Testing and Troubleshooting Guide - Automated Phone Answering System

Table of Contents

Testing Overview

Testing Methodology

The automated phone answering system requires comprehensive testing across multiple layers:

- **Unit Testing**: Individual component functionality
- **Integration Testing**: Component interaction verification
- **System Testing**: Complete workflow validation
- **Performance Testing**: Load and stress testing
- **Security Testing**: Vulnerability assessment
- **User Acceptance Testing**: Real-world scenario validation

Testing Environment Setup

```
# Create testing environment
cp .env.example .env.test

# Set test-specific variables
export FLASK_ENV=testing
export TESTING=true
export VONAGE_PHONE_NUMBER=+15551234567  # Test number
```

Pre-Deployment Testing

1. Configuration Validation

Run Configuration Validator

```
# Navigate to project directory
cd /opt/auto_call_system

# Run validation script
python3 validate_config.py
```

Expected Output:

```
Validating Auto Call System Configuration...
--------------------------------------------------
✅ All required environment variables are set
✅ All required files are present
✅ Google credentials format is valid
✅ Business hours configuration is valid

🎉 All configuration checks passed!
```

2. Dependency Testing

Test Python Dependencies

```
#!/usr/bin/env python3
"""Test all Python dependencies"""

def test_dependencies():
    """Test import of all required packages"""

    try:
        import flask
        print(f"✅ Flask: {flask.__version__}")
    except ImportError as e:
        print(f"❌ Flask import failed: {e}")

    try:
        import vonage
        print(f"✅ Vonage: {vonage.__version__}")
    except ImportError as e:
        print(f"❌ Vonage import failed: {e}")

    try:
        from google.auth import service_account
        from googleapiclient.discovery import build
        print("✅ Google API client libraries")
    except ImportError as e:
        print(f"❌ Google API libraries failed: {e}")

    try:
        import pandas as pd
        print(f"✅ Pandas: {pd.__version__}")
    except ImportError as e:
        print(f"❌ Pandas import failed: {e}")

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

3. File Structure Validation

```bash
#!/bin/bash
# validate_structure.sh

echo "Validating project structure..."

required_files=(
    "app.py"
    "nlu.py"
    "calendar_helper.py"
    "pricing.py"
    "escalation.py"
    "requirements.txt"
    ".env"
    "private.key"
    "credentials.json"
)

missing_files=()

for file in "${required_files[@]}"; do
    if [[ ! -f "$file" ]]; then
        missing_files+=("$file")
    fi
done

if [[ ${#missing_files[@]} -eq 0 ]]; then
    echo "✅ All required files present"
else
    echo "❌ Missing files: ${missing_files[*]}"
    exit 1
fi

echo "✅ Project structure validation complete"
```

Component Testing

1. NLU Module Testing

Create NLU Test Suite

```python
#!/usr/bin/env python3
"""Test NLU functionality"""

import unittest
from nlu import SportsRentalNLU

class TestNLU(unittest.TestCase):

    def setUp(self):
        self.nlu = SportsRentalNLU()

    def test_pricing_intent(self):
        """Test pricing intent recognition"""
        test_cases = [
            "How much does it cost?",
            "What are your rates?",
            "Price for basketball court",
            "How expensive is it?",
            "What's the hourly rate?"
        ]

        for text in test_cases:
            result = self.nlu.process_speech_input(text, {})
            self.assertEqual(result['intent'], 'pricing', f"Failed for: {text}")

    def test_availability_intent(self):
        """Test availability intent recognition"""
        test_cases = [
            "Are you available tomorrow?",
            "Do you have any free time slots?",
            "Check availability for next week",
            "What times are open?",
            "When can I book?"
        ]

        for text in test_cases:
            result = self.nlu.process_speech_input(text, {})
            self.assertEqual(result['intent'], 'availability', f"Failed for: {text}")

    def test_booking_intent(self):
        """Test booking intent recognition"""
        test_cases = [
            "I want to book a court",
            "Can I reserve for tomorrow?",
            "I'd like to schedule a session",
            "Book me for 2 hours",
            "Make a reservation"
        ]

        for text in test_cases:
            result = self.nlu.process_speech_input(text, {})
            self.assertEqual(result['intent'], 'booking', f"Failed for: {text}")

    def test_entity_extraction(self):
        """Test entity extraction"""
        text = "I want to book the court for 2 hours tomorrow at 3 PM for 8 people"
        result = self.nlu.process_speech_input(text, {})

        entities = result.get('entities', {})
        self.assertIn('duration', entities)
        self.assertIn('date', entities)
        self.assertIn('group_size', entities)
```

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

Run NLU Tests

```
python3 -m unittest test_nlu.py -v
```

2. Calendar Integration Testing

Test Calendar Helper

```python
#!/usr/bin/env python3
"""Test Google Calendar integration"""

import unittest
from datetime import datetime, timedelta
from calendar_helper import CalendarHelper

class TestCalendarHelper(unittest.TestCase):

    def setUp(self):
        self.calendar = CalendarHelper()

    def test_calendar_connection(self):
        """Test basic calendar connection"""
        try:
            # Attempt to get calendar info
            calendar_info = self.calendar.service.calendars().get(
                calendarId=self.calendar.calendar_id
            ).execute()

            self.assertIsNotNone(calendar_info)
            self.assertIn('summary', calendar_info)
            print("✅ Calendar connection successful")

        except Exception as e:
            self.fail(f"Calendar connection failed: {e}")

    def test_availability_check(self):
        """Test availability checking"""
        # Test tomorrow at 2 PM for 2 hours
        tomorrow = datetime.now() + timedelta(days=1)
        start_time = tomorrow.replace(hour=14, minute=0, second=0, microsecond=0)

        try:
            availability = self.calendar.check_availability(start_time, 2)
            self.assertIsInstance(availability, dict)
            self.assertIn('available', availability)
            print("✅ Availability check successful")

        except Exception as e:
            self.fail(f"Availability check failed: {e}")

    def test_event_creation(self):
        """Test event creation and deletion"""
        # Create test event for next week to avoid conflicts
        next_week = datetime.now() + timedelta(days=7)
        start_time = next_week.replace(hour=10, minute=0, second=0, microsecond=0)
        end_time = start_time + timedelta(hours=1)

        event_data = {
            'summary': 'TEST EVENT - Basketball Court Rental',
            'description': 'Test booking from automated system',
            'start_time': start_time,
            'end_time': end_time,
            'customer_name': 'Test Customer',
            'phone_number': '+15551234567',
            'group_size': 5
        }

        try:
            # Create event
            result = self.calendar.create_booking(event_data)
            self.assertTrue(result['success'])
            self.assertIn('event_id', result)

            event_id = result['event_id']
            print(f"✅ Event created: {event_id}")

