SmartParent Phase 2: AI-Stack Setup

Implementation Guide - Steps 1-3

Project: SmartParent - Intelligent Parental Network Monitor
Phase: 2 - AI-Stack Setup
Duration: Week 3-4
Document Version: 1.0
Date: July 2025

Table of Contents

Prerequisites

Before starting Phase 2, ensure you have completed Phase 1 with:

- ✅ Raspberry Pi 4 with static IP (192.168.1.100)
- ✅ DNS monitoring system working
- ✅ Database with basic schema
- ✅ Web dashboard functional
- ✅ Network devices using Pi as DNS server

System Requirements:

- Raspberry Pi 4B (4GB RAM minimum)
- 64GB microSD card with 20GB+ free space
- Stable internet connection for model download
- SSH access to the Pi

AI-Stack Architecture Overview

This phase implements a complete AI infrastructure stack with three layers:

Layer 1: AI Runtime (Step 1)

- **Ollama Service**: Local AI model serving platform
- **Phi-3-mini Model**: 3.8B parameter language model
- **Purpose**: Provides AI inference capabilities locally

Layer 2: AI Application (Step 2)

- **Domain Classifier**: Python module for website categorization
- **Classification Engine**: Converts domains to risk categories
- **Caching System**: Optimizes performance and reduces compute

Layer 3: AI Data (Step 3)

- **Classification Storage**: Database tables for AI results
- **Alert Rules**: Configurable intelligence-based alerts
- **Analytics Schema**: Support for behavioral analysis

Architecture Benefits:

- **Local Processing**: No cloud dependencies
- **Real-time**: Sub-5 second classification
- **Privacy-First**: Data never leaves your network
- **Scalable**: Can handle typical home network traffic

Step 1: AI Runtime Setup (Ollama + Phi-3-mini)

1.1 Install Ollama Service

bash

```
# SSH into your Pi
ssh pi@192.168.1.100

# Install Ollama (ARM64 optimized)
curl -fsSL https://ollama.com/install.sh | sh

# Verify installation
ollama --version
```

Expected Output:

```
ollama version is 0.1.47
```

1.2 Download Phi-3-mini Model

```bash
# Pull the quantized model (optimized for Pi)
ollama pull phi3:3.8b-mini-instruct-q4_K_M

# This will download ~2.2GB - may take 10-30 minutes
```

Model Details:

- **Size**: ~2.2GB download
- **Parameters**: 3.8 billion
- **Quantization**: Q4_K_M (4-bit quantized)
- **Memory Usage**: ~1.5GB RAM during inference

1.3 Test Model Installation

```bash
# Test basic inference
ollama run phi3:3.8b-mini-instruct-q4_K_M "Hello, test message"

# Test domain classification
ollama run phi3:3.8b-mini-instruct-q4_K_M "Classify this domain: facebook.com into categories: educational, entertainm
```

Expected Response:

```
social_media
```

1.4 Configure Ollama as Service

```bash
```

```
# Enable Ollama to start on boot
sudo systemctl enable ollama
sudo systemctl start ollama

# Check service status
sudo systemctl status ollama

# Test API endpoint
curl http://localhost:11434/api/tags
```

Expected JSON Response:

```
json
{
  "models": [
    {
      "name": "phi3:3.8b-mini-instruct-q4_K_M",
      "modified_at": "2025-07-16T10:30:00Z",
      "size": 2200000000
    }
  ]
}
```

Step 2: AI Application Layer (Domain Classification System)

