

SmartParent Phase 2: Steps 4-8 Implementation Guide

Alert System Integration & Performance Optimization

Prerequisites

✔ Completed Steps 1-3 (AI-Stack Setup):

- Ollama service running with Phi-3-mini model
- Domain classification system functional
- Database schema updated with AI tables
- All Step 3 tests passing

Step 4: Alert System Integration

4.1 Create Alert System Module

File: `src/alert_system.py`

```
python
```

```
#!/usr/bin/env python3
```

```
import sqlite3
import smtplib
import logging
import json
from datetime import datetime, timedelta
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from typing import List, Dict, Optional
import requests
```

```
class AlertSystem:
```

```
    def __init__(self, db_path: str = '../data/smartguard.db'):
        self.db_path = db_path
        self.logger = logging.getLogger(__name__)
        self.notification_methods = {
            'email': self._send_email_alert,
            'webhook': self._send_webhook_alert,
            'log': self._send_log_alert
        }
```

```
    def check_alert_rules(self, client_ip: str, domain: str, category: str, risk_level: str) -> List[Dict]:
```

```
        """Check if any alert rules are triggered"""
```

```
        alerts = []
```

```
        try:
```

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

```
            cursor = conn.cursor()
```

```
            # Get active alert rules for this category/risk level
```

```
            cursor.execute("""
```

```
                SELECT id, rule_name, category, risk_level, time_threshold,
```

```
                    request_threshold, notification_method
```

```
                FROM alert_rules
```

```
                WHERE (category = ? OR category = 'all')
```

```
                AND (risk_level = ? OR risk_level = 'all')
```

```
                AND enabled = 1
```

```
            """, (category, risk_level))
```

```
            rules = cursor.fetchall()
```

```
            for rule in rules:
```

```
                rule_id, rule_name, cat, risk, time_threshold, request_threshold, notification_method = rule
```

```
            # Check request count in time window
```

```
            cursor.execute("""
```

```
                SELECT COUNT(*) FROM dns_requests dr
```

```

JOIN domain_classifications dc ON dr.domain = dc.domain
WHERE dr.client_ip = ?
AND (dc.category = ? OR ? = 'all')
AND (dc.risk_level = ? OR ? = 'all')
AND dr.timestamp > datetime('now', '-{0} seconds')
'''format(time_threshold), (client_ip, cat, cat, risk, risk))

```

```
request_count = cursor.fetchone()[0]
```

```
if request_count >= request_threshold:
```

```
    # Check if this alert was already fired recently (prevent spam)
```

```

    cursor.execute(''
        SELECT COUNT(*) FROM alerts
        WHERE client_ip = ? AND alert_type = ?
        AND created_at > datetime('now', '-{0} seconds')
        AND acknowledged = 0
    ''format(time_threshold * 2), (client_ip, rule_name))

```

```
recent_alerts = cursor.fetchone()[0]
```

```
if recent_alerts == 0: # No recent unacknowledged alerts
```

```

    alert = {
        "rule_id": rule_id,
        "rule_name": rule_name,
        "client_ip": client_ip,
        "domain": domain,
        "category": category,
        "severity": risk,
        "message": f"Device {client_ip} accessed {request_count} {category} sites in {time_threshold} //60
        "request_count": request_count,
        "time_window": time_threshold,
        "notification_method": notification_method or 'log'
    }
    alerts.append(alert)

```

```
conn.close()
```

```
except Exception as e:
```

```
    self.logger.error(f"Alert check error: {e}")
```

```
return alerts
```

```
def create_alert(self, alert_data: Dict) -> int:
```

```
    """Create alert in database and return alert ID"""
```

```
    try:
```

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

```
        cursor = conn.cursor()
```

```

cursor.execute("""
    INSERT INTO alerts
    (alert_type, severity, client_ip, domain, category, message, metadata)
    VALUES (?, ?, ?, ?, ?, ?, ?)
""", (
    alert_data["rule_name"],
    alert_data["severity"],
    alert_data["client_ip"],
    alert_data["domain"],
    alert_data["category"],
    alert_data["message"],
    json.dumps({
        "request_count": alert_data["request_count"],
        "time_window": alert_data["time_window"]
    })
))


```

```

alert_id = cursor.lastrowid
conn.commit()
conn.close()

```

```

self.logger.info(f"

```

```

except Exception as e:
    self.logger.error(f"Alert creation error: {e}")
    return None

```

```

def send_notification(self, alert_data: Dict):
    """Send notification using configured method"""
    method = alert_data.get('notification_method', 'log')

    if method in self.notification_methods:
        self.notification_methods[method](alert_data)
    else:
        self.logger.warning(f"Unknown notification method: {method}")
        self._send_log_alert(alert_data)

```

```

def _send_email_alert(self, alert_data: Dict):
    """Send email alert"""
    try:
        # Get email configuration from database
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute("""

```

```
SELECT key, value FROM configuration
WHERE key IN ('smtp_server', 'smtp_port', 'email_user', 'email_password', 'alert_recipients')
''')
```

```
config = dict(cursor.fetchall())
conn.close()
```

```
if not all(k in config for k in ['smtp_server', 'email_user', 'alert_recipients']):
    self.logger.warning("Email configuration incomplete, skipping email alert")
    return
```

```
recipients = config['alert_recipients'].split(',')
```

```
for recipient in recipients:
    self._send_single_email(alert_data, recipient.strip(), config)
```

```
except Exception as e:
    self.logger.error(f"Email alert error: {e}")
```

```
def _send_single_email(self, alert_data: Dict, recipient: str, config: Dict):
```

```
    """Send single email alert"""
```

```
    try:
```

```
        msg = MIMEMultipart()
        msg['From'] = config['email_user']
        msg['To'] = recipient
        msg['Subject'] = f"🚨 SmartParent Alert: {alert_data['rule_name']}"
```

```
        severity_emoji = {"low": "🔍", "medium": "⚠️", "high": "🚨"}
        emoji = severity_emoji.get(alert_data['severity'], "📧")
```

```
        body = f"""
```

```
{emoji} SmartParent Alert
```

Alert Details:

- Rule: {alert_data['rule_name']}
- Device: {alert_data['client_ip']}
- Domain: {alert_data['domain']}
- Category: {alert_data['category'].title()}
- Severity: {alert_data['severity'].upper()}
- Activity: {alert_data['message']}
- Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}

Quick Actions:

- View Dashboard: <http://192.168.1.100:5000>
- Block Device: [http://192.168.1.100:5000/block/{alert_data\['client_ip'\]}](http://192.168.1.100:5000/block/{alert_data['client_ip']})
- Acknowledge Alert: [http://192.168.1.100:5000/acknowledge/{alert_data.get\('alert_id', ''\)}](http://192.168.1.100:5000/acknowledge/{alert_data.get('alert_id', '')})

This is an automated message from your SmartParent monitoring system.

```
"""
```

```
msg.attach(MIMEText(body, 'plain'))
```

```
server = smtplib.SMTP(config['smtp_server'], int(config.get('smtp_port', 587)))
```

```
server.starttls()
```

```
server.login(config['email_user'], config['email_password'])
```

```
server.send_message(msg)
```

```
server.quit()
```

```
self.logger.info(f"✉ Email alert sent to {recipient}")
```

```
except Exception as e:
```

```
self.logger.error(f"Email send error: {e}")
```

```
def _send_webhook_alert(self, alert_data: Dict):
```

```
    """Send webhook alert"""
```

```
    try:
```

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

```
        cursor = conn.cursor()
```

```
        cursor.execute("SELECT value FROM configuration WHERE key = 'webhook_url'")
```

```
        result = cursor.fetchone()
```

```
        conn.close()
```

```
    if not result:
```

```
        self.logger.warning("Webhook URL not configured")
```

```
        return
```

```
    webhook_url = result[0]
```

```
    payload = {
```

```
        "alert_type": alert_data["rule_name"],
```

```
        "severity": alert_data["severity"],
```

```
        "client_ip": alert_data["client_ip"],
```

```
        "domain": alert_data["domain"],
```

```
        "category": alert_data["category"],
```

```
        "message": alert_data["message"],
```

```
        "timestamp": datetime.now().isoformat(),
```

```
        "dashboard_url": "http://192.168.1.100:5000"
```

```
    }
```

```
    response = requests.post(webhook_url, json=payload, timeout=10)
```

```
    response.raise_for_status()
```

```
    self.logger.info(f"📡 Webhook alert sent to {webhook_url}")
```

```
except Exception as e:
    self.logger.error(f"Webhook alert error: {e}")
```

```
def _send_log_alert(self, alert_data: Dict):
    """Send log-only alert"""
    severity_emoji = {"low": "🔵", "medium": "⚠️", "high": "🔴"}
    emoji = severity_emoji.get(alert_data['severity'], "📡")

    self.logger.warning(
        f"{emoji} ALERT [{alert_data['severity'].upper()}]: "
        f"{alert_data['client_ip']} -> {alert_data['domain']} "
        f"[{alert_data['category'].upper()}] - {alert_data['message']}"
    )
```

```
def acknowledge_alert(self, alert_id: int, acknowledged_by: str = 'system') -> bool:
    """Acknowledge an alert"""
    try:
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute("""
            UPDATE alerts
            SET acknowledged = 1, acknowledged_by = ?, acknowledged_at = CURRENT_TIMESTAMP
            WHERE id = ?
            """, (acknowledged_by, alert_id))

        conn.commit()
        conn.close()

        self.logger.info(f"Alert {alert_id} acknowledged by {acknowledged_by}")
        return True
```

```
except Exception as e:
    self.logger.error(f"Alert acknowledgment error: {e}")
    return False
```

```
def get_active_alerts(self, limit: int = 50) -> List[Dict]:
    """Get active (unacknowledged) alerts"""
    try:
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute("""
            SELECT id, alert_type, severity, client_ip, domain, category,
                   message, created_at, metadata
            FROM alerts
```

```
WHERE acknowledged = 0
ORDER BY created_at DESC
LIMIT ?
'', (limit,))
```

```
alerts = []
for row in cursor.fetchall():
    alert = {
        'id': row[0],
        'alert_type': row[1],
        'severity': row[2],
        'client_ip': row[3],
        'domain': row[4],
        'category': row[5],
        'message': row[6],
        'created_at': row[7],
        'metadata': json.loads(row[8]) if row[8] else {}
    }
    alerts.append(alert)
```

```
conn.close()
return alerts
```

```
except Exception as e:
    self.logger.error(f"Get alerts error: {e}")
    return []
```

```
# Test the alert system
```

```
if __name__ == '__main__':
    alert_system = AlertSystem()
```

```
# Test alert creation
```

```
test_alert = {
    "rule_name": "Test Alert",
    "client_ip": "192.168.1.50",
    "domain": "example.com",
    "category": "social_media",
    "severity": "medium",
    "message": "Test alert message",
    "request_count": 5,
    "time_window": 300,
    "notification_method": "log"
}
```

```
alert_id = alert_system.create_alert(test_alert)
print(f"Created test alert with ID: {alert_id}")
```



```
# Test getting active alerts
```

```
active_alerts = alert_system.get_active_alerts()
```

```
print(f"Active alerts: {len(active_alerts)}")
```