            # Clean up - delete test event
            delete_result = self.calendar.delete_booking(event_id)
            self.assertTrue(delete_result['success'])
            print("✅ Test event cleaned up")

        except Exception as e:
            self.fail(f"Event creation/deletion failed: {e}")

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

3. Pricing Engine Testing

Test Pricing Logic

```python
#!/usr/bin/env python3
"""Test pricing engine functionality"""

import unittest
from datetime import datetime, time
from pricing import PricingEngine

class TestPricingEngine(unittest.TestCase):

    def setUp(self):
        self.pricing = PricingEngine()

    def test_basic_pricing(self):
        """Test basic hourly pricing"""
        # Test weekday standard rate
        weekday_morning = datetime(2025, 10, 1, 10, 0)  # Wednesday 10 AM

        rate_info = self.pricing.calculate_rate(
            service_type='basketball',
            start_time=weekday_morning,
            duration=1
        )

        self.assertIsInstance(rate_info, dict)
        self.assertIn('hourly_rate', rate_info)
        self.assertGreater(rate_info['hourly_rate'], 0)
        print(f"✅ Basic pricing: ${rate_info['hourly_rate']}/hour")

    def test_peak_pricing(self):
        """Test peak hour pricing"""
        # Test weekend peak time
        weekend_peak = datetime(2025, 10, 4, 15, 0)  # Saturday 3 PM

        rate_info = self.pricing.calculate_rate(
            service_type='basketball',
            start_time=weekend_peak,
            duration=2
        )

        # Peak rates should be higher than base rate
        self.assertIn('peak_multiplier', rate_info)
        self.assertGreaterEqual(rate_info['peak_multiplier'], 1.0)
        print(f"✅ Peak pricing: ${rate_info['hourly_rate']}/hour (multiplier: {rate_i
nfo['peak_multiplier']})")

    def test_package_pricing(self):
        """Test birthday party package pricing"""
        package_info = self.pricing.get_package_pricing(
            package_type='birthday_party_basic',
            customizations={
                'guest_count': 12,
                'addons': ['photographer']
            }
        )

        self.assertIsInstance(package_info, dict)
        self.assertIn('total_price', package_info)
        self.assertGreater(package_info['total_price'], 0)
        print(f"✅ Package pricing: ${package_info['total_price']}")

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

4. Escalation Handler Testing

Test Escalation Logic

```python
#!/usr/bin/env python3
"""Test escalation handler"""

import unittest
from escalation import EscalationHandler

class TestEscalationHandler(unittest.TestCase):

    def setUp(self):
        self.escalation = EscalationHandler()

    def test_escalation_detection(self):
        """Test escalation scenario detection"""
        test_cases = [
            ("I have a problem with my payment", "payment_issue"),
            ("The equipment is broken", "technical_issue"),
            ("I want to speak to a manager", "complaint"),
            ("This is a corporate event", "complex_booking")
        ]

        for text, expected_type in test_cases:
            escalation_type = self.escalation.detect_escalation_type(text)
            self.assertEqual(escalation_type, expected_type, f"Failed for: {text}")
            print(f"✅ Detected '{expected_type}' for: {text}")

    def test_ncco_generation(self):
        """Test NCCO generation for escalation"""
        ncco = self.escalation.create_escalation_ncco("payment_issue", {})

        self.assertIsInstance(ncco, list)
        self.assertGreater(len(ncco), 0)

        # Should contain talk and connect actions
        actions = [action['action'] for action in ncco]
        self.assertIn('talk', actions)
        self.assertIn('connect', actions)
        print("✅ Escalation NCCO generation successful")

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

End-to-End Testing

1. Webhook Testing

Test Vonage Webhook Integration

```python
#!/usr/bin/env python3
"""Test webhook endpoints"""

import unittest
import json
from app import app

class TestWebhooks(unittest.TestCase):

    def setUp(self):
        self.app = app.test_client()
        self.app.testing = True

    def test_answer_webhook(self):
        """Test incoming call webhook"""
        call_data = {
            'conversation_uuid': 'test-conversation-uuid',
            'from': '+15551234567',
            'to': '+15559876543'
        }

        response = self.app.post(
            '/webhooks/answer',
            data=json.dumps(call_data),
            content_type='application/json'
        )

        self.assertEqual(response.status_code, 200)

        response_data = json.loads(response.data)
        self.assertIsInstance(response_data, list)

        # Should contain at least one action
        self.assertGreater(len(response_data), 0)
        self.assertIn('action', response_data[0])
        print("✅ Answer webhook test passed")

    def test_speech_webhook(self):
        """Test speech input webhook"""
        speech_data = {
            'conversation_uuid': 'test-conversation-uuid',
            'speech': {
                'results': [{'text': 'How much does it cost?'}]
            }
        }

        # First, establish a session
        call_data = {
            'conversation_uuid': 'test-conversation-uuid',
            'from': '+15551234567',
            'to': '+15559876543'
        }
        self.app.post(
            '/webhooks/answer',
            data=json.dumps(call_data),
            content_type='application/json'
        )

        # Then test speech processing
        response = self.app.post(
            '/webhooks/speech',
            data=json.dumps(speech_data),
            content_type='application/json'
        )

        self.assertEqual(response.status_code, 200)
        response_data = json.loads(response.data)
        self.assertIsInstance(response_data, list)
        print("✅ Speech webhook test passed")

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

2. Complete Call Flow Testing

Automated Call Flow Test

