Maintenance Guide - Automated Phone Answering System

Table of Contents

1. Maintenance Overview
2. Regular Maintenance Tasks
3. System Updates and Upgrades
4. Security Maintenance
5. Performance Optimization
6. Data Management and Backups
7. Log Management
8. Certificate Management
9. Capacity Planning and Scaling
10. Maintenance Schedules

Maintenance Overview

Maintenance Philosophy

Proactive maintenance ensures your automated phone answering system remains:
- **Reliable**: Consistent uptime and performance
- **Secure**: Protected against vulnerabilities
- **Efficient**: Optimal resource utilization
- **Scalable**: Ready for growth and increased demand
- **Compliant**: Meeting regulatory and business requirements

Maintenance Categories

Category	Frequency	Impact	Examples
Critical	Immediate	High	Security patches, service failures
Important	Weekly	Medium	Performance monitoring, log rotation
Routine	Monthly	Low	Updates, capacity planning
Strategic	Quarterly	Variable	Architecture reviews, scaling

Regular Maintenance Tasks

Daily Tasks

1. System Health Check

Automated Health Monitoring

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

LOG_FILE="/var/log/auto-call-system/maintenance.log"

log_message() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" | tee -a $LOG_FILE
}

# Check system health
log_message "Starting daily health check"

# Application status
if systemctl is-active --quiet auto-call-system; then
    log_message "✅ Application service: Running"
else
    log_message "❌ Application service: Stopped"
    systemctl restart auto-call-system
    log_message "🔄 Attempted service restart"
fi

# Health endpoint check
if curl -sf http://localhost:5000/health > /dev/null; then
    log_message "✅ Health endpoint: OK"
else
    log_message "❌ Health endpoint: Failed"
fi

# Resource usage
CPU_USAGE=$(top -bn1 | grep "Cpu(s)" | awk '{print $2}' | awk -F'%' '{print $1}')
MEMORY_USAGE=$(free | awk 'FNR==2{printf "%.0f", $3/($3+$4)*100}')
DISK_USAGE=$(df / | tail -1 | awk '{print $5}' | sed 's/%//')

log_message "📊 CPU: ${CPU_USAGE}%, Memory: ${MEMORY_USAGE}%, Disk: ${DISK_USAGE}%"

# Alert if thresholds exceeded
if (( $(echo "$CPU_USAGE > 80" | bc -l) )); then
    log_message "⚠️  High CPU usage: ${CPU_USAGE}%"
fi

if [ "$MEMORY_USAGE" -gt 80 ]; then
    log_message "⚠️  High memory usage: ${MEMORY_USAGE}%"
fi

if [ "$DISK_USAGE" -gt 85 ]; then
    log_message "⚠️  High disk usage: ${DISK_USAGE}%"
fi

log_message "Daily health check completed"
```

Schedule Daily Check

```
# Add to crontab
crontab -e

# Run daily health check at 6 AM
0 6 * * * /opt/scripts/daily_health_check.sh
```

2. Call Volume and Performance Monitoring

Daily Metrics Collection

```python
#!/usr/bin/env python3
"""Daily metrics collection and reporting"""

import json
import sqlite3
from datetime import datetime, timedelta
from collections import defaultdict
import requests

class DailyMetricsCollector:

    def __init__(self):
        self.db_path = '/var/lib/auto-call-system/metrics.db'
        self.init_database()

    def init_database(self):
        """Initialize metrics database"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute('''
            CREATE TABLE IF NOT EXISTS daily_metrics (
                date TEXT PRIMARY KEY,
                total_calls INTEGER,
                successful_calls INTEGER,
                failed_calls INTEGER,
                escalations INTEGER,
                bookings_created INTEGER,
                avg_response_time REAL,
                peak_concurrent_calls INTEGER,
                system_uptime REAL
            )
        ''')

        conn.commit()
        conn.close()

    def collect_daily_metrics(self):
        """Collect metrics for yesterday"""
        yesterday = (datetime.now() - timedelta(days=1)).date()

        # Parse application logs for metrics
        metrics = self.parse_application_logs(yesterday)

        # Get system metrics
        system_metrics = self.get_system_metrics()

        # Combine metrics
        combined_metrics = {
            'date': yesterday.isoformat(),
            'total_calls': metrics['total_calls'],
            'successful_calls': metrics['successful_calls'],
            'failed_calls': metrics['failed_calls'],
            'escalations': metrics['escalations'],
            'bookings_created': metrics['bookings_created'],
            'avg_response_time': metrics['avg_response_time'],
            'peak_concurrent_calls': metrics['peak_concurrent_calls'],
            'system_uptime': system_metrics['uptime']
        }

        # Store metrics
        self.store_metrics(combined_metrics)
```

```
        # Generate daily report
        self.generate_daily_report(combined_metrics)

        return combined_metrics

    def parse_application_logs(self, target_date):
        """Parse application logs for metrics"""
        # Implementation would parse actual log files
        # This is a simplified version

        metrics = {
            'total_calls': 0,
            'successful_calls': 0,
            'failed_calls': 0,
            'escalations': 0,
            'bookings_created': 0,
            'avg_response_time': 0,
            'peak_concurrent_calls': 0
        }

        # Parse logs and populate metrics
        # Implementation details depend on log format

        return metrics

    def store_metrics(self, metrics):
        """Store metrics in database"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute('''
            INSERT OR REPLACE INTO daily_metrics
            (date, total_calls, successful_calls, failed_calls, escalations,
             bookings_created, avg_response_time, peak_concurrent_calls, sys-
tem_uptime)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
        ''', (
            metrics['date'],
            metrics['total_calls'],
            metrics['successful_calls'],
            metrics['failed_calls'],
            metrics['escalations'],
            metrics['bookings_created'],
            metrics['avg_response_time'],
            metrics['peak_concurrent_calls'],
            metrics['system_uptime']
        ))

        conn.commit()
        conn.close()