4.2 Update Database Schema for Enhanced Alerts

File: `src/update_database_step4.py`

```
python
```

```
#!/usr/bin/env python3
```

```
import sqlite3
```

```
from datetime import datetime
```

```
def update_database_step4():
```

```
    """Update database schema for enhanced alerts"""
```

```
    conn = sqlite3.connect('../data/smartguard.db')
```

```
    cursor = conn.cursor()
```

```
    print("🔄 Updating database schema for Step 4...")
```

```
    # Add new columns to alerts table
```

```
    try:
```

```
        cursor.execute('ALTER TABLE alerts ADD COLUMN metadata TEXT')
```

```
        print("✅ Added metadata column to alerts table")
```

```
    except sqlite3.OperationalError:
```

```
        print("ℹ️ Metadata column already exists")
```

```
    try:
```

```
        cursor.execute('ALTER TABLE alerts ADD COLUMN acknowledged_by TEXT')
```

```
        cursor.execute('ALTER TABLE alerts ADD COLUMN acknowledged_at DATETIME')
```

```
        print("✅ Added acknowledgment columns to alerts table")
```

```
    except sqlite3.OperationalError:
```

```
        print("ℹ️ Acknowledgment columns already exist")
```

```
    # Add notification_method to alert_rules
```

```
    try:
```

```
        cursor.execute('ALTER TABLE alert_rules ADD COLUMN notification_method TEXT DEFAULT "log"')
```

```
        print("✅ Added notification_method column to alert_rules table")
```

```
    except sqlite3.OperationalError:
```

```
        print("ℹ️ Notification_method column already exists")
```

```
    # Update existing alert rules with notification methods
```

```
    cursor.execute("""
```

```
        UPDATE alert_rules
```

```
        SET notification_method = CASE
```

```
            WHEN risk_level = 'high' THEN 'email'
```

```
            WHEN risk_level = 'medium' THEN 'webhook'
```

```
            ELSE 'log'
```

```
        END
```

```
        WHERE notification_method IS NULL
```

```
    """)
```

```
    # Add configuration entries for notifications
```

```
    config_entries = [
```

```
        ('smtp_server', '', 'SMTP server for email alerts'),
```

```

('smtp_port', '587', 'SMTP port'),
('email_user', '', 'Email username'),
('email_password', '', 'Email password (use app password)'),
('alert_recipients', '', 'Comma-separated email addresses'),
('webhook_url', '', 'Webhook URL for external integrations'),
('alert_cooldown', '300', 'Minimum seconds between duplicate alerts'),
('max_alerts_per_hour', '10', 'Maximum alerts per hour per device')
]

for key, value, description in config_entries:
    cursor.execute("""
        INSERT OR IGNORE INTO configuration (key, value, description)
        VALUES (?, ?, ?)
    """, (key, value, description))

print(f"✅ Added {len(config_entries)} configuration entries")

# Create alert statistics view
cursor.execute("""
    CREATE VIEW IF NOT EXISTS alert_statistics AS
    SELECT
        DATE(created_at) as alert_date,
        severity,
        category,
        COUNT(*) as alert_count,
        COUNT(CASE WHEN acknowledged = 1 THEN 1 END) as acknowledged_count
    FROM alerts
    GROUP BY DATE(created_at), severity, category
    ORDER BY alert_date DESC
""")
print("✅ Created alert_statistics view")

conn.commit()
conn.close()
print("🎉 Step 4 database updates completed!")

if __name__ == '__main__':
    update_database_step4()

```

4.3 Test Alert System

File: `src/test_step4.py`

python

```
#!/usr/bin/env python3
import sys
import time
import sqlite3
from alert_system import AlertSystem
from domain_classifier import DomainClassifier

def test_alert_system():
    """Comprehensive test of alert system"""
    print("🔪 Testing Alert System (Step 4)")
    print("=" * 50)

    alert_system = AlertSystem()
    classifier = DomainClassifier()

    # Test 1: Alert rule checking
    print("\n1. Testing alert rule checking...")
    test_client = "192.168.1.99"

    # Simulate multiple social media requests
    for i in range(5):
        conn = sqlite3.connect('../data/smartguard.db')
        cursor = conn.cursor()
        cursor.execute("""
            INSERT INTO dns_requests (client_ip, domain, query_type, classification, confidence_score)
            VALUES (?, ?, 'A', 'social_media', 0.85)
            """, (test_client, f"facebook{i}.com"))
        conn.commit()
        conn.close()
        time.sleep(0.1)

    alerts = alert_system.check_alert_rules(test_client, "facebook.com", "social_media", "medium")
    print(f"Generated {len(alerts)} alerts")

    # Test 2: Alert creation
    print("\n2. Testing alert creation...")
    if alerts:
        alert_id = alert_system.create_alert(alerts[0])
        print(f"Created alert with ID: {alert_id}")

    # Test 3: Alert acknowledgment
    print("\n3. Testing alert acknowledgment...")
    success = alert_system.acknowledge_alert(alert_id, "test_user")
    print(f"Acknowledgment {'successful' if success else 'failed'}")

    # Test 4: Get active alerts
```

```

print("\n4. Testing active alerts retrieval...")
active_alerts = alert_system.get_active_alerts()
print(f" Found {len(active_alerts)} active alerts")

# Test 5: Notification system (log only)
print("\n5. Testing notification system...")
test_alert = {
    "rule_name": "Test Notification",
    "client_ip": test_client,
    "domain": "test.com",
    "category": "gaming",
    "severity": "medium",
    "message": "Test notification message",
    "request_count": 3,
    "time_window": 300,
    "notification_method": "log"
}

alert_system.send_notification(test_alert)
print(" Log notification sent")

print("\n🎉 Alert system tests completed!")
print("\nNext steps:")
print("1. Configure email settings in database")
print("2. Test email notifications")
print("3. Set up webhook endpoints (optional)")
print("4. Proceed to Step 5: Enhanced DNS Monitor Integration")

if __name__ == '__main__':
    test_alert_system()

```

Step 5: Enhanced DNS Monitor Integration

5.1 Create Enhanced DNS Monitor

File: `src/enhanced_dns_monitor.py`

python

```
#!/usr/bin/env python3
```

```
import socket
```

```
import struct
```

```
import sqlite3
```

```
import threading
```

```
import time
```

```
import logging
```

```
import signal
```

```
import sys
```

```
from datetime import datetime
```

```
from typing import Optional, Tuple, Dict
```

```
from domain_classifier import DomainClassifier
```

```
from alert_system import AlertSystem
```

```
class EnhancedDNSMonitor:
```

```
    def __init__(self, db_path: str = '../data/smartguard.db'):
```

```
        self.db_path = db_path
```

```
        self.running = False
```

```
        self.socket = None
```

```
        self.classifier = DomainClassifier(db_path)
```

```
        self.alert_system = AlertSystem(db_path)
```

```
    # Performance tracking
```

```
    self.stats = {
```

```
        'total_requests': 0,
```

```
        'classified_requests': 0,
```

```
        'alerts_triggered': 0,
```

```
        'start_time': None
```

```
    }
```

```
    # Setup enhanced logging
```

```
    logging.basicConfig(
```

```
        level=logging.INFO,
```

```
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
```

```
        handlers=[
```

```
            logging.FileHandler('../logs/enhanced_dns_monitor.log'),
```

```
            logging.StreamHandler()
```

```
        ]
```

```
    )
```

```
    self.logger = logging.getLogger(__name__)
```

```
    # Setup signal handlers for graceful shutdown
```

```
    signal.signal(signal.SIGINT, self._signal_handler)
```

```
    signal.signal(signal.SIGTERM, self._signal_handler)
```

```
    def _signal_handler(self, signum, frame):
```

```

"""Handle shutdown signals gracefully"""
self.logger.info(f"Received signal {signum}, shutting down...")
self.stop()
sys.exit(0)

def start(self, interface: str = 'eth0', port: int = 53):
    """Start enhanced DNS monitoring"""
    self.running = True
    self.stats['start_time'] = datetime.now()

    try:
        # Create raw socket for DNS monitoring
        self.socket = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(0x0003))
        self.socket.bind((interface, 0))

        self.logger.info(f"🚀 Enhanced DNS Monitor started on {interface}:{port}")
        self.logger.info(f"🇩🇪 AI Classification: ENABLED")
        self.logger.info(f"🚨 Alert System: ENABLED")

        # Start statistics reporting thread
        stats_thread = threading.Thread(target=self._stats_reporter, daemon=True)
        stats_thread.start()

        while self.running:
            try:
                # Receive packet
                raw_data, addr = self.socket.recvfrom(65535)

                # Process packet in separate thread for better performance
                threading.Thread(
                    target=self._process_packet,
                    args=(raw_data,),
                    daemon=True
                ).start()

            except socket.error as e:
                if self.running:
                    self.logger.error(f"Socket error: {e}")
                    time.sleep(1)

            except Exception as e:
                self.logger.error(f"Monitor start error: {e}")
            finally:
                self.stop()

    def stop(self):
        """Stop DNS monitoring"""