```
#!/usr/bin/env python3
"""Test complete call flows"""

import unittest
import requests
import time
from datetime import datetime, timedelta

class TestCallFlows(unittest.TestCase):

    def setUp(self):
        self.base_url = "http://localhost:5000"  # Adjust for your deployment
        self.test_conversation_uuid = f"test-{int(time.time())}"

    def test_pricing_inquiry_flow(self):
        """Test complete pricing inquiry flow"""

        # Step 1: Simulate incoming call
        call_data = {
            'conversation_uuid': self.test_conversation_uuid,
            'from': '+15551234567',
            'to': '+15559876543'
        }

        response = requests.post(
            f"{self.base_url}/webhooks/answer",
            json=call_data
        )

        self.assertEqual(response.status_code, 200)
        ncco = response.json()
        self.assertIsInstance(ncco, list)
        print("✅ Call answered successfully")

        # Step 2: Simulate speech input asking about pricing
        speech_data = {
            'conversation_uuid': self.test_conversation_uuid,
            'speech': {
                'results': [{'text': 'How much does it cost to rent the basketball
court?'}]
            }
        }

        response = requests.post(
            f"{self.base_url}/webhooks/speech",
            json=speech_data
        )

        self.assertEqual(response.status_code, 200)
        ncco = response.json()

        # Should contain pricing information in the response
        response_text = ""
        for action in ncco:
            if action.get('action') == 'talk':
                response_text += action.get('text', '')

        self.assertIn('$', response_text)  # Should contain price
        print("✅ Pricing inquiry flow completed")

    def test_availability_check_flow(self):
        """Test availability checking flow"""
        # Initialize call
        call_data = {
            'conversation_uuid': self.test_conversation_uuid + "_avail",
            'from': '+15551234567',
            'to': '+15559876543'
        }

        requests.post(f"{self.base_url}/webhooks/answer", json=call_data)

        # Ask about availability
        speech_data = {
            'conversation_uuid': self.test_conversation_uuid + "_avail",
            'speech': {
                'results': [{'text': 'Are you available tomorrow at 2 PM?'}]
            }
        }

        response = requests.post(
            f"{self.base_url}/webhooks/speech",
            json=speech_data
        )

        self.assertEqual(response.status_code, 200)
        ncco = response.json()

        # Should contain availability information
        response_text = ""
        for action in ncco:
            if action.get('action') == 'talk':
                response_text += action.get('text', '')

        self.assertTrue(
            'available' in response_text.lower() or
            'booked' in response_text.lower() or
            'unavailable' in response_text.lower()
        )
        print("✅ Availability check flow completed")

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

3. Load Testing

Concurrent Call Simulation

```python
#!/usr/bin/env python3
"""Load testing for concurrent calls"""

import asyncio
import aiohttp
import time
from concurrent.futures import ThreadPoolExecutor

class LoadTester:

    def __init__(self, base_url="http://localhost:5000"):
        self.base_url = base_url
        self.session = None

    async def simulate_call(self, call_id):
        """Simulate a complete call interaction"""

        conversation_uuid = f"load-test-{call_id}-{int(time.time())}"

        try:
            # Step 1: Answer call
            async with self.session.post(
                f"{self.base_url}/webhooks/answer",
                json={
                    'conversation_uuid': conversation_uuid,
                    'from': f'+155512{call_id:05d}',
                    'to': '+15559876543'
                }
            ) as response:
                if response.status != 200:
                    return {'call_id': call_id, 'status': 'failed', 'step': 'answer'}

            # Step 2: Speech input
            await asyncio.sleep(0.5)  # Simulate thinking time

            async with self.session.post(
                f"{self.base_url}/webhooks/speech",
                json={
                    'conversation_uuid': conversation_uuid,
                    'speech': {
                        'results': [{'text': 'How much does it cost?'}]
                    }
                }
            ) as response:
                if response.status != 200:
                    return {'call_id': call_id, 'status': 'failed', 'step': 'speech'}

            return {'call_id': call_id, 'status': 'success', 'step': 'completed'}

        except Exception as e:
            return {'call_id': call_id, 'status': 'error', 'error': str(e)}

    async def run_load_test(self, concurrent_calls=10, total_calls=100):
        """Run load test with specified parameters"""

        connector = aiohttp.TCPConnector(limit=concurrent_calls * 2)
        timeout = aiohttp.ClientTimeout(total=30)

        async with aiohttp.ClientSession(connector=connector, timeout=timeout) as ses-
sion:
            self.session = session
            print(f"Starting load test: {total_calls} calls, {concurrent_calls} con-
current")

            semaphore = asyncio.Semaphore(concurrent_calls)

            async def limited_call(call_id):
                async with semaphore:
                    return await self.simulate_call(call_id)

            start_time = time.time()

            # Run all calls
            tasks = [limited_call(i) for i in range(total_calls)]
            results = await asyncio.gather(*tasks)

            end_time = time.time()
            duration = end_time - start_time

            # Analyze results
            successful = len([r for r in results if r['status'] == 'success'])
            failed = len([r for r in results if r['status'] != 'success'])

            print(f"\nLoad Test Results:")
            print(f"Duration: {duration:.2f} seconds")
            print(f"Calls per second: {total_calls / duration:.2f}")
            print(f"Successful: {successful}/{total_calls} ({successful/total_calls*10
0:.1f}%)")
            print(f"Failed: {failed}/{total_calls} ({failed/total_calls*100:.1f}%)")

            if failed > 0:
                print("\nFailures:")
                for result in results:
                    if result['status'] != 'success':
                        print(f"  Call {result['call_id']}: {result['status']} at {res
ult.get('step', 'unknown')}")

if __name__ == "__main__":
    import sys

    concurrent_calls = int(sys.argv[1]) if len(sys.argv) > 1 else 10
    total_calls = int(sys.argv[2]) if len(sys.argv) > 2 else 100

    tester = LoadTester()
    asyncio.run(tester.run_load_test(concurrent_calls, total_calls))
```