if __name__ == "__main__":
    collector = DailyMetricsCollector()
    collector.collect_daily_metrics()
```

Weekly Tasks

1. System Update Check

Weekly Update Script

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

LOG_FILE="/var/log/auto-call-system/maintenance.log"

log_message() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" | tee -a $LOG_FILE
}

log_message "Starting weekly maintenance"

# Update package lists
log_message "Updating package lists..."
apt update

# Check for security updates
SECURITY_UPDATES=$(apt list --upgradable 2>/dev/null | grep -c security)
if [ "$SECURITY_UPDATES" -gt 0 ]; then
    log_message "⚠️  $SECURITY_UPDATES security updates available"

    # Apply security updates automatically
    apt upgrade -y --only-upgrade $(apt list --upgradable 2>/dev/null | grep security
| cut -d'/' -f1)
    log_message "✅ Security updates applied"

    # Check if restart is required
    if [ -f /var/run/reboot-required ]; then
        log_message "⚠️  System restart required"
        # Schedule restart during maintenance window
        # at 02:00 next Sunday
        echo "shutdown -r +5 'System restart for updates'" | at 02:00 next sunday
    fi
fi

# Check Python package updates
log_message "Checking Python package updates..."
cd /opt/auto_call_system
source venv/bin/activate

pip list --outdated --format=json > /tmp/outdated_packages.json
OUTDATED_COUNT=$(cat /tmp/outdated_packages.json | jq length)

if [ "$OUTDATED_COUNT" -gt 0 ]; then
    log_message "📦 $OUTDATED_COUNT Python packages can be updated"

    # Update non-critical packages
    pip install --upgrade pip setuptools wheel

    # Log outdated packages for review
    cat /tmp/outdated_packages.json | jq -r '.[] | "\(.name): \(.version) -> \
(.latest_version)"' >> $LOG_FILE
fi

log_message "Weekly maintenance completed"
```

2. Log Analysis and Cleanup

Weekly Log Analysis

```
#!/usr/bin/env python3
"""Weekly log analysis and cleanup"""

import os
import gzip
import shutil
from datetime import datetime, timedelta
import glob
import subprocess

class WeeklyLogMaintenance:

    def __init__(self):
        self.log_dirs = [
            '/var/log/auto-call-system/',
            '/var/log/nginx/',
        ]
        self.retention_days = 30
        self.compress_after_days = 7

    def run_weekly_maintenance(self):
        """Run all weekly log maintenance tasks"""
        print("Starting weekly log maintenance...")

        for log_dir in self.log_dirs:
            if os.path.exists(log_dir):
                self.process_log_directory(log_dir)

        self.generate_weekly_log_report()
        self.cleanup_old_logs()

        print("Weekly log maintenance completed")

    def process_log_directory(self, log_dir):
        """Process logs in a directory"""
        print(f"Processing logs in {log_dir}")

        # Compress old logs
        week_ago = datetime.now() - timedelta(days=self.compress_after_days)

        for log_file in glob.glob(f"{log_dir}*.log"):
            file_mtime = datetime.fromtimestamp(os.path.getmtime(log_file))

            if file_mtime < week_ago and not log_file.endswith('.gz'):
                compressed_file = f"{log_file}.gz"

                if not os.path.exists(compressed_file):
                    print(f"Compressing {log_file}")
                    with open(log_file, 'rb') as f_in:
                        with gzip.open(compressed_file, 'wb') as f_out:
                            shutil.copyfileobj(f_in, f_out)

                    # Verify compression
                    if os.path.exists(compressed_file):
                        os.remove(log_file)
                        print(f"✅ Compressed and removed {log_file}")

    def cleanup_old_logs(self):
        """Remove logs older than retention period"""
        cutoff_date = datetime.now() - timedelta(days=self.retention_days)

        for log_dir in self.log_dirs:
```

```
            if not os.path.exists(log_dir):
                continue

            for log_file in glob.glob(f"{log_dir}*"):
                file_mtime = datetime.fromtimestamp(os.path.getmtime(log_file))

                if file_mtime < cutoff_date:
                    print(f"Removing old log: {log_file}")
                    os.remove(log_file)

    def generate_weekly_log_report(self):
        """Generate weekly log analysis report"""
        report_file = f"/var/log/auto-call-system/weekly_report_{datetime.now().strf-
time('%Y%m%d')}.txt"

        with open(report_file, 'w') as f:
            f.write(f"Weekly Log Analysis Report - {datetime.now().strftime('%Y-%m-
%d')}\n")
            f.write("=" * 60 + "\n\n")

            # Analyze application logs
            app_log_analysis = self.analyze_application_logs()
            f.write("Application Logs Analysis:\n")
            f.write(f"  Error count: {app_log_analysis['errors']}\n")
            f.write(f"  Warning count: {app_log_analysis['warnings']}\n")
            f.write(f"  Total calls: {app_log_analysis['total_calls']}\n")
            f.write(f"  Success rate: {app_log_analysis['success_rate']:.1f}%\n\n")

            # System resource trends
            f.write("System Resource Trends:\n")
            f.write("  See attached metrics for detailed analysis\n\n")

        print(f"Weekly report generated: {report_file}")

if __name__ == "__main__":
    maintenance = WeeklyLogMaintenance()
    maintenance.run_weekly_maintenance()
```