```

```

self.running = False
if self.socket:
    self.socket.close()
    self.socket = None

# Print final statistics
if self.stats['start_time']:
    runtime = datetime.now() - self.stats['start_time']
    self.logger.info(f"📊 Final Statistics:")
    self.logger.info(f"   Runtime: {runtime}")
    self.logger.info(f"   Total Requests: {self.stats['total_requests']}")
    self.logger.info(f"   Classified: {self.stats['classified_requests']}")
    self.logger.info(f"   Alerts: {self.stats['alerts_triggered']}")

self.logger.info(f"🛑 Enhanced DNS Monitor stopped")

def _process_packet(self, raw_data: bytes):
    """Process individual network packet"""
    try:
        # Parse Ethernet header
        eth_header = struct.unpack('!6s6sH', raw_data[:14])
        eth_protocol = socket.ntohs(eth_header[2])

        # Check if it's IP packet
        if eth_protocol != 0x0800:
            return

        # Parse IP header
        ip_header = struct.unpack('!BBHHHBBH4s4s', raw_data[14:34])
        protocol = ip_header[6]
        src_ip = socket.inet_ntoa(ip_header[8])
        dst_ip = socket.inet_ntoa(ip_header[9])

        # Check if it's UDP packet to port 53 (DNS)
        if protocol != 17:
            return

        # Parse UDP header
        udp_header = struct.unpack('!HHHH', raw_data[34:42])
        src_port = udp_header[0]
        dst_port = udp_header[1]

        if dst_port != 53:
            return

        # Parse DNS query
        dns_data = raw_data[42:]

```



```
domain = self._parse_dns_query(dns_data)
```

```
if domain:  
    self.stats['total_requests'] += 1  
    self._handle_dns_request(src_ip, domain)
```

```
except Exception as e:  
    self.logger.debug(f"Packet processing error: {e}")
```

```
def _parse_dns_query(self, dns_data: bytes) -> Optional[str]:
```

```
    """Parse DNS query to extract domain name"""
```

```
    try:
```

```
        if len(dns_data) < 12:  
            return None
```

```
        # Skip DNS header (12 bytes)
```

```
        offset = 12
```

```
        domain_parts = []
```

```
        while offset < len(dns_data):
```

```
            length = dns_data[offset]
```

```
            if length == 0:
```

```
                break
```

```
            if length > 63: # Compression pointer
```

```
                break
```

```
            if offset + 1 + length > len(dns_data):
```

```
                break
```

```
            part = dns_data[offset + 1:offset + 1 + length].decode('utf-8', errors='ignore')
```

```
            domain_parts.append(part)
```

```
            offset += 1 + length
```

```
        if domain_parts:
```

```
            domain = ''.join(domain_parts)
```

```
            # Filter out internal/local domains
```

```
            if self._is_valid_domain(domain):
```

```
                return domain.lower()
```

```
        return None
```

```
except Exception as e:  
    self.logger.debug(f"DNS parsing error: {e}")  
    return None
```

```
def _is_valid_domain(self, domain: str) -> bool:
```

```

"""Check if domain should be processed"""
# Skip local/internal domains
local_patterns = [
    '.local', '.lan', '.home', '.internal',
    'localhost', '192.168.', '10.', '172.',
    'in-addr.arpa', 'ip6.arpa'
]

domain_lower = domain.lower()
return not any(pattern in domain_lower for pattern in local_patterns)

def _handle_dns_request(self, client_ip: str, domain: str):
    """Handle DNS request with classification and alerting"""
    try:
        # Classify domain using AI
        classification = self.classifier.classify_domain(domain)
        self.stats['classified_requests'] += 1

        # Log request to database
        self._log_dns_request(client_ip, domain, classification)

        # Check for alerts
        alerts = self.alert_system.check_alert_rules(
            client_ip,
            domain,
            classification["category"],
            classification["risk_level"]
        )

        # Process any triggered alerts
        for alert in alerts:
            alert_id = self.alert_system.create_alert(alert)
            if alert_id:
                alert['alert_id'] = alert_id
                self.alert_system.send_notification(alert)
                self.stats['alerts_triggered'] += 1

        # Enhanced logging with visual indicators
        self._log_request_with_style(client_ip, domain, classification, len(alerts) > 0)

    except Exception as e:
        self.logger.error(f"Request handling error for {domain}: {e}")

def _log_dns_request(self, client_ip: str, domain: str, classification: Dict):
    """Log DNS request to database"""
    try:
        conn = sqlite3.connect(self.db_path)

```

```
cursor = conn.cursor()
```

```
cursor.execute("""
    INSERT INTO dns_requests
    (client_ip, domain, query_type, classification, confidence_score, metadata)
    VALUES (?, ?, ?, ?, ?, ?)
""", (
    client_ip,
    domain,
    'A', # Default to A record
    classification["category"],
    classification["confidence"],
    f'{{"risk_level": "{classification["risk_level"]}", "color": "{classification["color"]}"}}'
))

conn.commit()
conn.close()
```

```
except Exception as e:
    self.logger.error(f"Database logging error: {e}")
```

```
def _log_request_with_style(self, client_ip: str, domain: str, classification: Dict, has_alert: bool):
```

```
    """Log request with visual styling"""
```

```
    # Category emojis and colors
```

```
    category_style = {
        "educational": {"emoji": "📖 ", "color": "green"},
        "entertainment": {"emoji": "🎬 ", "color": "blue"},
        "social_media": {"emoji": "📱 ", "color": "yellow"},
        "gaming": {"emoji": "🎮 ", "color": "orange"},
        "inappropriate": {"emoji": "🔞 ", "color": "red"}
    }
```

```
    style = category_style.get(classification["category"], {"emoji": "📡 ", "color": "gray"})
```

```
    # Risk level indicator
```

```
    risk_indicator = {
        "low": "🟢 ",
        "medium": "🟡 ",
        "high": "🔴 "
    }
```

```
    risk_emoji = risk_indicator.get(classification["risk_level"], "🟡 ")
```

```
    alert_indicator = "🔴 " if has_alert else ""
```

```
    # Format log message
```

```
    confidence_str = f"{classification['confidence']:.0%}"
```

```

self.logger.info(
    f"{'style['emoji']}' {risk_emoji} {alert_indicator} "
    f"{'client_ip'} → {'domain'} "
    f"{'classification['category'].upper()}' "
    f"({'confidence_str'})"
)

```

```

def _stats_reporter(self):
    """Report statistics periodically"""
    while self.running:
        time.sleep(60) # Report every minute

        if self.stats['start_time']:
            runtime = datetime.now() - self.stats['start_time']
            runtime_minutes = runtime.total_seconds() / 60

            if runtime_minutes > 0:
                rate = self.stats['total_requests'] / runtime_minutes

                self.logger.info(
                    f"📊 Stats: {self.stats['total_requests']} requests, "
                    f"{self.stats['classified_requests']} classified, "
                    f"{self.stats['alerts_triggered']} alerts, "
                    f"{rate:.1f} req/min"
                )

```

```

def get_device_info(self, ip: str) -> Dict:
    """Get device information from database"""
    try:
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute("""
            SELECT device_name, device_type, last_seen
            FROM devices
            WHERE ip_address = ?
        """, (ip,))

        result = cursor.fetchone()
        conn.close()

        if result:
            return {
                'name': result[0],
                'type': result[1],
                'last_seen': result[2]
            }

```

```

else:
    return {'name': f'Device-{ip.split(".")[1]}', 'type': 'unknown', 'last_seen': None}

except Exception as e:
    self.logger.error(f"Device info error: {e}")
    return {'name': f'Device-{ip.split(".")[1]}', 'type': 'unknown', 'last_seen': None}

# Main execution
if __name__ == '__main__':
    monitor = EnhancedDNSMonitor()

    print("🚀 Starting Enhanced DNS Monitor with AI Classification")
    print("🏠 Features enabled:")
    print("  • Real-time domain classification")
    print("  • Intelligent alerting system")
    print("  • Performance monitoring")
    print("  • Visual activity logging")
    print("\nPress Ctrl+C to stop...")

    try:
        monitor.start()
    except KeyboardInterrupt:
        print("\n🛑 Stopping monitor...")
        monitor.stop()

