Praktische Beispiele: Lokale KI für dein moOde-Projekt

Beispiel 1: Netzwerk-Konfiguration Agent

Problem

Du hast immer wieder Netzwerk-Probleme mit dem Pi. Du möchtest einen Agenten, der:
• Die Konfiguration automatisch prüft
• Probleme findet
• Fixes vorschlägt
• Fixes anwendet

Lösung: Network Config Agent

Agent-Definition

```
Name: network-config-agent
Beschreibung: Prüft und fixt Netzwerk-Konfiguration automatisch

Tools:
  - read_file: Liest Konfigurationsdateien
  - check_database: Prüft moOde Datenbank
  - test_connection: Testet Netzwerk-Verbindung
  - apply_fix: Wendet Fixes an

Knowledge Base:
  - network.php
  - network-mode-manager.sh
  - Ethernet.nmconnection examples
  - cfg_network schema
```

Prompt

```
Du bist ein Netzwerk-Konfiguration Experte für moOde Audio.

Deine Aufgabe:
1. Prüfe die aktuelle Netzwerk-Konfiguration
2. Finde Probleme (fehlende Dateien, falsche Werte, etc.)
3. Erstelle Fixes basierend auf der Knowledge Base
4. Teste Fixes bevor du sie anwendest
5. Wende Fixes an (mit Backup)

Verwende die Tools um:
- Konfigurationsdateien zu lesen
- Datenbank zu prüfen
- Verbindungen zu testen
- Fixes anzuwenden
```

Verwendung

```
# Agent ausführen
agent: network-config-agent
task: "Prüfe die Netzwerk-Konfiguration und fixe alle Probleme"

# Agent macht automatisch:
1. Liest /etc/NetworkManager/system-connections/*
2. Prüft cfg_network in der Datenbank
3. Findet: Ethernet.nmconnection fehlt
4. Erstellt Fix: Setze Werte in Datenbank
5. Testet: Prüft ob cfgNetworks() die Datei erstellt
6. Wendet Fix an
```

Beispiel 2: Code-Dokumentation Generator

Problem

Deine Scripts haben keine oder unvollständige Dokumentation. Du möchtest automatisch:
• README.md generieren
• Inline-Kommentare hinzufügen
• Funktionen dokumentieren

Lösung: Documentation Agent

Agent-Definition

```
Name: documentation-agent
Beschreibung: Generiert automatisch Code-Dokumentation

Tools:
  - analyze_code: Analysiert Code-Struktur
  - read_file: Liest Code-Dateien
  - write_file: Schreibt Dokumentation
  - update_readme: Aktualisiert README.md
```

Prompt

```
Du bist ein Dokumentations-Experte.

Deine Aufgabe:
1. Analysiere Code-Dateien
2. Extrahiere:
   - Funktionen und ihre Parameter
   - Script-Zweck
   - Abhängigkeiten
   - Verwendung
3. Generiere:
   - README.md mit Übersicht
   - Inline-Kommentare für komplexe Stellen
   - Funktion-Dokumentation

Folge dem Projekt-Stil:
- Markdown-Format
- Ausführliche Beschreibungen
- Code-Beispiele
- Troubleshooting-Sektionen
```

Verwendung

```
# Dokumentation für ein Script generieren
agent: documentation-agent
task: "Generiere Dokumentation für scripts/deployment/DEPLOY.sh"

# Agent macht:
1. Liest das Script
2. Analysiert Funktionen
3. Generiert README.md
4. Fügt Inline-Kommentare hinzu
5. Erstellt Usage-Beispiele
```

Beispiel 3: Bug-Detector & Auto-Fix

Problem

Du findest Bugs erst spät. Du möchtest:
• Bugs automatisch finden
• Root Cause analysieren
• Fixes automatisch generieren

Lösung: Bug Detector Agent

Agent-Definition

```
Name: bug-detector-agent
Beschreibung: Findet und fixt Bugs automatisch

Tools:
  - code_analyzer: Analysiert Code auf Fehler
  - pattern_matcher: Findet bekannte Problem-Patterns
  - test_runner: Führt Tests aus
  - git_log: Prüft Git-Historie für ähnliche Fixes
  - apply_fix: Wendet Fixes an
```

Prompt

```
Du bist ein Bug-Detection Experte.

Deine Aufgabe:
1. Analysiere Code auf:
   - Syntax-Fehler
   - Logik-Fehler
   - Sicherheitsprobleme
   - Performance-Issues
2. Finde Root Cause
3. Suche ähnliche Fixes in der Historie
4. Erstelle Fix
5. Teste Fix
6. Wende Fix an (mit Backup)

Sei vorsichtig:
- Erstelle immer Backup
- Teste vor dem Anwenden
- Dokumentiere Änderungen
```