Monthly Tasks

1. Capacity Planning Review

Monthly Capacity Analysis

```
#!/usr/bin/env python3
"""Monthly capacity planning analysis"""

import sqlite3
import matplotlib.pyplot as plt
from datetime import datetime, timedelta
import pandas as pd
import numpy as np

class CapacityPlanningAnalysis:

    def __init__(self):
        self.db_path = '/var/lib/auto-call-system/metrics.db'

    def generate_monthly_report(self):
        """Generate monthly capacity planning report"""

        # Get last 3 months of data
        end_date = datetime.now().date()
        start_date = end_date - timedelta(days=90)

        # Load data
        df = self.load_metrics_data(start_date, end_date)

        if df.empty:
            print("No data available for analysis")
            return

        # Generate analysis
        analysis = {
            'call_volume_trend': self.analyze_call_volume_trend(df),
            'resource_utilization': self.analyze_resource_utilization(df),
            'performance_metrics': self.analyze_performance_trends(df),
            'capacity_recommendations': self.generate_capacity_recommendations(df)
        }

        # Generate visualizations
        self.create_capacity_charts(df)

        # Generate written report
        self.write_capacity_report(analysis)

        return analysis

    def load_metrics_data(self, start_date, end_date):
        """Load metrics data from database"""
        conn = sqlite3.connect(self.db_path)

        query = """
            SELECT * FROM daily_metrics
            WHERE date BETWEEN ? AND ?
            ORDER BY date
        """

        df = pd.read_sql_query(query, conn, params=[start_date.isoformat(), end_date.i
soformat()])
        conn.close()

        if not df.empty:
            df['date'] = pd.to_datetime(df['date'])

        return df
```

```
    def analyze_call_volume_trend(self, df):
        """Analyze call volume trends"""
        # Calculate growth rate
        recent_avg = df.tail(7)['total_calls'].mean()
        older_avg = df.head(7)['total_calls'].mean()

        growth_rate = ((recent_avg - older_avg) / older_avg * 100) if older_avg > 0 el
se 0

        # Identify patterns
        df['day_of_week'] = df['date'].dt.dayofweek
        weekly_pattern = df.groupby('day_of_week')['total_calls'].mean()

        peak_day = weekly_pattern.idxmax()
        peak_volume = weekly_pattern.max()

        return {
            'growth_rate': growth_rate,
            'peak_day': peak_day,
            'peak_volume': peak_volume,
            'weekly_pattern': weekly_pattern.to_dict()
        }

    def generate_capacity_recommendations(self, df):
        """Generate capacity recommendations"""

        recommendations = []

        # Analyze current utilization
        avg_calls = df['total_calls'].mean()
        max_calls = df['total_calls'].max()

        # CPU and memory trends
        if df['system_uptime'].mean() < 99.5:
            recommendations.append({
                'priority': 'high',
                'category': 'reliability',
                'recommendation': 'Investigate system downtime causes',
                'details': f"Average uptime: {df['system_uptime'].mean():.1f}%"
            })

        # Response time analysis
        avg_response_time = df['avg_response_time'].mean()
        if avg_response_time > 1000:  # > 1 second
            recommendations.append({
                'priority': 'medium',
                'category': 'performance',
                'recommendation': 'Consider performance optimization',
                'details': f"Average response time: {avg_response_time:.0f}ms"
            })

        # Volume-based scaling recommendations
        if max_calls > avg_calls * 1.5:
            recommendations.append({
                'priority': 'medium',
                'category': 'scaling',
                'recommendation': 'Implement auto-scaling for peak loads',
                'details': f"Peak calls {max_calls} vs average {avg_calls:.0f}"
            })

        return recommendations
```

```
    def create_capacity_charts(self, df):
        """Create visualization charts"""

        fig, axes = plt.subplots(2, 2, figsize=(15, 12))

        # Call volume trend
        axes[0,0].plot(df['date'], df['total_calls'])
        axes[0,0].set_title('Daily Call Volume Trend')
        axes[0,0].set_ylabel('Calls per Day')
        axes[0,0].tick_params(axis='x', rotation=45)

        # Success rate trend
        df['success_rate'] = (df['successful_calls'] / df['total_calls'] * 100).fillna
(0)
        axes[0,1].plot(df['date'], df['success_rate'])
        axes[0,1].set_title('Call Success Rate Trend')
        axes[0,1].set_ylabel('Success Rate (%)')
        axes[0,1].tick_params(axis='x', rotation=45)

        # Response time trend
        axes[1,0].plot(df['date'], df['avg_response_time'])
        axes[1,0].set_title('Average Response Time Trend')
        axes[1,0].set_ylabel('Response Time (ms)')
        axes[1,0].tick_params(axis='x', rotation=45)

        # Weekly pattern
        df['day_of_week'] = df['date'].dt.dayofweek
        weekly_avg = df.groupby('day_of_week')['total_calls'].mean()
        days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
        axes[1,1].bar(days, weekly_avg.values)
        axes[1,1].set_title('Average Calls by Day of Week')
        axes[1,1].set_ylabel('Average Calls')

        plt.tight_layout()

        # Save chart
        chart_path = f"/var/log/auto-call-system/capacity_analysis_{datetime.now().str
ftime('%Y%m')}.png"
        plt.savefig(chart_path, dpi=300, bbox_inches='tight')
        plt.close()

        print(f"Capacity analysis charts saved: {chart_path}")

    def write_capacity_report(self, analysis):
        """Write capacity planning report"""

        report_path = f"/var/log/auto-call-system/capacity_report_{datetime.now().strf
time('%Y%m')}.txt"

        with open(report_path, 'w') as f:
            f.write(f"Monthly Capacity Planning Report\n")
            f.write(f"Generated: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n")
            f.write("=" * 60 + "\n\n")

            # Call volume analysis
            f.write("CALL VOLUME ANALYSIS\n")
            f.write("-" * 30 + "\n")
            f.write(f"Growth rate: {analysis['call_volume_trend']['growth_rate']:+.1f}
%\n")
            f.write(f"Peak day: {analysis['call_volume_trend']['peak_day']} (Day 0 =
Monday)\n")
            f.write(f"Peak volume: {analysis['call_volume_trend']['peak_volume']:.0f}
calls\n\n")
```

```
            # Recommendations
            f.write("CAPACITY RECOMMENDATIONS\n")
            f.write("-" * 30 + "\n")

            for rec in analysis['capacity_recommendations']:
                f.write(f"Priority: {rec['priority'].upper()}\n")
                f.write(f"Category: {rec['category']}\n")
                f.write(f"Recommendation: {rec['recommendation']}\n")
                f.write(f"Details: {rec['details']}\n\n")

        print(f"Capacity report written: {report_path}")

if __name__ == "__main__":
    analyzer = CapacityPlanningAnalysis()
    analyzer.generate_monthly_report()
```

2. Security Audit

Monthly Security Checklist

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

LOG_FILE="/var/log/auto-call-system/security_audit.log"

log_message() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" | tee -a $LOG_FILE
}

log_message "Starting monthly security audit"

# Check file permissions
log_message "Checking file permissions..."

SENSITIVE_FILES=(
    "/opt/auto_call_system/.env"
    "/opt/auto_call_system/private.key"
    "/opt/auto_call_system/credentials.json"
)

for file in "${SENSITIVE_FILES[@]}"; do
    if [ -f "$file" ]; then
        PERMS=$(stat -c "%a" "$file")
        if [ "$PERMS" = "600" ]; then
            log_message "✅ $file: Correct permissions ($PERMS)"
        else
            log_message "⚠️  $file: Incorrect permissions ($PERMS), fixing..."
            chmod 600 "$file"
            log_message "✅ Fixed permissions for $file"
        fi
    else
        log_message "❌ $file: File not found"
    fi
done

# Check for unauthorized SSH keys
log_message "Checking SSH authorized keys..."
AUTH_KEYS_FILE="/home/ubuntu/.ssh/authorized_keys"

if [ -f "$AUTH_KEYS_FILE" ]; then
    KEY_COUNT=$(wc -l < "$AUTH_KEYS_FILE")
    log_message "📋 Found $KEY_COUNT authorized SSH keys"

    # Log key fingerprints for review
    ssh-keygen -lf "$AUTH_KEYS_FILE" >> $LOG_FILE
else
    log_message "ℹ️  No SSH authorized keys file found"
fi

# Check for failed login attempts
log_message "Checking failed login attempts..."
FAILED_LOGINS=$(grep "Failed password" /var/log/auth.log | wc -l)
log_message "📊 Failed login attempts in auth.log: $FAILED_LOGINS"

if [ "$FAILED_LOGINS" -gt 100 ]; then
    log_message "⚠️  High number of failed login attempts detected"
fi