```

5.2 Create Service Configuration

File: `src/setup_service.py`

```
python
```

```
#!/usr/bin/env python3
```

```
import os
```

```
import subprocess
```

```
import sys
```

```
def create_systemd_service():
```

```
    """Create systemd service for enhanced DNS monitor"""
```

```
    service_content = f"""[Unit]
```

```
Description=SmartParent Enhanced DNS Monitor
```

```
After=network.target ollama.service
```

```
Requires=ollama.service
```

```
[Service]
```

```
Type=simple
```

```
User=pi
```

```
Group=pi
```

```
WorkingDirectory={os.path.abspath('.')}
```

```
Environment=PYTHONPATH={os.path.abspath('.')}
```

```
ExecStart={sys.executable} {os.path.abspath('enhanced_dns_monitor.py')}
```

```
Restart=always
```

```
RestartSec=10
```

```
[Install]
```

```
WantedBy=multi-user.target
```

```
"""
```

```
    service_path = '/etc/systemd/system/smartparent-monitor.service'
```

```
    try:
```

```
        # Write service file
```

```
        with open('/tmp/smartparent-monitor.service', 'w') as f:
```

```
            f.write(service_content)
```

```
        # Move to systemd directory (requires sudo)
```

```
        subprocess.run(['sudo', 'mv', '/tmp/smartparent-monitor.service', service_path], check=True)
```

```
        # Set permissions
```

```
        subprocess.run(['sudo', 'chmod', '644', service_path], check=True)
```

```
        # Reload systemd
```

```
        subprocess.run(['sudo', 'systemctl', 'daemon-reload'], check=True)
```

```
        # Enable service
```

```
        subprocess.run(['sudo', 'systemctl', 'enable', 'smartparent-monitor'], check=True)
```

```
print("✅ SmartParent service created and enabled")
print("📁 Service commands:")
print("  Start:  sudo systemctl start smartparent-monitor")
print("  Stop:   sudo systemctl stop smartparent-monitor")
print("  Status: sudo systemctl status smartparent-monitor")
print("  Logs:   sudo journalctl -u smartparent-monitor -f")

except subprocess.CalledProcessError as e:
    print(f"❌ Service creation failed: {e}")
except Exception as e:
    print(f"❌ Error: {e}")

if __name__ == '__main__':
    create_systemd_service()
```

Step 6: Updated Web Dashboard

6.1 Enhanced Web Dashboard

File: `src/enhanced_web_dashboard.py`

```
python
```

```
#!/usr/bin/env python3
```

```
from flask import Flask, render_template, jsonify, request, redirect, url_for, flash
import sqlite3
import json
from datetime import datetime, timedelta
from typing import Dict, List
import logging
```

```
app = Flask(__name__)
app.secret_key = 'your-secret-key-change-this'
```

```
# Database connection helper
```

```
def get_db_connection():
    conn = sqlite3.connect('../data/smartguard.db')
    conn.row_factory = sqlite3.Row
    return conn
```

```
@app.route('/')
def dashboard():
    """Main dashboard page"""
    return render_template('dashboard.html')
```

```
@app.route('/api/overview')
def api_overview():
    """Get dashboard overview statistics"""
    try:
        conn = get_db_connection()
```

```
# Get today's statistics
```

```
today_stats = conn.execute("""
    SELECT
        COUNT(*) as total_requests,
        COUNT(DISTINCT client_ip) as active_devices,
        COUNT(DISTINCT domain) as unique_domains
    FROM dns_requests
    WHERE DATE(timestamp) = DATE('now')
    """).fetchone()
```

```
# Get category breakdown
```

```
categories = conn.execute("""
    SELECT
        dc.category,
        dc.color,
        COUNT(*) as count,
        ROUND(AVG(dc.confidence), 2) as avg_confidence
    FROM dns_requests dr
```



```
JOIN domain_classifications dc ON dr.domain = dc.domain
WHERE DATE(dr.timestamp) = DATE('now')
GROUP BY dc.category, dc.color
ORDER BY count DESC
''').fetchall()
```

Get risk level distribution

```
risk_levels = conn.execute("""
    SELECT
        dc.risk_level,
        COUNT(*) as count
    FROM dns_requests dr
    JOIN domain_classifications dc ON dr.domain = dc.domain
    WHERE DATE(dr.timestamp) = DATE('now')
    GROUP BY dc.risk_level
''').fetchall()
```

Get active alerts

```
active_alerts = conn.execute("""
    SELECT COUNT(*) as count
    FROM alerts
    WHERE acknowledged = 0
''').fetchone()
```

Get hourly activity for chart

```
hourly_activity = conn.execute("""
    SELECT
        strftime('%H', timestamp) as hour,
        COUNT(*) as requests
    FROM dns_requests
    WHERE DATE(timestamp) = DATE('now')
    GROUP BY strftime('%H', timestamp)
    ORDER BY hour
''').fetchall()
```

```
conn.close()
```

```
return jsonify({
    'overview': {
        'total_requests': today_stats['total_requests'],
        'active_devices': today_stats['active_devices'],
        'unique_domains': today_stats['unique_domains'],
        'active_alerts': active_alerts['count']
    },
    'categories': [dict(row) for row in categories],
    'risk_levels': [dict(row) for row in risk_levels],
    'hourly_activity': [dict(row) for row in hourly_activity]
```

```
})
```

```
except Exception as e:  
    return jsonify({'error': str(e)}), 500
```

```
@app.route('/api/devices')
```

```
def api_devices():
```

```
    """Get device activity"""
```

```
    try:
```

```
        conn = get_db_connection()
```

```
        devices = conn.execute("""
```

```
            SELECT
```

```
                dr.client_ip,
```

```
                d.device_name,
```

```
                d.device_type,
```

```
                COUNT(*) as request_count,
```

```
                COUNT(DISTINCT dr.domain) as unique_domains,
```

```
                MAX(dr.timestamp) as last_activity,
```

```
                GROUP_CONCAT(DISTINCT dc.category) as categories
```

```
            FROM dns_requests dr
```

```
            LEFT JOIN devices d ON dr.client_ip = d.ip_address
```

```
            LEFT JOIN domain_classifications dc ON dr.domain = dc.domain
```

```
            WHERE DATE(dr.timestamp) = DATE('now')
```

```
            GROUP BY dr.client_ip, d.device_name, d.device_type
```

```
            ORDER BY request_count DESC
```

```
        """).fetchall()
```

```
# Get alerts per device
```

```
        device_alerts = conn.execute("""
```

```
            SELECT
```

```
                client_ip,
```

```
                COUNT(*) as alert_count,
```

```
                COUNT(CASE WHEN acknowledged = 0 THEN 1 END) as active_alerts
```

```
            FROM alerts
```

```
            WHERE DATE(created_at) = DATE('now')
```

```
            GROUP BY client_ip
```

```
        """).fetchall()
```

```
        conn.close()
```

```
# Merge alerts with device data
```

```
        alert_dict = {row['client_ip']: row for row in device_alerts}
```

```
        device_list = []
```

```
        for device in devices:
```

```
            device_dict = dict(device)
```

```

device_ip = device['client_ip']

if device_ip in alert_dict:
    device_dict['alert_count'] = alert_dict[device_ip]['alert_count']
    device_dict['active_alerts'] = alert_dict[device_ip]['active_alerts']
else:
    device_dict['alert_count'] = 0
    device_dict['active_alerts'] = 0

device_list.append(device_dict)

return jsonify(device_list)

except Exception as e:
    return jsonify({'error': str(e)}), 500

@app.route('/api/alerts')
def api_alerts():
    """Get alerts with filtering"""
    try:
        conn = get_db_connection()

        # Get filter parameters
        severity = request.args.get('severity', 'all')
        acknowledged = request.args.get('acknowledged', 'active')
        limit = int(request.args.get('limit', 50))

        # Build query
        query = '''
            SELECT
                a.id,
                a.alert_type,
                a.severity,
                a.client_ip,
                a.domain,
                a.category,
                a.message,
                a.acknowledged,
                a.acknowledged_by,
                a.created_at,
                a.metadata,
                d.device_name
            FROM alerts a
            LEFT JOIN devices d ON a.client_ip = d.ip_address
            WHERE 1=1
        '''

```

```

params = []

if severity != 'all':
    query += ' AND a.severity = ?'
    params.append(severity)

if acknowledged == 'active':
    query += ' AND a.acknowledged = 0'
elif acknowledged == 'acknowledged':
    query += ' AND a.acknowledged = 1'

query += ' ORDER BY a.created_at DESC LIMIT ?'
params.append(limit)

alerts = conn.execute(query, params).fetchall()
conn.close()

# Process alerts
alert_list = []
for alert in alerts:
    alert_dict = dict(alert)

    # Parse metadata if present
    if alert['metadata']:
        try:
            alert_dict['metadata'] = json.loads(alert['metadata'])
        except json.JSONDecodeError:
            alert_dict['metadata'] = {}
    else:
        alert_dict['metadata'] = {}

    alert_list.append(alert_dict)

return jsonify(alert_list)

except Exception as e:
    return jsonify({'error': str(e)}), 500

@app.route('/api/alerts/<int:alert_id>/acknowledge', methods=['POST'])
def acknowledge_alert(alert_id):
    """Acknowledge an alert"""
    try:
        conn = get_db_connection()

        conn.execute("""
            UPDATE alerts
            SET acknowledged = 1,

```

```

        acknowledged_by = 'dashboard',
        acknowledged_at = CURRENT_TIMESTAMP
    WHERE id = ?
''' , (alert_id,))

conn.commit()
conn.close()

return jsonify({'success': True})

except Exception as e:
    return jsonify({'error': str(e)}), 500

@app.route('/api/domains/top')
def api_top_domains():
    """Get top domains by category"""
    try:
        conn = get_db_connection()

        category = request.args.get('category', 'all')
        days = int(request.args.get('days', 1))
        limit = int(request.args.get('limit', 20))

        query = '''
            SELECT
                dr.domain,
                dc.category,
                dc.risk_level,
                dc.color,
                COUNT(*) as request_count,
                COUNT(DISTINCT dr.client_ip) as device_count,
                AVG(dc.confidence) as avg_confidence
            FROM dns_requests dr
            JOIN domain_classifications dc ON dr.domain = dc.domain
            WHERE dr.timestamp > datetime('now', '-{} days')
            '''.format(days)

        params = []

        if category != 'all':
            query += ' AND dc.category = ?'
            params.append(category)

        query += '''
            GROUP BY dr.domain, dc.category, dc.risk_level, dc.color
            ORDER BY request_count DESC
            LIMIT ?
        '''