Verwendung

```
# Bug in einem Script finden und fixen
agent: bug-detector-agent
task: "Finde und fixe Bugs in scripts/fixes/FIX_NETWORK.sh"

# Agent macht:
1. Analysiert Code
2. Findet: Fehlende Fehlerbehandlung bei sqlite3
3. Root Cause: Kein Error-Handling
4. Erstellt Fix: Füge Error-Handling hinzu
5. Testet Fix
6. Wendet Fix an
```

Beispiel 4: Deployment Automation

Problem

Deployment ist manuell und fehleranfällig. Du möchtest:

• Automatisches Deployment
• Fehler-Prävention
• Rollback bei Problemen

Lösung: Deployment Agent

Agent-Definition

```
Name: deployment-agent
Beschreibung: Führt Deployment automatisch durch

Tools:
  - git_checkout: Checkt Code aus
  - run_tests: Führt Tests aus
  - create_backup: Erstellt Backup
  - deploy_script: Führt Deployment-Script aus
  - verify_deployment: Verifiziert Deployment
  - rollback: Rollback bei Fehlern
```

Workflow

```
Deployment Workflow:
  1. Code-Review
     - Prüfe Änderungen
     - Finde potentielle Probleme
  2. Tests
     - Führe alle Tests aus
     - Prüfe Code-Qualität
  3. Backup
     - Erstelle Backup der aktuellen Version
     - Speichere Konfiguration
  4. Deployment
     - Kopiere Dateien
     - Setze Permissions
     - Starte Services neu
  5. Verifizierung
     - Prüfe Services
     - Teste Funktionalität
  6. Rollback (bei Fehlern)
     - Stelle Backup wieder her
     - Revertiere Änderungen
```

Verwendung

```
# Deployment durchführen
agent: deployment-agent
task: "Deployiere die neuesten Änderungen auf den Pi"

# Agent macht automatisch:
1. Reviewt Änderungen
2. Führt Tests aus
3. Erstellt Backup
4. Deployiert
5. Verifiziert
6. Rollback bei Fehlern
```

Beispiel 5: Code-Review Pipeline

Problem

Code-Reviews sind zeitaufwendig. Du möchtest:
• Automatische Reviews
• Konsistente Qualität
• Best Practices durchsetzen

Lösung: Code-Review Pipeline

Multi-Agent System

```
Pipeline:
  1. Code Reviewer Agent
     - Prüft Funktionalität
     - Findet Bugs
     - Prüft Best Practices

  2. Security Agent
     - Prüft Sicherheit
     - Findet Vulnerabilities
     - Prüft Permissions

  3. Performance Agent
     - Analysiert Performance
     - Findet Bottlenecks
     - Schlägt Optimierungen vor

  4. Documentation Agent
     - Prüft Dokumentation
     - Generiert fehlende Docs
     - Verbessert Kommentare

  5. Coordinator Agent
     - Sammelt alle Reviews
     - Konsolidiert Ergebnisse
     - Erstellt finalen Report
```

Verwendung

```
# Code-Review durchführen
agent: code-review-pipeline
task: "Review scripts/deployment/DEPLOY.sh"

# Pipeline macht:
1. Code Reviewer: Findet 3 Probleme
2. Security Agent: Findet 1 Security-Issue
3. Performance Agent: Schlägt 2 Optimierungen vor
4. Documentation Agent: Generiert Dokumentation
5. Coordinator: Erstellt finalen Report mit allen Findings
```

Beispiel 6: RAG: Projekt-spezifische Fragen

Setup

```
# 1. Lade alle Projekt-Dateien hoch
- docs/*.md
- scripts/**/*.sh
- moode-source/**/*
- README.md

# 2. Aktiviere RAG in Open WebUI
```

Fragen die du stellen kannst

```
# Netzwerk-Fragen
"Wie funktioniert die Netzwerk-Konfiguration?"
"Warum werden NetworkManager-Dateien gelöscht?"
"Wie setze ich eine statische IP?"

# Wizard-Fragen
"Wie funktioniert der Room EQ Wizard?"
"Wie wird pink noise gestartet?"
"Wie werden EQ-Filter angewendet?"
# Deployment-Fragen
```

```
"Wie deploye ich Änderungen auf den Pi?"
"Welche Scripts gibt es für Deployment?"
"Wie teste ich vor dem Deployment?"

# Code-Fragen
"Wie funktioniert network-mode-manager.sh?"
"Was macht cfgNetworks()?"
"Wie wird CamillaDSP konfiguriert?"
```

Die KI antwortet basierend auf deinen tatsächlichen Dateien!