# Check SSL certificate expiration
log_message "Checking SSL certificate expiration..."
DOMAIN=$(hostname -f)
```

```
if command -v openssl >/dev/null 2>&1; then
    CERT_FILE="/etc/letsencrypt/live/$DOMAIN/cert.pem"

    if [ -f "$CERT_FILE" ]; then
        EXPIRY_DATE=$(openssl x509 -enddate -noout -in "$CERT_FILE" | cut -d= -f2)
        EXPIRY_TIMESTAMP=$(date -d "$EXPIRY_DATE" +%s)
        CURRENT_TIMESTAMP=$(date +%s)
        DAYS_UNTIL_EXPIRY=$(( (EXPIRY_TIMESTAMP - CURRENT_TIMESTAMP) / 86400 ))

        log_message "📅 SSL certificate expires in $DAYS_UNTIL_EXPIRY days"

        if [ "$DAYS_UNTIL_EXPIRY" -lt 30 ]; then
            log_message "⚠️  SSL certificate expires soon!"
        fi
    else
        log_message "ℹ️  No SSL certificate found at $CERT_FILE"
    fi
fi

# Check for security updates
log_message "Checking for security updates..."
apt update -qq
SECURITY_UPDATES=$(apt list --upgradable 2>/dev/null | grep security | wc -l)
log_message "📦 Available security updates: $SECURITY_UPDATES"

if [ "$SECURITY_UPDATES" -gt 0 ]; then
    log_message "⚠️  Security updates available - schedule installation"
fi

log_message "Monthly security audit completed"
```

System Updates and Upgrades

1. Python Package Management

Dependency Update Strategy

```
#!/usr/bin/env python3
"""Manage Python package updates safely"""

import subprocess
import json
import sys
import os
from packaging import version
import pkg_resources

class DependencyManager:

    def __init__(self, venv_path="/opt/auto_call_system/venv"):
        self.venv_path = venv_path
        self.pip_path = os.path.join(venv_path, "bin", "pip")

        # Critical packages that need careful handling
        self.critical_packages = {
            'flask': {'max_major_version': 3},
            'vonage': {'max_major_version': 5},
            'google-api-python-client': {'max_major_version': 3}
        }

    def check_outdated_packages(self):
        """Check for outdated packages"""
        result = subprocess.run([
            self.pip_path, "list", "--outdated", "--format=json"
        ], capture_output=True, text=True)

        if result.returncode != 0:
            print(f"Error checking packages: {result.stderr}")
            return []

        return json.loads(result.stdout)

    def safe_update_package(self, package_name, current_version, latest_version):
        """Safely update a package with compatibility checks"""

        current_ver = version.parse(current_version)
        latest_ver = version.parse(latest_version)

        # Check if it's a critical package
        if package_name in self.critical_packages:
            rules = self.critical_packages[package_name]

            if latest_ver.major > rules.get('max_major_version', float('inf')):
                print(f"⚠️  Skipping {package_name}: Major version change detected")
                return False

        # For minor/patch updates, proceed
        if latest_ver.major == current_ver.major:
            print(f"Updating {package_name}: {current_version} -> {latest_version}")

            # Create backup of current environment
            self.backup_environment()

            # Update package
            result = subprocess.run([
                self.pip_path, "install", "--upgrade", f"{package_name}=={latest_versi
on}"
            ], capture_output=True, text=True)
```

```
            if result.returncode == 0:
                # Test system after update
                if self.test_system_after_update():
                    print(f"✅ Successfully updated {package_name}")
                    return True
                else:
                    print(f"❌ Update failed tests, rolling back {package_name}")
                    self.rollback_package(package_name, current_version)
                    return False
            else:
                print(f"❌ Failed to update {package_name}: {result.stderr}")
                return False

        return False

    def backup_environment(self):
        """Backup current environment"""
        backup_file = f"/opt/backups/requirements_backup_{datetime.now().strftime('%Y%
m%d_%H%M%S')}.txt"

        subprocess.run([
            self.pip_path, "freeze"
        ], stdout=open(backup_file, 'w'))

        print(f"Environment backed up to {backup_file}")

    def test_system_after_update(self):
        """Test system functionality after package update"""
        try:
            # Test imports
            import flask
            import vonage
            from google.oauth2 import service_account

            # Test basic functionality
            from nlu import SportsRentalNLU
            nlu = SportsRentalNLU()
            test_result = nlu.process_speech_input("test", {})

            return True

        except Exception as e:
            print(f"System test failed: {e}")
            return False

    def update_all_safe_packages(self):
        """Update all packages safely"""
        outdated = self.check_outdated_packages()

        if not outdated:
            print("All packages are up to date")
            return

        print(f"Found {len(outdated)} outdated packages")

        updated_count = 0
        for package in outdated:
            if self.safe_update_package(
                package['name'],
                package['version'],
                package['latest_version']
            ):
                updated_count += 1
```

```
        print(f"Successfully updated {updated_count}/{len(outdated)} packages")

if __name__ == "__main__":
    manager = DependencyManager()
    manager.update_all_safe_packages()
```

2. System Package Updates

Automated System Updates

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

LOG_FILE="/var/log/auto-call-system/system_updates.log"

log_message() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" | tee -a $LOG_FILE
}

# Function to check if reboot is required
check_reboot_required() {
    if [ -f /var/run/reboot-required ]; then
        log_message "System reboot required after updates"

        # Schedule reboot during maintenance window (2 AM next day)
        echo "shutdown -r now 'Automated reboot for system updates'" | at 02:00 tomor-
row
        log_message "Reboot scheduled for 2 AM tomorrow"
    fi
}

# Function to update system packages
update_system_packages() {
    log_message "Starting system package updates"

    # Update package lists
    apt update

    # Get list of upgradable packages
    UPGRADABLE=$(apt list --upgradable 2>/dev/null | grep -v "WARNING" | wc -l)
    log_message "Found $UPGRADABLE upgradable packages"

    if [ "$UPGRADABLE" -gt 0 ]; then
        # Apply security updates first
        apt upgrade -y --only-upgrade $(apt-get --just-print upgrade 2>&1 | perl -ne '
if (/Inst\s([\w,\-,\d,\.,~,:,\+]+)\s\[([\w,\-,\d,\.,~,:,\+]+)\]\s\(([\w,\-,\d,\.,~,:,\
+]+)\)? /i) {print "$1 "}')

        log_message "System packages updated"

        # Clean up
        apt autoremove -y
        apt autoclean

        log_message "Package cleanup completed"
    else
        log_message "No package updates available"
    fi
}

# Function to backup before updates
backup_before_update() {
    log_message "Creating pre-update backup"

    BACKUP_DIR="/opt/backups/pre-update-$(date +%Y%m%d)"
    mkdir -p "$BACKUP_DIR"

    # Backup application
    tar -czf "$BACKUP_DIR/application.tar.gz" /opt/auto_call_system/

    # Backup configuration
    tar -czf "$BACKUP_DIR/configs.tar.gz" /etc/nginx/ /etc/systemd/system/auto-call-
```

```
system.service

    log_message "Backup completed to $BACKUP_DIR"
}

# Main update process
main() {
    log_message "Starting automated system update process"

    # Create backup
    backup_before_update

    # Update system packages
    update_system_packages

    # Check if reboot is needed
    check_reboot_required

    # Test system after updates
    if systemctl is-active --quiet auto-call-system; then
        log_message "✅ Application service running after updates"
    else
        log_message "❌ Application service not running, attempting restart"
        systemctl restart auto-call-system

        if systemctl is-active --quiet auto-call-system; then
            log_message "✅ Application service restarted successfully"
        else
            log_message "❌ Failed to restart application service - manual
intervention required"
        fi
    fi

    log_message "System update process completed"
}

# Run main function
main
```