```

```

'''
params.append(limit)

domains = conn.execute(query, params).fetchall()
conn.close()

return jsonify([dict(row) for row in domains])

except Exception as e:
    return jsonify({'error': str(e)}), 500

@app.route('/api/activity/timeline')
def api_activity_timeline():
    """Get activity timeline"""
    try:
        conn = get_db_connection()

        hours = int(request.args.get('hours', 24))

        timeline = conn.execute("""
            SELECT
                datetime((strftime('%s', timestamp) / 300) * 300, 'unixepoch') as time_bucket,
                dc.category,
                COUNT(*) as count
            FROM dns_requests dr
            JOIN domain_classifications dc ON dr.domain = dc.domain
            WHERE dr.timestamp > datetime('now', '-{} hours')
            GROUP BY time_bucket, dc.category
            ORDER BY time_bucket, dc.category
        """.format(hours)).fetchall()

        conn.close()

        return jsonify([dict(row) for row in timeline])

    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/api/config')
def api_config():
    """Get configuration settings"""
    try:
        conn = get_db_connection()

        config = conn.execute("""
            SELECT key, value, description
            FROM configuration

```

```

        WHERE key LIKE 'alert_%' OR key LIKE 'email_%' OR key LIKE 'smtp_%'
    ''').fetchall()

    conn.close()

    return jsonify([dict(row) for row in config])

except Exception as e:
    return jsonify({'error': str(e)}), 500

@app.route('/api/config', methods=['POST'])
def update_config():
    """Update configuration settings"""
    try:
        data = request.get_json()

        conn = get_db_connection()

        for key, value in data.items():
            conn.execute("""
                UPDATE configuration
                SET value = ?
                WHERE key = ?
            """, (value, key))

        conn.commit()
        conn.close()

        flash('Configuration updated successfully', 'success')
        return jsonify({'success': True})

    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/settings')
def settings():
    """Settings page"""
    return render_template('settings.html')

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=False)

```

6.2 Dashboard HTML Template

File: `templates/dashboard.html`


```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>SmartParent Dashboard</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    :root {
      --primary-color: #3498db;
      --success-color: #2ecc71;
      --warning-color: #f39c12;
      --danger-color: #e74c3c;
      --info-color: #17a2b8;
      --dark-color: #343a40;
      --light-color: #f8f9fa;
    }

    * {
      margin: 0;
      padding: 0;
      box-sizing: border-box;
    }

    body {
      font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, sans-serif;
      background-color: var(--light-color);
      color: var(--dark-color);
    }

    .container {
      max-width: 1200px;
      margin: 0 auto;
      padding: 20px;
    }

    .header {
      background: linear-gradient(135deg, var(--primary-color), var(--info-color));
      color: white;
      padding: 20px;
      margin-bottom: 30px;
      border-radius: 10px;
      box-shadow: 0 4px 6px rgba(0,0,0,0.1);
    }

    .header h1 {
```

```
font-size: 2.5rem;
margin-bottom: 10px;
}
```

```
.header p {
  opacity: 0.9;
  font-size: 1.1rem;
}
```

```
.stats-grid {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
  gap: 20px;
  margin-bottom: 30px;
}
```

```
.stat-card {
  background: white;
  padding: 25px;
  border-radius: 10px;
  box-shadow: 0 2px 10px rgba(0,0,0,0.1);
  border-left: 5px solid var(--primary-color);
  transition: transform 0.2s;
}
```

```
.stat-card:hover {
  transform: translateY(-5px);
}
```

```
.stat-value {
  font-size: 2.5rem;
  font-weight: bold;
  color: var(--primary-color);
  margin-bottom: 5px;
}
```

```
.stat-label {
  color: #666;
  font-size: 0.9rem;
  text-transform: uppercase;
  letter-spacing: 1px;
}
```

```
.content-grid {
  display: grid;
  grid-template-columns: 2fr 1fr;
  gap: 30px;
```

```
    margin-bottom: 30px;
}

.card {
    background: white;
    border-radius: 10px;
    box-shadow: 0 2px 10px rgba(0,0,0,0.1);
    overflow: hidden;
}

.card-header {
    background: var(--primary-color);
    color: white;
    padding: 20px;
    font-weight: bold;
    font-size: 1.2rem;
}

.card-body {
    padding: 20px;
}

.category-item {
    display: flex;
    justify-content: space-between;
    align-items: center;
    padding: 10px 0;
    border-bottom: 1px solid #eee;
}

.category-item:last-child {
    border-bottom: none;
}

.category-badge {
    display: inline-block;
    padding: 4px 8px;
    border-radius: 4px;
    color: white;
    font-size: 0.8rem;
    font-weight: bold;
    text-transform: uppercase;
}

.alert-item {
    padding: 15px;
    border-radius: 8px;
```

```
margin-bottom: 10px;
border-left: 4px solid;
}

.alert-high {
  background: #fef2f2;
  border-color: var(--danger-color);
}

.alert-medium {
  background: #fefbf2;
  border-color: var(--warning-color);
}

.alert-low {
  background: #f2fef7;
  border-color: var(--success-color);
}

.btn {
  background: var(--primary-color);
  color: white;
  border: none;
  padding: 8px 16px;
  border-radius: 4px;
  cursor: pointer;
  font-size: 0.9rem;
  transition: background 0.2s;
}

.btn:hover {
  background: #2980b9;
}

.btn-success {
  background: var(--success-color);
}

.btn-success:hover {
  background: #27ae60;
}

.loading {
  text-align: center;
  padding: 40px;
  color: #666;
}
```

```
.error {
  background: #fef2f2;
  color: var(--danger-color);
  padding: 15px;
  border-radius: 8px;
  margin: 20px 0;
}

@media (max-width: 768px) {
  .content-grid {
    grid-template-columns: 1fr;
  }

  .stats-grid {
    grid-template-columns: 1fr;
  }

  .header h1 {
    font-size: 2rem;
  }
}

.refresh-indicator {
  display: inline-block;
  width: 12px;
  height: 12px;
  background: var(--success-color);
  border-radius: 50%;
  margin-left: 10px;
  animation: pulse 2s infinite;
}

@keyframes pulse {
  0%, 100% { opacity: 1; }
  50% { opacity: 0.5; }
}

</style>
</head>
<body>
  <div class="container">
    <div class="header">
      <h1>🛡️ SmartParent Dashboard</h1>
      <p>AI-Powered Network Monitoring & Parental Controls</p>
      <span class="refresh-indicator" id="refreshIndicator"></span>
    </div>
```

```

<div class="stats-grid" id="statsGrid">
  <div class="stat-card">
    <div class="stat-value" id="totalRequests">-</div>
    <div class="stat-label">DNS Requests Today</div>
  </div>
  <div class="stat-card">
    <div class="stat-value" id="activeDevices">-</div>
    <div class="stat-label">Active Devices</div>
  </div>
  <div class="stat-card">
    <div class="stat-value" id="uniqueDomains">-</div>
    <div class="stat-label">Unique Domains</div>
  </div>
  <div class="stat-card">
    <div class="stat-value" id="activeAlerts">-</div>
    <div class="stat-label">Active Alerts</div>
  </div>
</div>

<div class="content-grid">
  <div class="card">
    <div class="card-header"><img alt="Activity Timeline icon" data-bbox="348 448 368 463"/> Activity Timeline</div>
    <div class="card-body">
      <canvas id="activityChart" width="400" height="200"></canvas>
    </div>
  </div>

  <div class="card">
    <div class="card-header"><img alt="Categories icon" data-bbox="351 588 368 603"/> Categories</div>
    <div class="card-body" id="categoriesBody">
      <div class="loading">Loading categories...</div>
    </div>
  </div>
</div>

<div class="content-grid">
  <div class="card">
    <div class="card-header"><img alt="Device Activity icon" data-bbox="348 773 368 788"/> Device Activity</div>
    <div class="card-body" id="devicesBody">
      <div class="loading">Loading devices...</div>
    </div>
  </div>

  <div class="card">
    <div class="card-header"><img alt="Recent Alerts icon" data-bbox="348 913 368 928"/> Recent Alerts</div>
    <div class="card-body" id="alertsBody">
      <div class="loading">Loading alerts...</div>
    </div>
  </div>
</div>

```

```
    </div>
  </div>
</div>
</div>

<script>
  let activityChart;

  // Initialize dashboard
  document.addEventListener('DOMContentLoaded', function() {
    loadDashboard();

    // Refresh every 30 seconds
    setInterval(loadDashboard, 30000);
  });

  async function loadDashboard() {
    try {
      await Promise.all([
        loadOverview(),
        loadDevices(),
        loadAlerts()
      ]);

      // Update refresh indicator
      const indicator = document.getElementById('refreshIndicator');
      indicator.style.animation = 'none';
      setTimeout(() => indicator.style.animation = 'pulse 2s infinite', 100);

    } catch (error) {
      console.error('Dashboard load error:', error);
      showError('Failed to load dashboard data');
    }
  }

  async function loadOverview() {
    const response = await fetch('/api/overview');
    const data = await response.json();

    if (data.error) {
      throw new Error(data.error);
    }

    // Update statistics
    document.getElementById('totalRequests').textContent = data.overview.total_requests.toLocaleString();
    document.getElementById('activeDevices').textContent = data.overview.active_devices;
    document.getElementById('uniqueDomains').textContent = data.overview.unique_domains.toLocaleString();
  }
}
```

```

document.getElementById('activeAlerts').textContent = data.overview.active_alerts;

// Update categories
const categoriesBody = document.getElementById('categoriesBody');
if (data.categories.length > 0) {
  categoriesBody.innerHTML = data.categories.map(cat => `
    <div class="category-item">
      <div>
        <span class="category-badge" style="background-color: ${cat.color}">${cat.category}</span>
      </div>
      <div style="font-weight: bold">${cat.count.toLocaleString()}</div>
    </div>
  `).join("");
} else {
  categoriesBody.innerHTML = '<div style="text-align: center; color: #666;">No activity today</div>';
}

// Update activity chart
updateActivityChart(data.hourly_activity);
}

async function loadDevices() {
  const response = await fetch('/api/devices');
  const data = await response.json();

  if (data.error) {
    throw new Error(data.error);
  }

  const devicesBody = document.getElementById('devicesBody');
  if (data.length > 0) {
    devicesBody.innerHTML = data.map(device => `
      <div style="padding: 10px; border-bottom: 1px solid #eee; display: flex; justify-content: space-between">
        <div>
          <div style="font-weight: bold;">${device.device_name || device.client_ip}</div>
          <div style="color: #666; font-size: 0.9rem;">${device.client_ip} • ${device.request_count} requests</div>
        </div>
        <div>
          ${device.active_alerts > 0 ? `<span style="background: var(--danger-color); color: white; padding: 2px 5px;">${device.active_alerts}</span>` : ''}
        </div>
      </div>
    `).join("");
  } else {
    devicesBody.innerHTML = '<div style="text-align: center; color: #666;">No devices active today</div>';
  }
}