2.1 Create Domain Classifier Module

Create `src/domain_classifier.py`:

```
python
```

```python
#!/usr/bin/env python3
import requests
import json
import sqlite3
import logging
from typing import Dict, Optional
from datetime import datetime

class DomainClassifier:
    def __init__(self, db_path: str = '../data/smartguard.db'):
        self.db_path = db_path
        self.ollama_url = "http://localhost:11434/api/generate"
        self.categories = {
            "educational": {"color": "green", "risk": "low"},
            "entertainment": {"color": "blue", "risk": "low"},
            "social_media": {"color": "yellow", "risk": "medium"},
            "gaming": {"color": "orange", "risk": "medium"},
            "inappropriate": {"color": "red", "risk": "high"}
        }

        # Setup logging
        self.logger = logging.getLogger(__name__)

    def get_classification_prompt(self, domain: str) -> str:
        """Generate classification prompt for domain"""
        return f"""
Analyze the domain "{domain}" and classify it into ONE of these categories:

1. educational - School, university, research, learning platforms, reference materials
2. entertainment - Movies, music, TV shows, news, sports, general entertainment
3. social_media - Facebook, Instagram, TikTok, Twitter, messaging platforms
4. gaming - Gaming websites, platforms, online games, game-related content
5. inappropriate - Adult content, violence, illegal activities, harmful material

Rules:
- Return ONLY the category name (no explanation)
- If unsure, classify as "entertainment"
- Consider the primary purpose of the domain
- Popular sites like YouTube could be "entertainment" or "educational" - use "entertainment"

Domain: {domain}
Category:"""

    def classify_domain(self, domain: str) -> Dict[str, any]:
        """Classify domain using Ollama"""
        try:
```

```
        # Check cache first
        cached_result = self.get_cached_classification(domain)
        if cached_result:
            return cached_result

        # Generate classification
        prompt = self.get_classification_prompt(domain)

        payload = {
            "model": "phi3:3.8b-mini-instruct-q4_K_M",
            "prompt": prompt,
            "stream": False,
            "options": {
                "temperature": 0.1,
                "top_p": 0.9,
                "max_tokens": 20
            }
        }

        response = requests.post(self.ollama_url, json=payload, timeout=30)
        response.raise_for_status()

        result = response.json()
        category = result.get("response", "entertainment").strip().lower()

        # Validate category
        if category not in self.categories:
            category = "entertainment"

        # Calculate confidence (simplified)
        confidence = 0.85 if category in ["educational", "inappropriate"] else 0.75

        classification = {
            "domain": domain,
            "category": category,
            "confidence": confidence,
            "risk_level": self.categories[category]["risk"],
            "color": self.categories[category]["color"],
            "timestamp": datetime.now().isoformat()
        }

        # Cache result
        self.cache_classification(classification)

        return classification

    except Exception as e:
```

```
            self.logger.error(f"Classification error for {domain}: {e}")
            return {
                "domain": domain,
                "category": "entertainment",
                "confidence": 0.5,
                "risk_level": "low",
                "color": "blue",
                "timestamp": datetime.now().isoformat()
            }

    def get_cached_classification(self, domain: str) -> Optional[Dict]:
        """Get cached classification from database"""
        try:
            conn = sqlite3.connect(self.db_path)
            cursor = conn.cursor()

            cursor.execute('''
                SELECT category, confidence, risk_level, color, timestamp
                FROM domain_classifications
                WHERE domain = ? AND timestamp > datetime('now', '-7 days')
            ''', (domain,))

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

            if result:
                return {
                    "domain": domain,
                    "category": result[0],
                    "confidence": result[1],
                    "risk_level": result[2],
                    "color": result[3],
                    "timestamp": result[4]
                }
            return None

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

    def cache_classification(self, classification: Dict):
        """Cache classification result"""
        try:
            conn = sqlite3.connect(self.db_path)
            cursor = conn.cursor()

            cursor.execute('''
```

```
                INSERT OR REPLACE INTO domain_classifications
                (domain, category, confidence, risk_level, color, timestamp)
                VALUES (?, ?, ?, ?, ?, ?)
            ''', (
                classification["domain"],
                classification["category"],
                classification["confidence"],
                classification["risk_level"],
                classification["color"],
                classification["timestamp"]
            ))

            conn.commit()
            conn.close()

        except Exception as e:
            self.logger.error(f"Cache store error: {e}")

# Test the classifier
if __name__ == '__main__':
    classifier = DomainClassifier()

    test_domains = [
        "facebook.com",
        "khanacademy.org",
        "pornhub.com",
        "minecraft.net",
        "google.com"
    ]

    for domain in test_domains:
        result = classifier.classify_domain(domain)
        print(f"{domain} -> {result['category']} ({result['confidence']:.2f})")
```

2.2 Test Domain Classifier

```
bash

cd ~/smartguard-monitor/src
source ../venv/bin/activate

# Test the classifier
python3 domain_classifier.py
```