Security Maintenance

1. SSL Certificate Management

Automated Certificate Renewal

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

LOG_FILE="/var/log/auto-call-system/ssl_maintenance.log"
DOMAIN=$(hostname -f)

log_message() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" | tee -a $LOG_FILE
}

# Check certificate expiration
check_certificate_expiration() {
    local cert_file="/etc/letsencrypt/live/$DOMAIN/cert.pem"

    if [ ! -f "$cert_file" ]; then
        log_message "❌ Certificate file not found: $cert_file"
        return 1
    fi

    local expiry_date=$(openssl x509 -enddate -noout -in "$cert_file" | cut -d= -f2)
    local expiry_timestamp=$(date -d "$expiry_date" +%s)
    local current_timestamp=$(date +%s)
    local days_until_expiry=$(( (expiry_timestamp - current_timestamp) / 86400 ))

    log_message "Certificate expires in $days_until_expiry days"

    if [ "$days_until_expiry" -lt 30 ]; then
        log_message "⚠️  Certificate expiring soon, triggering renewal"
        return 0
    fi

    return 1
}

# Renew certificate
renew_certificate() {
    log_message "Starting certificate renewal process"

    # Stop nginx temporarily for renewal
    systemctl stop nginx

    # Renew certificate
    certbot renew --standalone --non-interactive

    if [ $? -eq 0 ]; then
        log_message "✅ Certificate renewal successful"

        # Restart services
        systemctl start nginx
        systemctl reload nginx

        log_message "✅ Services restarted"

        # Test HTTPS
        if curl -sf "https://$DOMAIN/health" > /dev/null; then
            log_message "✅ HTTPS health check passed"
        else
            log_message "❌ HTTPS health check failed"
        fi

    else
        log_message "❌ Certificate renewal failed"
```

```
        # Start nginx even if renewal failed
        systemctl start nginx

        return 1
    fi
}

# Main certificate management
main() {
    log_message "Starting SSL certificate maintenance"

    if check_certificate_expiration; then
        renew_certificate
    else
        log_message "Certificate renewal not needed"
    fi

    log_message "SSL certificate maintenance completed"
}

# Schedule this to run daily
main
```

2. Security Hardening Maintenance

Regular Security Hardening Checks

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

LOG_FILE="/var/log/auto-call-system/security_hardening.log"

log_message() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" | tee -a $LOG_FILE
}

# Check and harden SSH configuration
harden_ssh() {
    log_message "Checking SSH configuration"

    SSH_CONFIG="/etc/ssh/sshd_config"
    CHANGES_MADE=0

    # Disable root login
    if ! grep -q "^PermitRootLogin no" "$SSH_CONFIG"; then
        echo "PermitRootLogin no" >> "$SSH_CONFIG"
        log_message "Disabled root login"
        CHANGES_MADE=1
    fi

    # Disable password authentication (if key-based auth is set up)
    if [ -f "/home/ubuntu/.ssh/authorized_keys" ] && ! grep -q "^PasswordAuthentica-
tion no" "$SSH_CONFIG"; then
        echo "PasswordAuthentication no" >> "$SSH_CONFIG"
        log_message "Disabled password authentication"
        CHANGES_MADE=1
    fi

    # Set maximum authentication attempts
    if ! grep -q "^MaxAuthTries" "$SSH_CONFIG"; then
        echo "MaxAuthTries 3" >> "$SSH_CONFIG"
        log_message "Set maximum auth tries to 3"
        CHANGES_MADE=1
    fi

    if [ "$CHANGES_MADE" -eq 1 ]; then
        systemctl reload sshd
        log_message "SSH configuration reloaded"
    fi
}

# Check firewall configuration
check_firewall() {
    log_message "Checking firewall configuration"

    if ! ufw status | grep -q "Status: active"; then
        log_message "⚠️  UFW firewall is not active"

        # Configure basic firewall rules
        ufw default deny incoming
        ufw default allow outgoing
        ufw allow ssh
        ufw allow 80/tcp
        ufw allow 443/tcp
        ufw --force enable

        log_message "✅ UFW firewall configured and enabled"
    else
        log_message "✅ UFW firewall is active"
```

```
    fi
}

# Check for rootkits and malware
security_scan() {
    log_message "Running security scan"

    # Install security tools if not present
    if ! command -v rkhunter &> /dev/null; then
        apt update
        apt install -y rkhunter chkrootkit
        log_message "Installed security scanning tools"
    fi

    # Update rkhunter database
    rkhunter --update --quiet

    # Run rootkit scan
    rkhunter --check --skip-keypress --report-warnings-only > /tmp/
rkhunter_results.txt

    if [ -s /tmp/rkhunter_results.txt ]; then
        log_message "⚠️  Rootkit scan found warnings:"
        cat /tmp/rkhunter_results.txt >> $LOG_FILE
    else
        log_message "✅ Rootkit scan completed - no issues found"
    fi

    # Clean up
    rm -f /tmp/rkhunter_results.txt
}

# Main security hardening function
main() {
    log_message "Starting security hardening check"

    harden_ssh
    check_firewall
    security_scan

    log_message "Security hardening check completed"
}

main
```