Run Load Test

```
# Test with 10 concurrent calls, 100 total
python3 load_test.py 10 100

# Test with higher load
python3 load_test.py 25 500
```

Performance Testing

1. Response Time Testing

Measure Component Performance

```python
#!/usr/bin/env python3
"""Performance testing for individual components"""

import time
import statistics
from datetime import datetime, timedelta

from nlu import SportsRentalNLU
from calendar_helper import CalendarHelper
from pricing import PricingEngine

class PerformanceTester:

    def __init__(self):
        self.nlu = SportsRentalNLU()
        self.calendar = CalendarHelper()
        self.pricing = PricingEngine()

    def measure_execution_time(self, func, *args, **kwargs):
        """Measure function execution time"""
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        return end_time - start_time, result

    def test_nlu_performance(self, iterations=100):
        """Test NLU processing performance"""

        test_inputs = [
            "How much does it cost?",
            "Are you available tomorrow?",
            "I want to book a court for 2 hours",
            "What are your prices for birthday parties?",
            "Can I speak to a manager?"
        ]

        times = []

        for i in range(iterations):
            test_input = test_inputs[i % len(test_inputs)]
            duration, _ = self.measure_execution_time(
                self.nlu.process_speech_input,
                test_input,
                {}
            )
            times.append(duration)

        avg_time = statistics.mean(times)
        max_time = max(times)
        min_time = min(times)
        p95_time = statistics.quantiles(times, n=20)[18]  # 95th percentile

        print(f"NLU Performance ({iterations} iterations):")
        print(f"  Average: {avg_time*1000:.2f}ms")
        print(f"  Min: {min_time*1000:.2f}ms")
        print(f"  Max: {max_time*1000:.2f}ms")
        print(f"  95th percentile: {p95_time*1000:.2f}ms")

        return avg_time < 0.1  # Should be under 100ms

    def test_calendar_performance(self, iterations=20):
        """Test calendar API performance"""
        times = []
        tomorrow = datetime.now() + timedelta(days=1)

        for i in range(iterations):
            test_time = tomorrow.replace(hour=10 + (i % 8), minute=0, second=0, micro-
second=0)

            duration, _ = self.measure_execution_time(
                self.calendar.check_availability,
                test_time,
                1
            )
            times.append(duration)

        avg_time = statistics.mean(times)
        max_time = max(times)
        min_time = min(times)

        print(f"Calendar Performance ({iterations} iterations):")
        print(f"  Average: {avg_time*1000:.2f}ms")
        print(f"  Min: {min_time*1000:.2f}ms")
        print(f"  Max: {max_time*1000:.2f}ms")

        return avg_time < 2.0  # Should be under 2 seconds

    def test_pricing_performance(self, iterations=1000):
        """Test pricing calculation performance"""

        times = []
        test_time = datetime.now() + timedelta(days=1)

        for i in range(iterations):
            duration, _ = self.measure_execution_time(
                self.pricing.calculate_rate,
                'basketball',
                test_time,
                1 + (i % 5)  # 1-5 hours
            )
            times.append(duration)

        avg_time = statistics.mean(times)
        max_time = max(times)
        min_time = min(times)

        print(f"Pricing Performance ({iterations} iterations):")
        print(f"  Average: {avg_time*1000:.2f}ms")
        print(f"  Min: {min_time*1000:.2f}ms")
        print(f"  Max: {max_time*1000:.2f}ms")

        return avg_time < 0.01  # Should be under 10ms

if __name__ == "__main__":
    tester = PerformanceTester()

    print("Running Performance Tests...\n")

    nlu_ok = tester.test_nlu_performance()
    print(f"NLU Performance: {'✅ PASS' if nlu_ok else '❌ FAIL'}\n")

    calendar_ok = tester.test_calendar_performance()
    print(f"Calendar Performance: {'✅ PASS' if calendar_ok else '❌ FAIL'}\n")
    pricing_ok = tester.test_pricing_performance()
    print(f"Pricing Performance: {'✅ PASS' if pricing_ok else '❌ FAIL'}\n")

    if all([nlu_ok, calendar_ok, pricing_ok]):
        print("🎉 All performance tests passed!")
    else:
        print("⚠️ Some performance tests failed - consider optimization")
```

Common Issues and Solutions

1. Vonage API Issues

Problem: "Invalid Application ID"

Symptoms:
- Calls not being answered
- Webhook not receiving requests
- Error logs: "Application not found"

Diagnosis:

```
# Check application configuration
curl -X GET "https://api.nexmo.com/v2/applications/$VONAGE_APPLICATION_ID" \
  -H "Authorization: Bearer $JWT_TOKEN"
```

Solutions:
1. Verify application ID in `.env` file
2. Check if application was accidentally deleted
3. Ensure webhook URLs are correct and accessible
4. Verify domain name and SSL certificate

Problem: "Private Key Authentication Failed"

Symptoms:
- Authentication errors in logs
- Unable to make API calls
- JWT token generation failures

Diagnosis:

```
# Check private key file
ls -la private.key
head -n 1 private.key  # Should start with "-----BEGIN PRIVATE KEY-----"
```

Solutions:
1. Re-download private key from Vonage dashboard
2. Check file permissions (should be 600)
3. Ensure no extra whitespace or characters
4. Verify path in `VONAGE_PRIVATE_KEY_PATH`

Problem: "Webhook Timeout"

Symptoms:
- Calls connect but no response

- Webhook receiving requests but timing out
- Error: “No response from webhook”