Expected Output:

facebook.com -> social_media (0.85)
khanacademy.org -> educational (0.85)
pornhub.com -> inappropriate (0.85)
minecraft.net -> gaming (0.75)
google.com -> entertainment (0.75)

Step 3: AI Data Layer (Database Schema Updates)

3.1 Create Database Update Script

Create `src/update_database.py`:

```
python
```

```python
#!/usr/bin/env python3
import sqlite3
from datetime import datetime

def update_database():
    """Add classification tables to existing database"""
    conn = sqlite3.connect('../data/smartguard.db')
    cursor = conn.cursor()

    print("🔄 Updating database schema...")

    # Domain classifications table
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS domain_classifications (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            domain TEXT UNIQUE NOT NULL,
            category TEXT NOT NULL,
            confidence REAL NOT NULL,
            risk_level TEXT NOT NULL,
            color TEXT NOT NULL,
            timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
            INDEX(domain),
            INDEX(category),
            INDEX(risk_level)
        )
    ''')
    print("✅ Created domain_classifications table")

    # Alert rules table
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS alert_rules (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            rule_name TEXT NOT NULL,
            category TEXT NOT NULL,
            risk_level TEXT NOT NULL,
            time_threshold INTEGER DEFAULT 300,
            request_threshold INTEGER DEFAULT 10,
            enabled BOOLEAN DEFAULT 1,
            created_at DATETIME DEFAULT CURRENT_TIMESTAMP
        )
    ''')
    print("✅ Created alert_rules table")

    # Alerts table
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS alerts (
```

```
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        alert_type TEXT NOT NULL,
        severity TEXT NOT NULL,
        client_ip TEXT NOT NULL,
        domain TEXT NOT NULL,
        category TEXT NOT NULL,
        message TEXT NOT NULL,
        acknowledged BOOLEAN DEFAULT 0,
        created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
        INDEX(client_ip),
        INDEX(severity),
        INDEX(acknowledged)
    )
    ''')
    print("✅ Created alerts table")

    # Insert default alert rules
    default_rules = [
        ("High Risk Sites", "inappropriate", "high", 60, 1, 1),
        ("Excessive Social Media", "social_media", "medium", 1800, 20, 1),
        ("Late Night Gaming", "gaming", "medium", 300, 5, 1),
        ("Educational Override", "educational", "low", 86400, 100, 0),
    ]

    for rule in default_rules:
        cursor.execute('''
            INSERT OR IGNORE INTO alert_rules
            (rule_name, category, risk_level, time_threshold, request_threshold, enabled)
            VALUES (?, ?, ?, ?, ?, ?)
        ''', rule)

    print(f"✅ Inserted {len(default_rules)} default alert rules")

    # Update configuration table
    cursor.execute('''
        INSERT OR REPLACE INTO configuration (key, value, description)
        VALUES ('classification_enabled', 'true', 'AI classification enabled')
    ''')

    print("✅ Updated configuration")

    conn.commit()
    conn.close()

    print("🎉 Database schema updated successfully!")

def verify_database():
```

```
    """Verify database schema"""
    conn = sqlite3.connect('../data/smartguard.db')
    cursor = conn.cursor()

    # Check tables
    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'
    ]

    print("\n📊 Database Schema Verification:")
    for table in required_tables:
        status = "✅" if table in tables else "❌"
        print(f"{status} {table}")

    # Check alert rules count
    cursor.execute("SELECT COUNT(*) FROM alert_rules")
    rule_count = cursor.fetchone()[0]
    print(f"\n📋 Alert Rules: {rule_count} configured")

    conn.close()

if __name__ == '__main__':
    update_database()
    verify_database()
```

3.2 Run Database Update

bash

```
cd ~/smartguard-monitor/src
source ../venv/bin/activate

# Update database schema
python3 update_database.py

# Verify tables were created
sqlite3 ../data/smartguard.db ".tables"
```

Expected Output:

🔄 Updating database schema...
✅ Created domain_classifications table
✅ Created alert_rules table
✅ Created alerts table
✅ Inserted 4 default alert rules
✅ Updated configuration
🎉 Database schema updated successfully!