```



```

async function loadAlerts() {
  const response = await fetch('/api/alerts?limit=10');
  const data = await response.json();

  if (data.error) {
    throw new Error(data.error);
  }

  const alertsBody = document.getElementById('alertsBody');
  if (data.length > 0) {
    alertsBody.innerHTML = data.map(alert => `
      <div class="alert-item alert-${alert.severity}">
        <div style="display: flex; justify-content: space-between; align-items: center; margin-bottom: 5px;">
          <strong>${alert.alert_type}</strong>
          ${!alert.acknowledged ? `<button class="btn btn-success" onclick="acknowledgeAlert('${alert.id}'`
        </div>
        <div style="font-size: 0.9rem; color: #666;">
          ${alert.device_name || alert.client_ip} • ${alert.domain} • ${new Date(alert.created_at).toLocaleSt
        </div>
        <div style="margin-top: 5px;">${alert.message}</div>
      </div>
    `).join('');
  } else {
    alertsBody.innerHTML = '<div style="text-align: center; color: #666;">No recent alerts</div>';
  }
}

function updateActivityChart(hourly_data) {
  const ctx = document.getElementById('activityChart').getContext('2d');

  // Create 24-hour labels
  const labels = Array.from({length: 24}, (_, i) => `${i.toString().padStart(2, '0')}:00`);

  // Create data array with zeros
  const data = new Array(24).fill(0);

  // Fill with actual data
  hourly_data.forEach(item => {
    const hour = parseInt(item.hour);
    data[hour] = item.requests;
  });

  if (activityChart) {
    activityChart.destroy();
  }

  activityChart = new Chart(ctx, {

```

```

    type: 'line',
    data: {
      labels: labels,
      datasets: [{
        label: 'DNS Requests',
        data: data,
        borderColor: 'rgb(52, 152, 219)',
        backgroundColor: 'rgba(52, 152, 219, 0.1)',
        tension: 0.4,
        fill: true
      }]
    },
    options: {
      responsive: true,
      plugins: {
        legend: {
          display: false
        }
      },
      scales: {
        y: {
          beginAtZero: true
        }
      }
    }
  });
}

```

```

async function acknowledgeAlert(alertId) {
  try {
    const response = await fetch(`/api/alerts/${alertId}/acknowledge`, {
      method: 'POST'
    });

    if (response.ok) {
      loadAlerts(); // Refresh alerts
    } else {
      showError('Failed to acknowledge alert');
    }
  } catch (error) {
    console.error('Acknowledge error:', error);
    showError('Failed to acknowledge alert');
  }
}

```

```

function showError(message) {
  const errorDiv = document.createElement('div');

```

```
errorDiv.className = 'error';
errorDiv.textContent = message;

const container = document.querySelector('.container');
container.insertBefore(errorDiv, container.firstChild);

setTimeout(() => errorDiv.remove(), 5000);
}
</script>
</body>
</html>
```

Step 7: End-to-End Testing

7.1 Comprehensive Test Suite

File: `src/test_step7_comprehensive.py`

```
python
```

```
#!/usr/bin/env python3
import sys
import time
import sqlite3
import requests
import subprocess
import threading
from datetime import datetime, timedelta
from domain_classifier import DomainClassifier
from alert_system import AlertSystem
from enhanced_dns_monitor import EnhancedDNSMonitor
```

```
class ComprehensiveTestSuite:
```

```
    def __init__(self):
```

```
        self.db_path = './data/smartguard.db'
```

```
        self.results = {
```

```
            'total_tests': 0,
```

```
            'passed_tests': 0,
```

```
            'failed_tests': 0,
```

```
            'test_details': []
```

```
        }
```

```
    def run_test(self, test_name, test_func):
```

```
        """Run a single test and record results"""
```

```
        self.results['total_tests'] += 1
```

```
        try:
```

```
            print(f"\n🏁 Running: {test_name}")
```

```
            start_time = time.time()
```

```
            result = test_func()
```

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

```
            if result:
```

```
                print(f"✅ PASSED ({duration:.2f}s)")
```

```
                self.results['passed_tests'] += 1
```

```
                self.results['test_details'].append({
```

```
                    'name': test_name,
```

```
                    'status': 'PASSED',
```

```
                    'duration': duration,
```

```
                    'message': 'Test completed successfully'
```

```
                })
```

```
            else:
```

```
                print(f"❌ FAILED ({duration:.2f}s)")
```

```
                self.results['failed_tests'] += 1
```

```
self.results['test_details'].append({
    'name': test_name,
    'status': 'FAILED',
    'duration': duration,
    'message': 'Test assertion failed'
})
```

```
except Exception as e:
```

```
    print(f"❌ ERROR: {e}")
```

```
    self.results['failed_tests'] += 1
```

```
    self.results['test_details'].append({
```

```
        'name': test_name,
```

```
        'status': 'ERROR',
```

```
        'duration': 0,
```

```
        'message': str(e)
```

```
    })
```

```
def test_database_schema(self):
```

```
    """Test database schema completeness"""
```

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

```
    cursor = conn.cursor()
```

```
    # Check all required tables exist
```

```
    cursor.execute("SELECT name FROM sqlite_master WHERE type='table'")
```

```
    tables = [row[0] for row in cursor.fetchall()]
```

```
    required_tables = [
```

```
        'dns_requests', 'devices', 'traffic_stats', 'system_events',
```

```
        'configuration', 'domain_classifications', 'alert_rules', 'alerts'
```

```
    ]
```

```
    missing_tables = [table for table in required_tables if table not in tables]
```

```
    if missing_tables:
```

```
        print(f"Missing tables: {missing_tables}")
```

```
        return False
```

```
    # Check alert_rules has default rules
```

```
    cursor.execute("SELECT COUNT(*) FROM alert_rules")
```

```
    rule_count = cursor.fetchone()[0]
```

```
    if rule_count < 3:
```

```
        print(f"Insufficient alert rules: {rule_count}")
```

```
        return False
```

```
    conn.close()
```

```
    print("All database tables and rules present")
```

```

return True

def test_ollama_service(self):
    """Test Ollama service is running and responding"""
    try:
        response = requests.get('http://localhost:11434/api/tags', timeout=5)

        if response.status_code != 200:
            print(f" Ollama service not responding: {response.status_code}")
            return False

        data = response.json()

        # Check if phi3 model is available
        models = [model['name'] for model in data.get('models', [])]
        phi3_models = [m for m in models if 'phi3' in m.lower()]

        if not phi3_models:
            print(f" Phi3 model not found. Available models: {models}")
            return False

        print(f" Ollama service running with {len(models)} models")
        return True

    except requests.RequestException as e:
        print(f" Ollama service connection failed: {e}")
        return False

def test_domain_classification(self):
    """Test domain classification accuracy"""
    classifier = DomainClassifier(self.db_path)

    test_domains = [
        ('facebook.com', 'social_media'),
        ('khanacademy.org', 'educational'),
        ('youtube.com', 'entertainment'),
        ('minecraft.net', 'gaming'),
        ('google.com', 'entertainment') # Could be educational but defaults to entertainment
    ]

    correct_classifications = 0
    total_tests = len(test_domains)

    for domain, expected_category in test_domains:
        try:
            result = classifier.classify_domain(domain)
            if result['category'] == expected_category:

```

```

        correct_classifications += 1
        print(f"  ✓ {domain} -> {result['category']}")
    else:
        print(f"  ✗ {domain} -> {result['category']} (expected {expected_category})")

except Exception as e:
    print(f"  ✗ {domain} -> ERROR: {e}")

accuracy = correct_classifications / total_tests
print(f"  Classification accuracy: {accuracy:.1%} ({correct_classifications}/{total_tests})")

return accuracy >= 0.6 # Require 60% accuracy

def test_alert_system(self):
    """Test alert system functionality"""
    alert_system = AlertSystem(self.db_path)

    # Test alert creation
    test_alert = {
        "rule_name": "Test Alert",
        "client_ip": "192.168.1.99",
        "domain": "test.com",
        "category": "gaming",
        "severity": "medium",
        "message": "Test alert message",
        "request_count": 5,
        "time_window": 300
    }

    alert_id = alert_system.create_alert(test_alert)

    if not alert_id:
        print("  Alert creation failed")
        return False

    # Test alert retrieval
    active_alerts = alert_system.get_active_alerts()

    if not any(alert['id'] == alert_id for alert in active_alerts):
        print("  Alert not found in active alerts")
        return False

    # Test alert acknowledgment
    success = alert_system.acknowledge_alert(alert_id, "test_system")

    if not success:
        print("  Alert acknowledgment failed")