Diagnosis:

```
# Check webhook response time
curl -w "@curl-format.txt" -X POST http://localhost:5000/webhooks/answer \
  -H "Content-Type: application/json" \
  -d '{"conversation_uuid": "test", "from": "+15551234567"}'
```

Solutions:
1. Optimize database queries
2. Implement caching for frequent operations
3. Add timeout handling in webhook handlers
4. Scale server resources

2. Google Calendar API Issues

Problem: “Calendar Not Found” or “Access Denied”

Symptoms:
- Cannot check availability
- Booking creation fails
- Error: “Calendar not accessible”

Diagnosis:

```
# Test calendar access
from google.oauth2 import service_account
from googleapiclient.discovery import build

credentials = ser-
vice_account.Credentials.from_service_account_file('credentials.json')
service = build('calendar', 'v3', credentials=credentials)

try:
    calendar = service.calendars().get(calendarId='your_calendar_id').execute()
    print(f"Calendar accessible: {calendar['summary']}")
except Exception as e:
    print(f"Calendar access error: {e}")
```

Solutions:
1. Verify service account email has been shared with calendar
2. Check calendar ID is correct
3. Ensure service account has appropriate permissions
4. Re-generate and download service account credentials

Problem: “Quota Exceeded”

Symptoms:
- Intermittent calendar operations fail
- Error: “Rate limit exceeded” or “Quota exceeded”
- Works sometimes, fails other times

Solutions:
1. Implement exponential backoff for API calls

2. Cache calendar data when possible
3. Batch calendar operations
4. Request quota increase from Google

```
# Implement exponential backoff
import time
import random
from googleapiclient.errors import HttpError

def calendar_operation_with_retry(operation, max_retries=5):
    """Execute calendar operation with exponential backoff"""

    for attempt in range(max_retries):
        try:
            return operation()
        except HttpError as e:
            if e.resp.status in [429, 500, 503, 504]:  # Retriable errors
                wait_time = (2 ** attempt) + random.uniform(0, 1)
                print(f"Rate limited, waiting {wait_time:.2f} seconds...")
                time.sleep(wait_time)
                continue
            else:
                raise e

    raise Exception(f"Operation failed after {max_retries} retries")
```

3. Application Performance Issues

Problem: "Slow Response Times"

Symptoms:
- Calls take long to be answered
- Long pauses during conversation
- Webhook timeouts

Diagnosis:

```
# Monitor application performance
top -p $(pgrep -f "python.*app.py")

# Check memory usage
free -h

# Monitor disk I/O
iostat -x 1

# Check database connections (if applicable)
netstat -tn | grep :5432 | wc -l
```

Solutions:
1. **Add Caching:**

```python
from functools import lru_cache
import time

# Cache pricing calculations
@lru_cache(maxsize=128)
def get_cached_pricing(service_type, date_str, duration):
    return pricing_engine.calculate_rate(service_type, date_str, duration)

# Cache calendar availability for 5 minutes
availability_cache = {}

def get_cached_availability(date_key):
    if date_key in availability_cache:
        cache_time, data = availability_cache[date_key]
        if time.time() - cache_time < 300:  # 5 minutes
            return data

    data = calendar_helper.check_availability(date_key)
    availability_cache[date_key] = (time.time(), data)
    return data
```

1. **Database Connection Pooling:**

```python
from sqlalchemy import create_engine
from sqlalchemy.pool import QueuePool

# Configure connection pool
engine = create_engine(
    'postgresql://user:pass@localhost/db',
    poolclass=QueuePool,
    pool_size=10,
    max_overflow=20,
    pool_pre_ping=True
)
```

1. **Async Processing:**

```python
import asyncio
from concurrent.futures import ThreadPoolExecutor

# Process components concurrently
async def process_speech_async(speech_input, context):
    loop = asyncio.get_event_loop()

    with ThreadPoolExecutor(max_workers=3) as executor:
        # Run NLU, pricing, and calendar checks concurrently
        nlu_future = loop.run_in_executor(executor, nlu.process_speech_input,
speech_input, context)
        pricing_future = loop.run_in_executor(executor, pricing.get_default_rates)
        calendar_future = loop.run_in_executor(executor, calendar.get_today_availabili
ty)

        nlu_result, pricing_data, calendar_data = await asyncio.gather(
            nlu_future, pricing_future, calendar_future
        )

        return combine_results(nlu_result, pricing_data, calendar_data)
```

4. Audio Quality Issues

Problem: "Poor Speech Recognition"

Symptoms:
- System frequently asks to repeat
- Wrong intent detection
- Cannot understand customer speech

Solutions:
1. **Improve Speech Recognition Settings:**

```
# In NCCO generation
{
    'action': 'input',
    'speech': {
        'language': 'en-US',
        'context': ['basketball', 'court', 'rental', 'booking'],  # Context hints
        'endOnSilence': 2,  # Wait 2 seconds after speech stops
        'maxDuration': 10,  # Maximum 10 seconds of speech
        'saveAudio': True   # Save for analysis
    }
}
```

1. **Add Confirmation Steps:**

```
def create_confirmation_ncco(understood_intent, entities):
    """Confirm understanding before proceeding"""

    confirmation_text = f"I understand you want to {understood_intent}"
    if entities.get('date'):
        confirmation_text += f" for {entities['date']}"
    if entities.get('duration'):
        confirmation_text += f" for {entities['duration']} hours"

    confirmation_text += ". Is that correct? Say yes or no."

    return [
        {
            'action': 'talk',
            'text': confirmation_text,
            'voiceName': 'Amy'
        },
        {
            'action': 'input',
            'speech': {
                'language': 'en-US',
                'endOnSilence': 2,
                'maxDuration': 5,
                'context': ['yes', 'no', 'correct', 'wrong']
            }
        }
    ]
```

5. Business Logic Issues

Problem: "Incorrect Pricing Calculations"