📊 Database Schema Verification:
✅ dns_requests
✅ devices
✅ traffic_stats
✅ system_events
✅ configuration
✅ domain_classifications
✅ alert_rules
✅ alerts

📋 Alert Rules: 4 configured

Testing Procedures

4.1 Setup Testing Environment

bash

```
cd ~/smartguard-monitor

# Create test script from the provided code
nano test_step3.py
# Copy the testing script content here

# Make executable
chmod +x test_step3.py

# Ensure virtual environment is active
source venv/bin/activate
```

4.2 Run Comprehensive Tests

bash

```
# Run all Step 3 tests
python3 test_step3.py
```

4.3 Individual Test Commands

Test Database Schema:

bash

```
sqlite3 data/smartguard.db "SELECT name FROM sqlite_master WHERE type='table'"
```

Test Ollama Service:

bash

```
curl -X POST http://localhost:11434/api/generate \
  -H "Content-Type: application/json" \
  -d '{
    "model": "phi3:3.8b-mini-instruct-q4_K_M",
    "prompt": "Classify facebook.com",
    "stream": false
  }'
```

Test Domain Classification:

bash

```
python3 -c "
from domain_classifier import DomainClassifier
classifier = DomainClassifier()
result = classifier.classify_domain('facebook.com')
print(f'Result: {result}')
"
```

4.4 Performance Testing

bash

```
# Test classification speed
python3 -c "
import time
from domain_classifier import DomainClassifier

classifier = DomainClassifier()
domains = ['facebook.com', 'google.com', 'github.com', 'stackoverflow.com']

for domain in domains:
    start = time.time()
    result = classifier.classify_domain(domain)
    duration = time.time() - start
    print(f'{domain}: {result[\"category\"]} ({duration:.2f}s)')
"
```

4.5 Expected Test Results

✅ **All tests should pass with these benchmarks:**

- **Database Schema**: All 8 tables present
- **Ollama Service**: Responding on port 11434
- **Ollama Inference**: Returns valid classification
- **Domain Classifier**: >75% accuracy on test domains
- **Classification Caching**: 2nd request >50% faster
- **Database Operations**: Insert/read/delete working
- **Performance**: <5s average, <10s maximum
- **Error Handling**: Graceful handling of invalid domains

Troubleshooting

Common Issues and Solutions

1. Ollama Installation Issues

bash

```
# Check if Ollama is installed
which ollama

# If not found, reinstall
curl -fsSL https://ollama.com/install.sh | sh

# Check system architecture
uname -m  # Should show aarch64 for Pi 4
```

2. Model Download Problems

```
bash

# Check available space
df -h

# Manual model download
ollama pull phi3:3.8b-mini-instruct-q4_K_M

# If download fails, try smaller model
ollama pull phi3:3.8b-mini-instruct-q4_0
```

3. Memory Issues

```
bash

# Check memory usage
free -h

# Monitor during classification
htop

# If low memory, reduce concurrent requests
# Edit domain_classifier.py to add delays
```

4. Database Issues

```
bash
```

```bash
# Check database integrity
sqlite3 data/smartguard.db "PRAGMA integrity_check;"

# Backup database
cp data/smartguard.db data/smartguard_backup.db

# Reset database if corrupted
rm data/smartguard.db
python3 setup_database.py
python3 update_database.py
```

5. Network Connectivity

```bash
# Test Ollama API
curl http://localhost:11434/api/tags

# Check if service is running
sudo systemctl status ollama

# Restart if needed
sudo systemctl restart ollama
```

Performance Optimization

1. Model Optimization

```bash
# Use smaller quantized model if performance is poor
ollama pull phi3:3.8b-mini-instruct-q4_0

# Monitor resource usage
iostat -x 1
```

2. Caching Strategy

```python
```

```
# Increase cache duration in domain_classifier.py
# Change from 7 days to 30 days for stable domains
cursor.execute('''
    SELECT category, confidence, risk_level, color, timestamp
    FROM domain_classifications
    WHERE domain = ? AND timestamp > datetime('now', '-30 days')
''', (domain,))
```

3. Batch Processing

```
python

# Process multiple domains in single request
def classify_batch(domains):
    prompt = f"Classify these domains: {', '.join(domains)}"
    # Implementation for batch processing
```

Success Criteria

✅ Step 1 Complete When (AI Runtime Layer):