```

```
return False
```

```
print(f" Alert system fully functional (Alert ID: {alert_id})")
```

```
return True
```

```
def test_web_dashboard(self):
```

```
    """Test web dashboard API endpoints"""
```

```
    base_url = 'http://localhost:5000'
```

```
    endpoints = [
```

```
        '/api/overview',
```

```
        '/api/devices',
```

```
        '/api/alerts',
```

```
        '/api/domains/top',
```

```
        '/api/config'
```

```
    ]
```

```
    working_endpoints = 0
```

```
    for endpoint in endpoints:
```

```
        try:
```

```
            response = requests.get(f"{base_url}{endpoint}", timeout=5)
```

```
            if response.status_code == 200:
```

```
                working_endpoints += 1
```

```
                print(f" ✓ {endpoint}")
```

```
            else:
```

```
                print(f" ✗ {endpoint} -> {response.status_code}")
```

```
        except requests.RequestException as e:
```

```
            print(f" ✗ {endpoint} -> {e}")
```

```
    success_rate = working_endpoints / len(endpoints)
```

```
    print(f" Dashboard API success rate: {success_rate:.1%} ({working_endpoints}/{len(endpoints)})")
```

```
    return success_rate >= 0.8 # Require 80% of endpoints working
```

```
def test_performance(self):
```

```
    """Test system performance benchmarks"""
```

```
    classifier = DomainClassifier(self.db_path)
```

```
    test_domains = [
```

```
        'example.com', 'test.org', 'sample.net', 'demo.edu', 'trial.gov'
```

```
    ]
```

```
    total_time = 0
```

```
    successful_classifications = 0
```



```

for domain in test_domains:
    start_time = time.time()

    try:
        result = classifier.classify_domain(domain)

        if result and 'category' in result:
            successful_classifications += 1

    except Exception as e:
        print(f" Classification error for {domain}: {e}")

    classification_time = time.time() - start_time
    total_time += classification_time

avg_time = total_time / len(test_domains) if test_domains else 0
success_rate = successful_classifications / len(test_domains) if test_domains else 0

print(f" Average classification time: {avg_time:.2f}s")
print(f" Classification success rate: {success_rate:.1%}")

# Performance benchmarks
return avg_time < 10.0 and success_rate >= 0.8

def test_data_integrity(self):
    """Test data consistency and integrity"""
    conn = sqlite3.connect(self.db_path)
    cursor = conn.cursor()

    # Test foreign key constraints
    cursor.execute("""
        SELECT COUNT(*) FROM dns_requests dr
        LEFT JOIN domain_classifications dc ON dr.domain = dc.domain
        WHERE dc.domain IS NULL AND dr.classification IS NOT NULL
    """)

    orphaned_requests = cursor.fetchone()[0]

    if orphaned_requests > 0:
        print(f" Found {orphaned_requests} orphaned DNS requests")
        return False

    # Test data consistency
    cursor.execute("SELECT COUNT(*) FROM dns_requests WHERE timestamp IS NULL")
    null_timestamps = cursor.fetchone()[0]

```

```

if null_timestamps > 0:
    print(f" Found {null_timestamps} requests with null timestamps")
    return False

print(" Data integrity checks passed")
conn.close()
return True

def run_all_tests(self):
    """Run complete test suite"""
    print("🚀 Starting Comprehensive Test Suite for Steps 4-8")
    print("=" * 60)

    # Core functionality tests
    self.run_test("Database Schema Validation", self.test_database_schema)
    self.run_test("Ollama Service Connectivity", self.test_ollama_service)
    self.run_test("Domain Classification Accuracy", self.test_domain_classification)
    self.run_test("Alert System Functionality", self.test_alert_system)
    self.run_test("Web Dashboard API", self.test_web_dashboard)

    # Performance and integrity tests
    self.run_test("Performance Benchmarks", self.test_performance)
    self.run_test("Data Integrity", self.test_data_integrity)

    # Print summary
    self.print_summary()

    return self.results['failed_tests'] == 0

def print_summary(self):
    """Print test summary"""
    print("\n" + "=" * 60)
    print("📊 TEST SUMMARY")
    print("=" * 60)

    total = self.results['total_tests']
    passed = self.results['passed_tests']
    failed = self.results['failed_tests']

    success_rate = (passed / total * 100) if total > 0 else 0

    print(f"Total Tests: {total}")
    print(f"Passed: {passed} ✅ ")
    print(f"Failed: {failed} ❌ ")
    print(f"Success Rate: {success_rate:.1f}%")

    if failed > 0:

```

```
print(f"\n❌ FAILED TESTS:")
for test in self.results['test_details']:
    if test['status'] in ['FAILED', 'ERROR']:
        print(f"    • {test['name']}: {test['message']}")

print(f"\n🎉 ALL TESTS PASSED! if failed == 0 else '⚠️ SOME TESTS FAILED'")

if failed == 0:
    print("\n✅ System is ready for production deployment!")
else:
    print("\n🔧 Please address failed tests before proceeding.")

if __name__ == '__main__':
    test_suite = ComprehensiveTestSuite()
    success = test_suite.run_all_tests()

    sys.exit(0 if success else 1)
```

Step 8: Performance Optimization

8.1 Performance Monitoring and Optimization

File: `src/performance_optimizer.py`

python

```
#!/usr/bin/env python3
```

```
import sqlite3
```

```
import psutil
```

```
import time
```

```
import logging
```

```
import threading
```

```
from datetime import datetime, timedelta
```

```
from typing import Dict, List
```

```
class PerformanceOptimizer:
```

```
    def __init__(self, db_path: str = './data/smartguard.db'):
```

```
        self.db_path = db_path
```

```
        self.logger = logging.getLogger(__name__)
```

```
        self.monitoring = False
```

```
    def optimize_database(self):
```

```
        """Optimize database for better performance"""
```

```
        print("🔧 Optimizing database...")
```

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

```
        cursor = conn.cursor()
```

```
        # Create indexes for better query performance
```

```
        indexes = [
```

```
            "CREATE INDEX IF NOT EXISTS idx_dns_requests_timestamp ON dns_requests(timestamp)",
```

```
            "CREATE INDEX IF NOT EXISTS idx_dns_requests_client_ip ON dns_requests(client_ip)",
```

```
            "CREATE INDEX IF NOT EXISTS idx_dns_requests_domain ON dns_requests(domain)",
```

```
            "CREATE INDEX IF NOT EXISTS idx_dns_requests_classification ON dns_requests(classification)",
```

```
            "CREATE INDEX IF NOT EXISTS idx_domain_classifications_category ON domain_classifications(category)",
```

```
            "CREATE INDEX IF NOT EXISTS idx_domain_classifications_risk ON domain_classifications(risk_level)",
```

```
            "CREATE INDEX IF NOT EXISTS idx_alerts_client_ip ON alerts(client_ip)",
```

```
            "CREATE INDEX IF NOT EXISTS idx_alerts_created_at ON alerts(created_at)",
```

```
            "CREATE INDEX IF NOT EXISTS idx_alerts_acknowledged ON alerts(acknowledged)",
```

```
        ]
```

```
        for index_sql in indexes:
```

```
            cursor.execute(index_sql)
```

```
        # Analyze tables for query optimization
```

```
        cursor.execute("ANALYZE")
```

```
        # Vacuum database to reclaim space
```

```
        cursor.execute("VACUUM")
```

```
        conn.commit()
```

```
        conn.close()
```

```

print("✅ Database optimization completed")

def cleanup_old_data(self, retention_days: int = 30):
    """Clean up old data to maintain performance"""
    print(f"🧹 Cleaning up data older than {retention_days} days...")

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

    # Clean old DNS requests (keep last 30 days)
    cursor.execute("""
        DELETE FROM dns_requests
        WHERE timestamp < datetime('now', '-{} days')
    """.format(retention_days))
    old_requests = cursor.rowcount

    # Clean old acknowledged alerts (keep last 7 days)
    cursor.execute("""
        DELETE FROM alerts
        WHERE acknowledged = 1
        AND acknowledged_at < datetime('now', '-7 days')
    """)
    old_alerts = cursor.rowcount

    # Clean old domain classifications not referenced by recent requests
    cursor.execute("""
        DELETE FROM domain_classifications
        WHERE domain NOT IN (
            SELECT DISTINCT domain FROM dns_requests
            WHERE timestamp > datetime('now', '-{} days')
        )
        AND timestamp < datetime('now', '-{} days')
    """.format(retention_days, retention_days))
    old_classifications = cursor.rowcount

    conn.commit()
    conn.close()

    print(f"✅ Cleanup completed:")
    print(f"  • Removed {old_requests} old DNS requests")
    print(f"  • Removed {old_alerts} old alerts")
    print(f"  • Removed {old_classifications} old classifications")

def optimize_classification_cache(self):
    """Optimize classification caching strategy"""
    print("🗄️ Optimizing classification cache...")