Diagnosis Script:

```
#!/usr/bin/env python3
"""Diagnose pricing issues"""

from datetime import datetime, timedelta
from pricing import PricingEngine

def diagnose_pricing_issues():
    pricing = PricingEngine()

    # Test scenarios
    test_cases = [
        ('basketball', datetime(2025, 10, 6, 14, 0), 2),  # Monday 2 PM, 2 hours
        ('basketball', datetime(2025, 10, 11, 19, 0), 1), # Saturday 7 PM, 1 hour
        ('birthday_party', datetime(2025, 10, 12, 15, 0), 3) # Sunday 3 PM, 3 hours
    ]

    for service_type, start_time, duration in test_cases:
        try:
            result = pricing.calculate_rate(service_type, start_time, duration)
            print(f"\n{service_type} - {start_time.strftime('%A %I:%M %p')} - {dura-
tion}h:")
            print(f"  Base rate: ${result['base_rate']}")
            print(f"  Final rate: ${result['hourly_rate']}")
            print(f"  Total cost: ${result['total_cost']}")
            print(f"  Multipliers: {result.get('multipliers', {})}")

        except Exception as e:
            print(f"Error calculating pricing for {service_type}: {e}")

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

Diagnostic Tools

1. System Health Check

Comprehensive Health Check Script

```python
#!/usr/bin/env python3
"""System health check script"""

import os
import sys
import subprocess
import requests
import json
from datetime import datetime

class SystemHealthChecker:

    def __init__(self):
        self.checks = [
            ('Environment Variables', self.check_environment),
            ('File System', self.check_files),
            ('Python Dependencies', self.check_dependencies),
            ('Application Health', self.check_application),
            ('Vonage Connectivity', self.check_vonage),
            ('Google Calendar', self.check_calendar),
            ('System Resources', self.check_resources)
        ]

    def run_all_checks(self):
        """Run all health checks"""
        print(f"System Health Check - {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
        print("=" * 60)

        results = {}

        for check_name, check_func in self.checks:
            print(f"\n{check_name}:")
            print("-" * len(check_name))

            try:
                result = check_func()
                results[check_name] = result
                status = "✅ PASS" if result['status'] else "❌ FAIL"
                print(f"Status: {status}")

                if result.get('details'):
                    for detail in result['details']:
                        print(f"  {detail}")

                if result.get('warnings'):
                    for warning in result['warnings']:
                        print(f"  ⚠️  {warning}")

            except Exception as e:
                results[check_name] = {'status': False, 'error': str(e)}
                print(f"Status: ❌ ERROR - {e}")

        # Summary
        passed = sum(1 for r in results.values() if r['status'])
        total = len(results)

        print(f"\n{'='*60}")
        print(f"Summary: {passed}/{total} checks passed")

        if passed == total:
            print("🎉 System is healthy!")
            return True
        else:
            print("⚠️ System has issues that need attention")
            return False

    def check_environment(self):
        """Check environment variables"""
        required_vars = [
            'VONAGE_API_KEY', 'VONAGE_API_SECRET', 'VONAGE_APPLICATION_ID',
            'VONAGE_PRIVATE_KEY_PATH', 'VONAGE_PHONE_NUMBER',
            'GOOGLE_CALENDAR_ID', 'GOOGLE_CREDENTIALS_PATH'
        ]

        missing = [var for var in required_vars if not os.getenv(var)]

        return {
            'status': len(missing) == 0,
            'details': [f"Found {len(required_vars) - len(missing)}/{len(re-
quired_vars)} required variables"],
            'warnings': [f"Missing: {var}" for var in missing]
        }

    def check_files(self):
        """Check required files"""
        required_files = [
            os.getenv('VONAGE_PRIVATE_KEY_PATH', './private.key'),
            os.getenv('GOOGLE_CREDENTIALS_PATH', './credentials.json'),
            'app.py', 'nlu.py', 'calendar_helper.py', 'pricing.py'
        ]

        missing = [f for f in required_files if not os.path.exists(f)]

        details = []
        for file in required_files:
            if os.path.exists(file):
                size = os.path.getsize(file)
                details.append(f"{file}: {size} bytes")

        return {
            'status': len(missing) == 0,
            'details': details,
            'warnings': [f"Missing file: {f}" for f in missing]
        }

    def check_application(self):
        """Check if application is running"""
        try:
            response = requests.get('http://localhost:5000/health', timeout=5)
            return {
                'status': response.status_code == 200,
                'details': [f"HTTP {response.status_code}: {response.text}"]
            }
        except requests.exceptions.ConnectionError:
            return {
                'status': False,
                'details': ['Application not responding on port 5000']
            }
        except Exception as e:
            return {
                'status': False,
                'details': [f'Health check error: {e}']
            }

    def check_resources(self):
        """Check system resources"""
        # Check disk space
        disk_usage = subprocess.check_output(['df', '-h', '/']).decode().split('\n')
[1].split()
        disk_free = disk_usage[3]
        disk_percent = disk_usage[4]

        # Check memory
        memory_info = subprocess.check_output(['free', '-h']).decode().split('\n')
[1].split()
        memory_used = memory_info[2]
        memory_available = memory_info[6]

        # Check load average
        load_avg = os.getloadavg()[0]

        warnings = []
        if int(disk_percent[:-1]) > 90:
            warnings.append(f"High disk usage: {disk_percent}")
        if load_avg > 2.0:
            warnings.append(f"High load average: {load_avg}")

        return {
            'status': len(warnings) == 0,
            'details': [
                f"Disk free: {disk_free} ({disk_percent} used)",
                f"Memory: {memory_used} used, {memory_available} available",
                f"Load average: {load_avg}"
            ],
            'warnings': warnings
        }

if __name__ == "__main__":
    checker = SystemHealthChecker()
    healthy = checker.run_all_checks()
    sys.exit(0 if healthy else 1)
```

2. Log Analysis Tools

Log Analyzer Script