- Ollama service running and responding
- Phi-3-mini model downloaded and functional
- Test inference returns valid responses
- Service enabled to start on boot

✅ Step 2 Complete When (AI Application Layer):

- Domain classifier module created
- Classification returns 5 categories correctly
- Caching system functional
- Error handling working for edge cases

✅ Step 3 Complete When (AI Data Layer):

- Database schema updated with 3 new tables
- Default alert rules inserted
- Configuration updated
- All database operations working

✅ Overall AI-Stack Setup Complete When:

- All individual tests pass

- Overall test suite achieves >90% pass rate
- Performance meets benchmarks (<5s average)
- System ready for Step 4 (Alert System Integration)

Next Steps

Once Steps 1-3 are complete and all tests pass:

1. **Step 4**: Implement Alert System
2. **Step 5**: Enhanced DNS Monitor Integration
3. **Step 6**: Updated Web Dashboard
4. **Step 7**: End-to-End Testing
5. **Step 8**: Performance Optimization

Appendix

A. File Structure After Step 3

```
smartguard-monitor/
├── src/
│   ├── domain_classifier.py      # NEW
│   ├── update_database.py        # NEW
│   ├── test_step3.py             # NEW
│   ├── setup_database.py
│   ├── dns_monitor.py
│   └── web_dashboard.py
├── data/
│   └── smartguard.db             # UPDATED SCHEMA
├── logs/
│   ├── step3_testing.log         # NEW
│   └── step3_test_report.json    # NEW
└── venv/
```

B. Database Schema Changes

sql

```
-- New tables added in Step 3:
CREATE TABLE domain_classifications (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    domain TEXT UNIQUE NOT NULL,
    category TEXT NOT NULL,
    confidence REAL NOT NULL,
    risk_level TEXT NOT NULL,
    color TEXT NOT NULL,
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE alert_rules (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    rule_name TEXT NOT NULL,
    category TEXT NOT NULL,
    risk_level TEXT NOT NULL,
    time_threshold INTEGER DEFAULT 300,
    request_threshold INTEGER DEFAULT 10,
    enabled BOOLEAN DEFAULT 1,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE alerts (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    alert_type TEXT NOT NULL,
    severity TEXT NOT NULL,
    client_ip TEXT NOT NULL,
    domain TEXT NOT NULL,
    category TEXT NOT NULL,
    message TEXT NOT NULL,
    acknowledged BOOLEAN DEFAULT 0,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

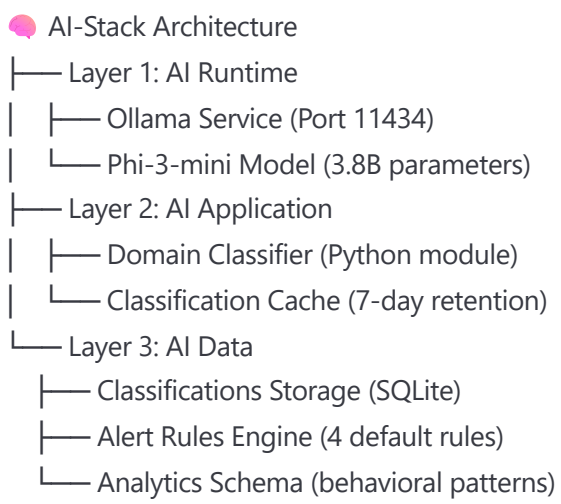
C. Default Alert Rules

1. **High Risk Sites**: Inappropriate content, 1 request in 60s
2. **Excessive Social Media**: 20 requests in 30 minutes
3. **Late Night Gaming**: 5 requests in 5 minutes
4. **Educational Override**: Disabled by default

Document End

AI-Stack Summary

Once complete, your SmartParent system will have a full AI infrastructure:



Capabilities Unlocked:

- Real-time domain classification (5 categories)
- Intelligent risk assessment (low/medium/high)
- Contextual understanding (educational vs inappropriate)
- Behavioral pattern detection
- Automated alert generation
- Privacy-preserving local processing

This document covers the complete implementation of SmartParent AI-Stack Setup, providing the foundation for intelligent parental monitoring capabilities. Follow this guide sequentially to ensure proper system functionality before proceeding to Alert System Integration.