Performance Optimization

1. Database Optimization (if using database)

Database Maintenance Script

```
#!/usr/bin/env python3
"""Database optimization and maintenance"""

import sqlite3
import os
from datetime import datetime, timedelta

class DatabaseMaintenance:

    def __init__(self, db_path="/var/lib/auto-call-system/metrics.db"):
        self.db_path = db_path

    def optimize_database(self):
        """Optimize database performance"""
        if not os.path.exists(self.db_path):
            print("Database file not found")
            return

        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        print("Starting database optimization...")

        # Analyze table statistics
        cursor.execute("ANALYZE")

        # Rebuild indexes
        cursor.execute("REINDEX")

        # Vacuum database to reclaim space
        cursor.execute("VACUUM")

        # Get database size
        size_bytes = os.path.getsize(self.db_path)
        size_mb = size_bytes / 1024 / 1024

        print(f"Database optimized. Size: {size_mb:.2f} MB")

        conn.close()

    def cleanup_old_records(self, retention_days=90):
        """Remove old records beyond retention period"""
        if not os.path.exists(self.db_path):
            return

        cutoff_date = (datetime.now() - timedelta(days=retention_days)).date()

        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        # Count records to be deleted
        cursor.execute("SELECT COUNT(*) FROM daily_metrics WHERE date < ?",
(cutoff_date.isoformat(),))
        count_to_delete = cursor.fetchone()[0]

        if count_to_delete > 0:
            cursor.execute("DELETE FROM daily_metrics WHERE date < ?", (cutoff_date.is
oformat(),))
            conn.commit()
            print(f"Deleted {count_to_delete} old records")
        else:
            print("No old records to delete")
```

```
        conn.close()

if __name__ == "__main__":
    maintenance = DatabaseMaintenance()
    maintenance.cleanup_old_records()
    maintenance.optimize_database()
```

2. Application Performance Tuning

Performance Optimization Script

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

LOG_FILE="/var/log/auto-call-system/performance_optimization.log"

log_message() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" | tee -a $LOG_FILE
}

# Optimize system parameters
optimize_system_parameters() {
    log_message "Optimizing system parameters"

    # Increase file descriptor limits
    echo "fs.file-max = 65536" >> /etc/sysctl.conf
    echo "www-data soft nofile 65536" >> /etc/security/limits.conf
    echo "www-data hard nofile 65536" >> /etc/security/limits.conf

    # Optimize network parameters
    echo "net.core.somaxconn = 1024" >> /etc/sysctl.conf
    echo "net.ipv4.tcp_max_syn_backlog = 1024" >> /etc/sysctl.conf

    # Apply changes
    sysctl -p

    log_message "System parameters optimized"
}

# Configure Nginx for better performance
optimize_nginx() {
    log_message "Optimizing Nginx configuration"

    # Backup current configuration
    cp /etc/nginx/nginx.conf /etc/nginx/nginx.conf.backup

    # Apply performance optimizations
    cat >> /etc/nginx/nginx.conf << 'EOF'

# Performance optimizations
worker_processes auto;
worker_connections 1024;

# Enable gzip compression
gzip on;
gzip_vary on;
gzip_min_length 1024;
gzip_types text/plain text/css application/json application/javascript text/xml
application/xml application/xml+rss text/javascript;

# Buffer settings
client_body_buffer_size 128k;
client_max_body_size 10m;
client_header_buffer_size 1k;
large_client_header_buffers 4 4k;
output_buffers 1 32k;
postpone_output 1460;

# Timeout settings
client_body_timeout 12;
client_header_timeout 12;
keepalive_timeout 15;
send_timeout 10;
```

```
EOF

    # Test configuration
    if nginx -t; then
        systemctl reload nginx
        log_message "✅ Nginx configuration optimized and reloaded"
    else
        log_message "❌ Nginx configuration test failed, reverting"
        cp /etc/nginx/nginx.conf.backup /etc/nginx/nginx.conf
    fi
}

# Monitor and optimize Python application
optimize_python_app() {
    log_message "Optimizing Python application"

    # Install Python profiling tools if needed
    cd /opt/auto_call_system
    source venv/bin/activate

    pip install memory_profiler psutil

    # Create application monitoring script
    cat > monitor_app.py << 'EOF'
#!/usr/bin/env python3
"""Monitor application performance"""

import psutil
import os
import time

def monitor_application():
    # Find application process
    for proc in psutil.process_iter(['pid', 'name', 'cmdline']):
        if 'app.py' in ' '.join(proc.info['cmdline']):
            app_process = psutil.Process(proc.info['pid'])

            # Get memory and CPU usage
            memory_usage = app_process.memory_info().rss / 1024 / 1024  # MB
            cpu_percent = app_process.cpu_percent()

            print(f"Application PID: {proc.info['pid']}")
            print(f"Memory usage: {memory_usage:.2f} MB")
            print(f"CPU usage: {cpu_percent:.1f}%")

            return {
                'pid': proc.info['pid'],
                'memory_mb': memory_usage,
                'cpu_percent': cpu_percent
            }

    return None

if __name__ == "__main__":
    stats = monitor_application()
    if stats:
        print("Application performance monitoring completed")
    else:
        print("Application process not found")
EOF