Beispiel 7: Fine-tuning auf deinen Code-Stil

Trainingsdaten sammeln

```
# Sammle Code-Beispiele
find ~/moodeaudio-cursor -name "*.sh" -type f > training_files.txt

# Extrahiere Funktionen und Patterns
grep -r "function\|log()\|set -e" scripts/ > patterns.txt
```

Modelfile erstellen

```
cat > Modelfile <<EOF
FROM llama3.2:3b

SYSTEM """
Du bist ein Experte für Shell-Scripting und moOde Audio Konfiguration.

Dein Code-Stil:
- Verwendet bash
- Enthält Fehlerbehandlung: set -e
- Hat ausführliche Kommentare mit ##
- Verwendet log() Funktion für Logging
- Folgt den Projekt-Konventionen
- Verwendet absolute Pfade: cd ~/moodeaudio-cursor && ...
- Erstellt Backup vor Änderungen
- Prüft ob Dateien existieren
- Verwendet sqlite3 für Datenbank-Zugriffe
"""

TEMPLATE """{{ .Prompt }}

{{ .Response }}"""
EOF

# Erstelle das Modell
ollama create my-moode-style -f Modelfile
```

Verwenden

```
# Teste das fine-tuned Modell
ollama run my-moode-style "Erstelle ein Script, das die Netzwerk-Konfiguration prüft"

# Das Modell generiert Code in deinem Stil!
```

Beispiel 8: Custom Tool: Network Checker

Tool erstellen

```
# tools/network_checker.py
#!/usr/bin/env python3
```

```
import subprocess
import json
import sys

def check_network_config():
    """Prüft Netzwerk-Konfiguration"""
    results = {
        'ethernet_file_exists': False,
        'database_config': None,
        'connection_status': None
    }

    # Prüfe Ethernet.nmconnection
    import os
    if os.path.exists('/etc/NetworkManager/system-connections/Ethernet.nmconnection'):
        results['ethernet_file_exists'] = True

    # Prüfe Datenbank
    import sqlite3
    db = sqlite3.connect('/var/local/www/db/moode-sqlite3.db')
    cursor = db.cursor()
    cursor.execute("SELECT * FROM cfg_network WHERE iface = 'eth0'")
    row = cursor.fetchone()
    if row:
        results['database_config'] = {
            'method': row[2],
            'ipaddr': row[3],
            'gateway': row[5]
        }

    return results

if __name__ == '__main__':
    results = check_network_config()
    print(json.dumps(results, indent=2))
```

Tool in Agent integrieren

```
Tools:
  - name: check_network
    description: "Prüft Netzwerk-Konfiguration"
    command: "python3 tools/network_checker.py"
    returns: JSON
```

Beispiel 9: Workflow: Neues Feature entwickeln

Workflow-Definition

```
Workflow: new-feature-development
Beschreibung: Entwickelt ein neues Feature von Anfang bis Ende

Schritte:
  1. Planning
     - Feature-Request analysieren
     - Anforderungen definieren
     - Architektur planen

  2. Development
     - Code generieren
     - Tests schreiben
     - Dokumentation erstellen

  3. Review
     - Code reviewen
     - Tests ausführen
     - Qualität prüfen

  4. Deployment
     - Deployment vorbereiten
     - Testen
     - Deployieren
```

Verwendung

```
# Neues Feature entwickeln
workflow: new-feature-development
feature: "Automatische Netzwerk-Diagnose"

# Workflow macht:
1. Analysiert Anforderungen
2. Plant Architektur
3. Generiert Code
4. Schreibt Tests
5. Reviewt Code
6. Deployiert
```

Beispiel 10: Monitoring Agent

Problem

Du möchtest wissen, ob alles funktioniert, ohne manuell zu prüfen.

Lösung: Monitoring Agent

```
Name: monitoring-agent
Beschreibung: Überwacht System und meldet Probleme

Tools:
  - check_services: Prüft ob Services laufen
  - check_network: Prüft Netzwerk-Verbindung
  - check_disk_space: Prüft Festplattenplatz
  - check_logs: Prüft Logs auf Fehler

Schedule: Alle 5 Minuten
```

Verwendung

```
# Agent läuft automatisch alle 5 Minuten
# Meldet Probleme automatisch
# Kann auch automatisch fixen (wenn konfiguriert)
```

Zusammenfassung

Diese Beispiele zeigen, wie du die lokale KI für dein moOde-Projekt nutzen kannst:

1. **Network Config Agent** - Automatische Netzwerk-Fixes
2. **Documentation Agent** - Automatische Dokumentation
3. **Bug Detector** - Automatische Bug-Fixes
4. **Deployment Agent** - Automatisches Deployment
5. **Code-Review Pipeline** - Automatische Reviews
6. **RAG** - Projekt-spezifische Fragen
7. **Fine-tuning** - Code in deinem Stil
8. **Custom Tools** - Eigene Tools integrieren
9. **Workflows** - Komplexe Prozesse automatisieren
10. **Monitoring** - Kontinuierliche Überwachung

Viel Erfolg beim Aufbau deiner lokalen KI! ■