```
#!/usr/bin/env python3
"""Analyze application logs for issues"""

import re
import json
from datetime import datetime, timedelta
from collections import Counter, defaultdict

class LogAnalyzer:

    def __init__(self, log_file='/var/log/auto-call-system/app.log'):
        self.log_file = log_file
        self.patterns = {
            'error': re.compile(r'ERROR|Exception|Traceback'),
            'warning': re.compile(r'WARNING|WARN'),
            'call_start': re.compile(r'conversation_uuid.*from.*(\+\d+)'),
            'intent_detected': re.compile(r'intent.*detected.*(\w+)'),
            'escalation': re.compile(r'escalation.*type.*(\w+)'),
            'booking_created': re.compile(r'booking.*created.*event_id.*([a-zA-
Z0-9]+)'),
            'response_time': re.compile(r'response_time.*(\d+\.?\d*).*ms')
        }

    def analyze_recent_logs(self, hours=24):
        """Analyze logs from the last N hours"""

        cutoff_time = datetime.now() - timedelta(hours=hours)

        stats = {
            'total_calls': 0,
            'errors': [],
            'warnings': [],
            'intent_distribution': Counter(),
            'escalation_types': Counter(),
            'bookings_created': 0,
            'avg_response_time': 0,
            'call_volume_by_hour': defaultdict(int)
        }

        response_times = []

        try:
            with open(self.log_file, 'r') as f:
                for line in f:
                    # Parse timestamp (assuming ISO format)
                    if line.startswith('2025'):
                        timestamp_str = line.split()[0] + ' ' + line.split()[1]
                        try:
                            log_time = datetime.fromisoformat(timestamp_str.replace('T
', ' ').split('.')[0])
                            if log_time < cutoff_time:
                                continue
                        except:
                            continue

                    # Analyze log line
                    if self.patterns['error'].search(line):
                        stats['errors'].append(line.strip())

                    if self.patterns['warning'].search(line):
                        stats['warnings'].append(line.strip())
                    if self.patterns['call_start'].search(line):
                        stats['total_calls'] += 1
                        hour = log_time.hour if 'log_time' in locals() else 0
                        stats['call_volume_by_hour'][hour] += 1

                    intent_match = self.patterns['intent_detected'].search(line)
                    if intent_match:
                        stats['intent_distribution'][intent_match.group(1)] += 1

                    escalation_match = self.patterns['escalation'].search(line)
                    if escalation_match:
                        stats['escalation_types'][escalation_match.group(1)] += 1

                    if self.patterns['booking_created'].search(line):
                        stats['bookings_created'] += 1

                    response_match = self.patterns['response_time'].search(line)
                    if response_match:
                        response_times.append(float(response_match.group(1)))

        except FileNotFoundError:
            print(f"Log file not found: {self.log_file}")
            return None

        if response_times:
            stats['avg_response_time'] = sum(response_times) / len(response_times)

        return stats

    def generate_report(self, hours=24):
        """Generate analysis report"""

        stats = self.analyze_recent_logs(hours)
        if not stats:
            return

        print(f"Log Analysis Report - Last {hours} Hours")
        print("=" * 50)

        print(f"\nCall Statistics:")
        print(f"  Total calls: {stats['total_calls']}")
        print(f"  Bookings created: {stats['bookings_created']}")
        print(f"  Conversion rate: {(stats['bookings_created']/stats['total_calls']*10
0):.1f}%" if stats['total_calls'] > 0 else "  No calls to analyze")
        print(f"  Average response time: {stats['avg_response_time']:.1f}ms")

        print(f"\nIntent Distribution:")
        for intent, count in stats['intent_distribution'].most_common():
            percentage = (count / sum(stats['intent_distribution'].values())) * 100 if
stats['intent_distribution'] else 0
            print(f"  {intent}: {count} ({percentage:.1f}%)")

        if stats['escalation_types']:
            print(f"\nEscalation Types:")
            for escalation_type, count in stats['escalation_types'].most_common():
                print(f"  {escalation_type}: {count}")

        print(f"\nCall Volume by Hour:")
        for hour in range(24):
            if hour in stats['call_volume_by_hour']:
                print(f"  {hour:02d}:00 - {stats['call_volume_by_hour'][hour]} calls")

        if stats['errors']:
            print(f"\nRecent Errors ({len(stats['errors'])}):")
            for error in stats['errors'][-5:]:  # Show last 5 errors
                print(f"  {error}")

        if stats['warnings']:
            print(f"\nRecent Warnings ({len(stats['warnings'])}):")
            for warning in stats['warnings'][-5:]:  # Show last 5 warnings
                print(f"  {warning}")

if __name__ == "__main__":
    analyzer = LogAnalyzer()
    analyzer.generate_report(24)
```

Monitoring and Health Checks

1. Continuous Monitoring Setup

Application Health Endpoint

```python
# Add to app.py
@app.route('/health', methods=['GET'])
def health_check():
    """Comprehensive health check endpoint"""

    health_status = {
        'timestamp': datetime.now().isoformat(),
        'status': 'healthy',
        'checks': {}
    }

    # Check database connectivity (if applicable)
    try:
        # Test database connection
        health_status['checks']['database'] = {'status': 'ok'}
    except Exception as e:
        health_status['checks']['database'] = {'status': 'error', 'message': str(e)}
        health_status['status'] = 'unhealthy'

    # Check Google Calendar API
    try:
        calendar_helper.service.calendars().get(calendarId=calendar_helper.calen-
dar_id).execute()
        health_status['checks']['google_calendar'] = {'status': 'ok'}
    except Exception as e:
        health_status['checks']['google_calendar'] = {'status': 'error', 'message': st
r(e)}
        health_status['status'] = 'unhealthy'

    # Check Vonage API
    try:
        if vonage_client:
            # Test API connectivity
            health_status['checks']['vonage_api'] = {'status': 'ok'}
        else:
            health_status['checks']['vonage_api'] = {'status': 'error', 'message': 'Cl
ient not initialized'}
            health_status['status'] = 'unhealthy'
    except Exception as e:
        health_status['checks']['vonage_api'] = {'status': 'error', 'message': str(e)}
        health_status['status'] = 'unhealthy'