    python3 monitor_app.py
```

```
    log_message "Application monitoring completed"
}

# Main optimization function
main() {
    log_message "Starting performance optimization"

    optimize_system_parameters
    optimize_nginx
    optimize_python_app

    log_message "Performance optimization completed"
}

main
```

Data Management and Backups

1. Automated Backup System

Comprehensive Backup Script

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

BACKUP_BASE_DIR="/opt/backups"
TIMESTAMP=$(date +%Y%m%d_%H%M%S)
BACKUP_DIR="$BACKUP_BASE_DIR/full_backup_$TIMESTAMP"
RETENTION_DAYS=30
LOG_FILE="/var/log/auto-call-system/backup.log"

log_message() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" | tee -a $LOG_FILE
}

# Create backup directory
mkdir -p "$BACKUP_DIR"

# Backup application files
backup_application() {
    log_message "Backing up application files"

    tar -czf "$BACKUP_DIR/application.tar.gz" \
        --exclude="venv" \
        --exclude="__pycache__" \
        --exclude="*.pyc" \
        --exclude=".git" \
        /opt/auto_call_system/

    if [ $? -eq 0 ]; then
        log_message "✅ Application backup completed"
    else
        log_message "❌ Application backup failed"
        return 1
    fi
}

# Backup configuration files
backup_configurations() {
    log_message "Backing up configuration files"

    CONFIG_FILES=(
        "/etc/nginx/sites-available/auto-call-system"
        "/etc/systemd/system/auto-call-system.service"
        "/etc/letsencrypt/live/$(hostname -f)/"
    )

    for config in "${CONFIG_FILES[@]}"; do
        if [ -e "$config" ]; then
            cp -r "$config" "$BACKUP_DIR/"
            log_message "Backed up: $config"
        fi
    done
}

# Backup database files
backup_database() {
    log_message "Backing up database files"

    DB_DIR="/var/lib/auto-call-system"
    if [ -d "$DB_DIR" ]; then
        tar -czf "$BACKUP_DIR/database.tar.gz" "$DB_DIR/"
        log_message "✅ Database backup completed"
    else
```

```
        log_message "ℹ️  No database directory found"
    fi
}

# Backup logs
backup_logs() {
    log_message "Backing up recent logs"

    # Only backup last 7 days of logs to save space
    find /var/log/auto-call-system/ -type f -mtime -7 -exec tar -czf "$BACKUP_DIR/
logs.tar.gz" {} +

    if [ $? -eq 0 ]; then
        log_message "✅ Log backup completed"
    else
        log_message "ℹ️  No recent logs to backup"
    fi
}

# Cleanup old backups
cleanup_old_backups() {
    log_message "Cleaning up old backups"

    find "$BACKUP_BASE_DIR" -name "full_backup_*" -type d -mtime +$RETENTION_DAYS -
exec rm -rf {} + 2>/dev/null

    REMAINING_BACKUPS=$(find "$BACKUP_BASE_DIR" -name "full_backup_*" -type d | wc -l)
    log_message "Cleanup completed. $REMAINING_BACKUPS backup sets remaining"
}

# Generate backup report
generate_backup_report() {
    BACKUP_SIZE=$(du -sh "$BACKUP_DIR" | cut -f1)

    cat > "$BACKUP_DIR/backup_report.txt" << EOF
Backup Report
=============
Date: $(date)
Location: $BACKUP_DIR
Size: $BACKUP_SIZE
Retention: $RETENTION_DAYS days

Contents:
- Application files: $([ -f "$BACKUP_DIR/application.tar.gz" ] && echo "✅" || echo
"❌")
- Configuration files: $([ "$(ls -A "$BACKUP_DIR"/*.service "$BACKUP_DIR"/*nginx* 2>/
dev/null)" ] && echo "✅" || echo "❌")
- Database: $([ -f "$BACKUP_DIR/database.tar.gz" ] && echo "✅" || echo "❌")
- Logs: $([ -f "$BACKUP_DIR/logs.tar.gz" ] && echo "✅" || echo "❌")
EOF

    log_message "Backup completed - Size: $BACKUP_SIZE"
}

# Main backup function
main() {
    log_message "Starting full system backup"

    backup_application
    backup_configurations
    backup_database
    backup_logs
    generate_backup_report
```

```
    cleanup_old_backups

    log_message "Full system backup completed"
}

# Run backup
main
```

2. Backup Verification and Restore Testing

Backup Verification Script

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

BACKUP_DIR="$1"
LOG_FILE="/var/log/auto-call-system/backup_verification.log"

if [ -z "$BACKUP_DIR" ]; then
    echo "Usage: $0 <backup_directory>"
    exit 1
fi

log_message() {
    echo "$(date '+%Y-%m-%d %H:%M:%S') - $1" | tee -a $LOG_FILE
}

# Verify backup integrity
verify_backup_integrity() {
    log_message "Verifying backup integrity for: $BACKUP_DIR"

    BACKUP_FILES=(
        "application.tar.gz"
        "database.tar.gz"
        "logs.tar.gz"
    )

    VERIFIED=0
    TOTAL=0

    for file in "${BACKUP_FILES[@]}"; do
        TOTAL=$((TOTAL + 1))
        FILEPATH="$BACKUP_DIR/$file"

        if [ -f "$FILEPATH" ]; then
            # Test archive integrity
            if tar -tzf "$FILEPATH" >/dev/null 2>&1; then
                log_message "✅ $file: Archive integrity verified"
                VERIFIED=$((VERIFIED + 1))
            else
                log_message "❌ $file: Archive corrupted"
            fi
        else
            log_message "⚠️  $file: File missing"
        fi
    done

    log_message "Verification completed: $VERIFIED/$TOTAL files verified"

    return $([ $VERIFIED -eq $TOTAL ] && echo 0 || echo 1)
}

# Test restore procedure (dry run)
test_restore_procedure() {
    log_message "Testing restore procedure (dry run)"

    TEST_RESTORE_DIR="/tmp/restore_test_$(date +%s)"
    mkdir -p "$TEST_RESTORE_DIR"

    # Test extracting application backup
    if tar -tzf "$BACKUP_DIR/application.tar.gz" | head -10; then
        log_message "✅ Application backup can be extracted"
    else
        log_message "❌ Application backup extraction failed"
```

```
    fi

    # Test database backup
    if [ -f "$BACKUP_DIR/database.tar.gz" ]; then
        if tar -tzf "$BACKUP_DIR/database.tar.gz" | head -5; then
            log_message "✅ Database backup can be extracted"
        else
            log_message "❌ Database backup extraction failed"
        fi
    fi

    # Cleanup test directory
    rm -rf "$TEST_RESTORE_DIR"

    log_message "Restore procedure test completed"
}

# Main verification function
main() {
    log_message "Starting backup verification"

    if verify_backup_integrity; then
        test_restore_procedure
        log_message "✅ Backup verification passed"
        exit 0
    else
        log_message "❌ Backup verification failed"
        exit 1
    fi
}

main
```

Maintenance Schedules

1. Automated Maintenance Scheduling

Master Maintenance Scheduler

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

SCRIPT_DIR="/opt/scripts"
CRON_FILE="/etc/cron.d/auto-call-system-maintenance"

# Create scripts directory
mkdir -p "$SCRIPT_DIR"

# Create master cron file
cat > "$CRON_FILE" << 'EOF'
# Automated Phone Answering System Maintenance Schedule
# Generated automatically - do not edit manually

SHELL=/bin/bash
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# Daily tasks - 6 AM
0 6 * * * root /opt/scripts/daily_health_check.sh
0 6 * * * root /opt/scripts/daily_metrics_collection.py

# Weekly tasks - Sunday 2 AM
0 2 * * 0 root /opt/scripts/weekly_updates.sh
30 2 * * 0 root /opt/scripts/weekly_log_maintenance.py

# Monthly tasks - 1st of month, 3 AM
0 3 1 * * root /opt/scripts/monthly_capacity_analysis.py
0 4 1 * * root /opt/scripts/monthly_security_audit.sh

# Quarterly tasks - 1st of Jan/Apr/Jul/Oct, 5 AM
0 5 1 1,4,7,10 * root /opt/scripts/quarterly_system_review.sh

# Backup tasks - Daily 1 AM
0 1 * * * root /opt/scripts/automated_backup.sh

# SSL certificate check - Daily 7 AM
0 7 * * * root /opt/scripts/ssl_certificate_management.sh

# Performance monitoring - Every 6 hours
0 */6 * * * root /opt/scripts/performance_monitoring.sh

EOF

echo "Maintenance schedule configured in $CRON_FILE"
echo "Individual scripts should be placed in $SCRIPT_DIR"
```

