(
                amount=int(amount * 100),  # Convert to cents
                currency=self.payment_config['currency'],
                description=f"Court rental - {booking_info['date']} {booking_info['tim
e']}",
                metadata={
                    'booking_id': booking_info.get('id'),
                    'customer_phone': customer_info.get('phone'),
                    'duration': str(booking_info.get('duration'))
                }
            )

            return {
                'success': True,
                'payment_intent_id': intent.id,
                'client_secret': intent.client_secret
            }

        except Exception as e:
            logging.error(f"Payment intent creation failed: {e}")
            return {'success': False, 'error': str(e)}

    def handle_payment_over_phone(self, customer_info, amount):
        """Handle payment collection over phone"""

        # Generate payment link
        payment_link = self._create_payment_link(customer_info, amount)

        # Send via SMS if possible
        if customer_info.get('phone'):
            self._send_payment_link_sms(customer_info['phone'], payment_link)

        return {
            'payment_link': payment_link,
            'instructions': "I've sent a secure payment link to your phone. You can
also complete payment when you arrive for your booking."
        }
```

Customization Complete! 🎨

Your automated phone answering system is now fully customizable. These modifications allow you to tailor every aspect of the system to your specific business needs, from voice responses to complex business logic.

Next, review the testing and troubleshooting guides to ensure your customizations work perfectly!