    # Check system resources
    import psutil

    cpu_percent = psutil.cpu_percent(interval=1)
    memory_percent = psutil.virtual_memory().percent
    disk_percent = psutil.disk_usage('/').percent

    health_status['checks']['system_resources'] = {
        'cpu_percent': cpu_percent,
        'memory_percent': memory_percent,
        'disk_percent': disk_percent,
        'status': 'ok' if cpu_percent < 80 and memory_percent < 80 and disk_percent <
90 else 'warning'
    }

    status_code = 200 if health_status['status'] == 'healthy' else 503
    return jsonify(health_status), status_code
```

2. External Monitoring

Monitoring Script for Cron

```
#!/bin/bash
# monitor_system.sh - Run every 5 minutes via cron

APP_URL="http://localhost:5000"
LOG_FILE="/var/log/auto-call-system/monitoring.log"
ALERT_EMAIL="admin@yourcompany.com"

# Function to log with timestamp
log_message() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" >> $LOG_FILE
}

# Check application health
check_health() {
    local response=$(curl -s -o /dev/null -w "%{http_code}" "$APP_URL/health" --max-
time 10)

    if [ "$response" = "200" ]; then
        log_message "Health check: OK"
        return 0
    else
        log_message "Health check: FAILED (HTTP $response)"
        return 1
    fi
}

# Check disk space
check_disk_space() {
    local usage=$(df / | tail -1 | awk '{print $5}' | sed 's/%//')

    if [ "$usage" -gt 90 ]; then
        log_message "Disk space warning: ${usage}% used"
        return 1
    else
        log_message "Disk space: OK (${usage}% used)"
        return 0
    fi
}

# Check memory usage
check_memory() {
    local usage=$(free | awk 'FNR==2{printf "%.0f", $3/($3+$4)*100}')

    if [ "$usage" -gt 85 ]; then
        log_message "Memory warning: ${usage}% used"
        return 1
    else
        log_message "Memory: OK (${usage}% used)"
        return 0
    fi
}

# Send alert email
send_alert() {
    local message="$1"
    echo "$message" | mail -s "Auto Call System Alert" "$ALERT_EMAIL"
    log_message "Alert sent: $message"
}

# Main monitoring logic
main() {
    local issues=0
    if ! check_health; then
        issues=$((issues + 1))
        send_alert "Application health check failed"
    fi

    if ! check_disk_space; then
        issues=$((issues + 1))
        send_alert "Disk space warning"
    fi

    if ! check_memory; then
        issues=$((issues + 1))
        send_alert "Memory usage warning"
    fi

    if [ $issues -eq 0 ]; then
        log_message "All systems normal"
    else
        log_message "Found $issues issues"
    fi
}

# Run main function
main
```

Setup Cron Job

```
# Add to crontab
crontab -e

# Add this line to run monitoring every 5 minutes
*/5 * * * * /opt/scripts/monitor_system.sh
```

Emergency Procedures

1. System Failure Response

Emergency Recovery Script

```bash
#!/bin/bash
# emergency_recovery.sh

echo "Starting emergency recovery procedures..."

# 1. Check if application is running
if ! pgrep -f "python.*app.py" > /dev/null; then
    echo "Application not running, attempting restart..."
    systemctl restart auto-call-system
    sleep 10
fi

# 2. Check application health
if ! curl -f http://localhost:5000/health > /dev/null 2>&1; then
    echo "Application health check failed"

    # Try restarting the service
    systemctl stop auto-call-system
    sleep 5
    systemctl start auto-call-system
    sleep 15

    # Check again
    if ! curl -f http://localhost:5000/health > /dev/null 2>&1; then
        echo "Application still failing, checking logs..."
        journalctl -u auto-call-system --no-pager -n 50

        # Emergency fallback: redirect calls to staff
        # This requires configuring Vonage to redirect to a backup number
        echo "Consider manual intervention required"
        exit 1
    fi
fi

echo "System recovery completed successfully"
```

2. Backup and Restore Procedures

Emergency Backup

```bash
#!/bin/bash
# emergency_backup.sh

BACKUP_DIR="/opt/backups/emergency"
TIMESTAMP=$(date +%Y%m%d_%H%M%S)
BACKUP_FILE="$BACKUP_DIR/emergency_backup_$TIMESTAMP.tar.gz"

mkdir -p $BACKUP_DIR

# Backup application and configuration
tar -czf "$BACKUP_FILE" \
    --exclude="venv" \
    --exclude="__pycache__" \
    --exclude="*.pyc" \
    /opt/auto_call_system/

# Backup logs
tar -czf "$BACKUP_DIR/logs_$TIMESTAMP.tar.gz" /var/log/auto-call-system/

echo "Emergency backup completed: $BACKUP_FILE"
```

3. Call Routing Failover

Vonage Failover Configuration

```python
# emergency_failover.py
"""Emergency call routing to staff"""

import os
import vonage

def activate_emergency_routing():
    """Redirect all calls to staff member"""

    client = vonage.Client(
        key=os.getenv('VONAGE_API_KEY'),
        secret=os.getenv('VONAGE_API_SECRET')
    )

    # Update application to route directly to staff
    emergency_ncco = [
        {
            "action": "talk",
            "text": "We're experiencing technical difficulties. Connecting you to a staff member.",
            "voiceName": "Amy"
        },
        {
            "action": "connect",
            "endpoint": [
                {
                    "type": "phone",
                    "number": os.getenv('STAFF_PHONE_NUMBER')
                }
            ]
        }
    ]

    # This would require updating your webhook to return emergency NCCO
    print("Emergency routing activated")
    return emergency_ncco

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

Testing and Troubleshooting Complete! 🔧

This comprehensive guide provides you with all the tools and procedures needed to test, monitor, and troubleshoot your automated phone answering system. Regular testing and monitoring will ensure optimal performance and quick resolution of any issues that arise.

Use these tools proactively to maintain system health and provide excellent customer service!