```

```

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

# Find frequently requested domains for permanent caching
cursor.execute("""
    SELECT domain, COUNT(*) as request_count
    FROM dns_requests
    WHERE timestamp > datetime('now', '-7 days')
    GROUP BY domain
    HAVING COUNT(*) >= 10
    ORDER BY request_count DESC
""")

frequent_domains = cursor.fetchall()

print(f" Found {len(frequent_domains)} frequently requested domains")

# Extend cache time for frequent domains
for domain, count in frequent_domains:
    cursor.execute("""
        UPDATE domain_classifications
        SET timestamp = datetime('now')
        WHERE domain = ?
    """, (domain,))

conn.commit()
conn.close()

print("✅ Classification cache optimized")

def monitor_system_resources(self, duration_minutes: int = 5):
    """Monitor system resources"""
    print(f"📊 Monitoring system resources for {duration_minutes} minutes...")

    self.monitoring = True
    metrics = {
        'cpu_usage': [],
        'memory_usage': [],
        'disk_usage': [],
        'network_connections': []
    }

    def collect_metrics():
        while self.monitoring:
            metrics['cpu_usage'].append(psutil.cpu_percent(interval=1))
            metrics['memory_usage'].append(psutil.virtual_memory().percent)

```

```
metrics['disk_usage'].append(psutil.disk_usage('/').percent)
metrics['network_connections'].append(len(psutil.net_connections()))
```

```
if not self.monitoring:
    break
```

```
monitor_thread = threading.Thread(target=collect_metrics)
monitor_thread.start()
```

```
time.sleep(duration_minutes * 60)
self.monitoring = False
monitor_thread.join()
```

```
# Calculate averages
```

```
avg_cpu = sum(metrics['cpu_usage']) / len(metrics['cpu_usage'])
avg_memory = sum(metrics['memory_usage']) / len(metrics['memory_usage'])
avg_disk = sum(metrics['disk_usage']) / len(metrics['disk_usage'])
avg_connections = sum(metrics['network_connections']) / len(metrics['network_connections'])
```

```
print("📊 Resource Usage Summary:")
print(f"   • Average CPU: {avg_cpu:.1f}%")
print(f"   • Average Memory: {avg_memory:.1f}%")
print(f"   • Average Disk: {avg_disk:.1f}%")
print(f"   • Average Connections: {avg_connections:.0f}")
```

```
# Performance recommendations
```

```
if avg_cpu > 80:
    print("⚠️ High CPU usage detected - consider reducing classification frequency")

if avg_memory > 85:
    print("⚠️ High memory usage detected - consider increasing cache cleanup frequency")

if avg_disk > 90:
    print("⚠️ High disk usage detected - consider more aggressive data cleanup")
```

```
return metrics
```

```
def tune_alert_rules(self):
```

```
    """Tune alert rules for better performance"""
    print("🔧 Tuning alert rules...")
```

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

```
# Analyze alert frequency
```

```
cursor.execute("""
    SELECT rule_name, COUNT(*) as alert_count,
```

```

        AVG(CASE WHEN acknowledged = 1 THEN 1 ELSE 0 END) as ack_rate
FROM alerts
WHERE created_at > datetime('now', '-7 days')
GROUP BY rule_name
'''

```

```

alert_stats = cursor.fetchall()

```

```

for rule_name, count, ack_rate in alert_stats:

```

```

    if ack_rate < 0.5 and count > 10: # Low acknowledgment rate, high volume

```

```

        print(f" ⚠️ Rule '{rule_name}' has low acknowledgment rate ({ack_rate:.1%}) - consider tuning")

```

```

    # Increase thresholds for noisy rules

```

```

    cursor.execute("""

```

```

        UPDATE alert_rules

```

```

        SET request_threshold = request_threshold * 1.5,

```

```

            time_threshold = time_threshold * 1.2

```

```

        WHERE rule_name = ?

```

```

    """, (rule_name,))

```

```

conn.commit()

```

```

conn.close()

```

```

print("✅ Alert rules tuned")

```

```

def generate_performance_report(self):

```

```

    """Generate comprehensive performance report"""

```

```

    print("📊 Generating performance report...")

```

```

    conn = sqlite3.connect(self.db_path)

```

```

    cursor = conn.cursor()

```

```

    # Database statistics

```

```

    cursor.execute("SELECT name FROM sqlite_master WHERE type='table'")

```

```

    tables = [row[0] for row in cursor.fetchall()]

```

```

    table_stats = {}

```

```

    for table in tables:

```

```

        cursor.execute(f"SELECT COUNT(*) FROM {table}")

```

```

        table_stats[table] = cursor.fetchone()[0]

```

```

    # Recent activity statistics

```

```

    cursor.execute("""

```

```

        SELECT

```

```

            COUNT(*) as total_requests,

```

```

            COUNT(DISTINCT client_ip) as unique_devices,

```

```

            COUNT(DISTINCT domain) as unique_domains

```



```
FROM dns_requests
WHERE timestamp > datetime('now', '-24 hours')
'''
```

```
activity_stats = cursor.fetchone()
```

```
# Classification performance
```

```
cursor.execute("""
SELECT
    category,
    COUNT(*) as count,
    AVG(confidence_score) as avg_confidence
FROM dns_requests dr
JOIN domain_classifications dc ON dr.domain = dc.domain
WHERE dr.timestamp > datetime('now', '-24 hours')
GROUP BY category
''')
```

```
classification_stats = cursor.fetchall()
```

```
# Alert statistics
```

```
cursor.execute("""
SELECT
    severity,
    COUNT(*) as count,
    AVG(CASE WHEN acknowledged = 1 THEN 1 ELSE 0 END) as ack_rate
FROM alerts
WHERE created_at > datetime('now', '-24 hours')
GROUP BY severity
''')
```

```
alert_stats = cursor.fetchall()
```

```
conn.close()
```

```
# System information
```

```
system_info = {
    'cpu_count': psutil.cpu_count(),
    'memory_total': psutil.virtual_memory().total // (1024**3), # GB
    'disk_total': psutil.disk_usage('/').total // (1024**3),    # GB
    'disk_free': psutil.disk_usage('/').free // (1024**3)      # GB
}
```

```
# Generate report
```

```
report = f"""
```

SmartParent Performance Report

Generated: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}

```
{'='*50}
```

SYSTEM INFORMATION:

- CPU Cores: {system_info['cpu_count']}
- Total Memory: {system_info['memory_total']} GB
- Total Disk: {system_info['disk_total']} GB
- Free Disk: {system_info['disk_free']} GB

DATABASE STATISTICS:

```
"""
```

```
for table, count in table_stats.items():  
    report += f" • {table}: {count:,} records\n"
```

```
report += f""
```

24-HOUR ACTIVITY:

- Total DNS Requests: {activity_stats[0]:,}
- Unique Devices: {activity_stats[1]}
- Unique Domains: {activity_stats[2]:,}

CLASSIFICATION BREAKDOWN:

```
"""
```

```
for category, count, confidence in classification_stats:  
    report += f" • {category.title():} {count:,} requests (avg confidence: {confidence:.1%})\n"
```

```
report += "\nALERT STATISTICS:\n"
```

```
for severity, count, ack_rate in alert_stats:  
    report += f" • {severity.title():} {count} alerts (ack rate: {ack_rate:.1%})\n"
```

```
# Write report to file
```

```
with open('../logs/performance_report.txt', 'w') as f:  
    f.write(report)
```

```
print("✅ Performance report saved to logs/performance_report.txt")
```

```
print("\n📊 Quick Summary:")
```

```
print(f" • Total records: {sum(table_stats.values()):,}")
```

```
print(f" • 24h requests: {activity_stats[0]:,}")
```

```
print(f" • Active devices: {activity_stats[1]}")
```

```
print(f" • Disk free: {system_info['disk_free']} GB")
```

```
def main():
```

```
    """Run performance optimization suite"""
```

```
    print("🚀 SmartParent Performance Optimization Suite")
```

```
    print("=" * 50)
```

```
optimizer = PerformanceOptimizer()
```

```
# Run optimizations
```

```
optimizer.optimize_database()
```

```
optimizer.cleanup_old_data()
```

```
optimizer.optimize_classification_cache()
```

```
optimizer.tune_alert_rules()
```

```
# Generate report
```

```
optimizer.generate_performance_report()
```

```
print("\n🎉 Performance optimization completed!")
```

```
print("\n📋 Recommended next steps:")
```

```
print("1. Monitor system for 24 hours")
```

```
print("2. Review performance report")
```

```
print("3. Adjust settings based on usage patterns")
```

```
print("4. Schedule regular maintenance")
```

```
if __name__ == '__main__':
```

```
    main()
```

Deployment and Final Steps

9.1 Production Deployment Script

File: `src/deploy_production.py`

```
python
```

```
#!/usr/bin/env python3
import os
import subprocess
import sys
import json
from datetime import datetime

def deploy_production():
    """Deploy SmartParent system for production use"""
    print("🚀 Deploying SmartParent for Production")
    print("=" * 50)

    # Check prerequisites
    if not check_prerequisites():
        return False

    # Create necessary directories
    create_directories()

    # Set up logging
    setup_logging()

    # Configure services
    setup_services()

    # Set up monitoring
    setup_monitoring()

    # Create backup script
    create_backup_script()

    # Final verification
    if verify_deployment():
        print("\n🎉 Production deployment completed successfully!")
        print_deployment_summary()
        return True
    else:
        print("\n❌ Deployment verification failed")
        return False

def check_prerequisites():
    """Check all prerequisites are met"""
    print("🔍 Checking prerequisites...")

    checks = [
        ("Python 3.7+", lambda: sys.version_info >= (3, 7)),
    ]
```

```
("Ollama service", lambda: subprocess.run(['systemctl', 'is-active', 'ollama'], capture_output=True).returncode == 0),
("Database exists", lambda: os.path.exists('../data/smartguard.db')),
("Web dashboard", lambda: os.path.exists('enhanced_web_dashboard.py')),
("AI classifier", lambda: os.path.exists('domain_classifier.py')),
("Alert system", lambda: os.path.exists('alert_system.py'))
]
```

```
all_passed = True
```

```
for check_name, check_func in checks:
```

```
    try:
```

```
        if check_func():
```

```
            print(f"✅ {check_name}")
```

```
        else:
```

```
            print(f"❌ {check_name}")
```

```
            all_passed = False
```

```
    except Exception as e:
```

```
        print(f"
```