2. Maintenance Calendar and Notifications

Maintenance Tracking System

```
#!/usr/bin/env python3
"""Maintenance tracking and notification system"""

import json
import smtplib
from email.mime.text import MimeText
from email.mime.multipart import MimeMultipart
from datetime import datetime, timedelta
import os

class MaintenanceTracker:

    def __init__(self):
        self.tracking_file = "/var/lib/auto-call-system/maintenance_tracking.json"
        self.admin_email = os.getenv('ADMIN_EMAIL', 'admin@yourcompany.com')

        self.maintenance_tasks = {
            'system_updates': {'frequency': 'weekly', 'last_run': None},
            'security_audit': {'frequency': 'monthly', 'last_run': None},
            'capacity_planning': {'frequency': 'monthly', 'last_run': None},
            'backup_verification': {'frequency': 'weekly', 'last_run': None},
            'ssl_renewal': {'frequency': 'quarterly', 'last_run': None},
            'performance_optimization': {'frequency': 'quarterly', 'last_run': None}
        }

    def load_tracking_data(self):
        """Load maintenance tracking data"""
        if os.path.exists(self.tracking_file):
            with open(self.tracking_file, 'r') as f:
                data = json.load(f)
                self.maintenance_tasks.update(data)

    def save_tracking_data(self):
        """Save maintenance tracking data"""
        os.makedirs(os.path.dirname(self.tracking_file), exist_ok=True)
        with open(self.tracking_file, 'w') as f:
            json.dump(self.maintenance_tasks, f, indent=2)

    def record_task_completion(self, task_name):
        """Record completion of a maintenance task"""
        if task_name in self.maintenance_tasks:
            self.maintenance_tasks[task_name]['last_run'] = datetime.now().isoformat()
            self.save_tracking_data()
            print(f"Recorded completion of {task_name}")

    def check_overdue_tasks(self):
        """Check for overdue maintenance tasks"""
        overdue_tasks = []
        now = datetime.now()

        frequency_days = {
            'daily': 1,
            'weekly': 7,
            'monthly': 30,
            'quarterly': 90
        }

        for task_name, task_info in self.maintenance_tasks.items():
            frequency = task_info['frequency']
            last_run_str = task_info['last_run']

            if last_run_str is None:
```

```
                overdue_tasks.append({
                    'task': task_name,
                    'days_overdue': 'Never run',
                    'frequency': frequency
                })
            else:
                last_run = datetime.fromisoformat(last_run_str)
                days_since_last_run = (now - last_run).days
                expected_interval = frequency_days[frequency]

                if days_since_last_run > expected_interval:
                    overdue_tasks.append({
                        'task': task_name,
                        'days_overdue': days_since_last_run - expected_interval,
                        'frequency': frequency,
                        'last_run': last_run_str
                    })

        return overdue_tasks

    def send_maintenance_reminder(self, overdue_tasks):
        """Send email reminder for overdue tasks"""
        if not overdue_tasks:
            return

        subject = f"Maintenance Tasks Overdue - {datetime.now().strftime('%Y-%m-%d')}"

        body = "The following maintenance tasks are overdue:\n\n"

        for task in overdue_tasks:
            body += f"• {task['task'].replace('_', ' ').title()}\n"
            body += f"  Frequency: {task['frequency']}\n"

            if task['days_overdue'] == 'Never run':
                body += f"  Status: Never been run\n"
            else:
                body += f"  Days overdue: {task['days_overdue']}\n"
                body += f"  Last run: {task['last_run']}\n"

            body += "\n"

        body += "Please schedule these maintenance tasks as soon as possible.\n"
        body += f"\nGenerated by Auto Call System Maintenance Tracker\n"
        body += f"Server: {os.uname().nodename}\n"

        # Send email (simplified - implement with your email settings)
        print(f"Maintenance reminder email would be sent to {self.admin_email}")
        print(f"Subject: {subject}")
        print(f"Body:\n{body}")

    def generate_maintenance_report(self):
        """Generate maintenance status report"""
        report_file = f"/var/log/auto-call-system/maintenance_report_{datetime.now().s
trftime('%Y%m%d')}.txt"

        with open(report_file, 'w') as f:
            f.write(f"Maintenance Status Report\n")
            f.write(f"Generated: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n")
            f.write("=" * 50 + "\n\n")

            for task_name, task_info in self.maintenance_tasks.items():
                f.write(f"Task: {task_name.replace('_', ' ').title()}\n")
                f.write(f"Frequency: {task_info['frequency']}\n")
```

```
                if task_info['last_run']:
                    last_run = datetime.fromisoformat(task_info['last_run'])
                    days_ago = (datetime.now() - last_run).days
                    f.write(f"Last run: {task_info['last_run']} ({days_ago} days ago)\
n")
                else:
                    f.write(f"Last run: Never\n")

                f.write("\n")

            # Add overdue tasks section
            overdue_tasks = self.check_overdue_tasks()
            if overdue_tasks:
                f.write("OVERDUE TASKS:\n")
                f.write("-" * 20 + "\n")

                for task in overdue_tasks:
                    f.write(f"• {task['task'].replace('_', ' ').title()}")
                    if task['days_overdue'] != 'Never run':
                        f.write(f" ({task['days_overdue']} days overdue)")
                    f.write("\n")

        print(f"Maintenance report generated: {report_file}")

    def run_maintenance_check(self):
        """Run complete maintenance check"""
        print("Running maintenance tracking check...")

        self.load_tracking_data()
        overdue_tasks = self.check_overdue_tasks()

        if overdue_tasks:
            print(f"Found {len(overdue_tasks)} overdue maintenance tasks")
            self.send_maintenance_reminder(overdue_tasks)
        else:
            print("All maintenance tasks are up to date")

        self.generate_maintenance_report()

if __name__ == "__main__":
    import sys

    tracker = MaintenanceTracker()

    if len(sys.argv) > 2 and sys.argv[1] == "record":
        # Record task completion
        task_name = sys.argv[2]
        tracker.load_tracking_data()
        tracker.record_task_completion(task_name)
    else:
        # Run maintenance check
        tracker.run_maintenance_check()
```

Maintenance Guide Complete! 🛠️

This comprehensive maintenance guide provides you with all the tools and procedures needed to keep your automated phone answering system running optimally. Regular maintenance following these guidelines will ensure:

- **High Availability**: Minimal downtime and reliable service
- **Security**: Protection against vulnerabilities and threats
- **Performance**: Optimal response times and resource utilization
- **Scalability**: Ready for growth and increased demand
- **Data Integrity**: Secure backups and recovery procedures

Final Setup Steps

1. **Schedule all maintenance tasks** using the provided cron configurations
2. **Set up monitoring and alerting** for critical issues
3. **Test backup and restore procedures** regularly
4. **Review and adjust** maintenance schedules based on your usage patterns
5. **Document any customizations** you make to the maintenance procedures

Your automated phone answering system is now fully deployed with comprehensive maintenance procedures! 🎉