Erweiterte Features: Lokale KI auf dem Mac

Übersicht der erweiterten Features

1. **RAG (Retrieval Augmented Generation)** - KI lernt aus deinen Dateien
2. **Agenten erstellen** - KI arbeitet selbstständig
3. **Fine-tuning** - KI lernt deinen Code-Stil
4. **Tools & Funktionen** - KI kann Dateien lesen/schreiben, Commands ausführen
5. **Multi-Agent Systeme** - Mehrere Agenten arbeiten zusammen
6. **Custom Prompts & Templates** - Wiederverwendbare Prompt-Vorlagen
7. **Code-Analyse & Auto-Fixes** - Automatische Code-Verbesserungen

---

1. RAG: KI lernt aus deinen Dateien

Was ist RAG?

RAG (Retrieval Augmented Generation) ermöglicht es der KI, aus deinen eigenen Dokumenten und Code-Dateien zu lernen. Die KI kann dann Fragen zu deinen spezifischen Projekten beantworten.

Setup in Open WebUI

Schritt 1: RAG aktivieren

1. Öffne Open WebUI: `http://localhost:3000`
2. Gehe zu **Settings** → **RAG**
3. Aktiviere **"Enable RAG"**
4. Wähle **"Vector Database"** (z.B. ChromaDB oder Qdrant)

Schritt 2: Dokumente hochladen

1. Gehe zu **"Knowledge"** oder **"RAG"** Tab
2. Klicke auf **"Upload Documents"**
3. Lade deine Dateien hoch:
• Code-Dateien (`.py`, `.sh`, `.js`, etc.)
• Dokumentation (`.md`, `.txt`)
• Konfigurationsdateien
• Projekt-Dokumentation

Schritt 3: Verwenden

```
Frage: "Wie funktioniert die Netzwerk-Konfiguration in meinem Projekt?"
```

Die KI durchsucht deine hochgeladenen Dateien und gibt eine Antwort basierend auf deinem Code.

Erweiterte RAG-Konfiguration

Chunking-Einstellungen

```
# In Open WebUI Settings → RAG
Chunk Size: 512        # Größe der Text-Blöcke
Chunk Overlap: 50      # Überlappung zwischen Blöcken
Top K Results: 5       # Anzahl der relevanten Dokumente
```

Metadaten hinzufügen

• Tagge Dokumente mit Kategorien (z.B. "network", "audio", "wizard")
• Füge Beschreibungen hinzu
• Organisiere nach Projekten

Praktisches Beispiel: moOde-Projekt

```
# 1. Lade alle Scripts hoch
# 2. Lade Dokumentation hoch
# 3. Frage die KI:

"Wie funktioniert die Netzwerk-Konfiguration?"
"Welche Scripts gibt es für den Wizard?"
"Wie wird CamillaDSP konfiguriert?"
```

Die KI antwortet basierend auf deinen tatsächlichen Dateien!

---

2. Agenten erstellen

Was sind Agenten?

Agenten sind KI-Assistenten, die selbstständig Aufgaben erledigen können. Sie können:
• Dateien lesen und schreiben
• Commands ausführen
• Entscheidungen treffen
• Mehrere Schritte ausführen

Einfacher Agent: Code-Reviewer

Schritt 1: Agent erstellen

1. In Open WebUI: **"Create"** → **"Agent"**
2. Name: `code-reviewer`
3. Beschreibung: `Reviewt Code automatisch und schlägt Verbesserungen vor`

Schritt 2: Tools hinzufügen

```
Tools:
  - name: read_file
    description: "Liest eine Datei und gibt den Inhalt zurück"
    command: "cat {file_path}"

  - name: write_file
    description: "Schreibt Inhalt in eine Datei"
    command: "echo '{content}' > {file_path}"

  - name: run_command
    description: "Führt einen Command aus"
    command: "{command}"
```

Schritt 3: Prompt definieren

```
Du bist ein Code-Reviewer. Deine Aufgabe:

1. Lies die angegebene Datei
2. Analysiere den Code auf:
   - Fehler
   - Best Practices
   - Verbesserungsmöglichkeiten
   - Sicherheitsprobleme
3. Erstelle einen Review-Report
4. Schlage konkrete Fixes vor

Verwende die Tools, um Dateien zu lesen und Verbesserungen zu schreiben.
```

Schritt 4: Agent verwenden

```
Agent: code-reviewer
Aufgabe: "Review die Datei scripts/deployment/DEPLOY.sh"
```

Der Agent:
1. Liest die Datei automatisch
2. Analysiert sie
3. Erstellt einen Review-Report
4. Schlägt Fixes vor

Erweiterter Agent: Auto-Fix Agent

```
Name: auto-fix-agent
Beschreibung: Findet und fixt Code-Fehler automatisch

Tools:
  - read_file
  - write_file
  - run_command
  - git_diff
  - run_tests

Prompt: |
  Du bist ein Auto-Fix Agent. Deine Aufgabe:

  1. Analysiere Code auf Fehler
  2. Finde die Ursache
  3. Erstelle einen Fix
  4. Teste den Fix
  5. Wende den Fix an (wenn Tests bestehen)

  Sei vorsichtig: Backup erstellen vor Änderungen!
```

Multi-Step Agent: Deployment Agent

```
Name: deployment-agent
Beschreibung: Führt Deployment-Schritte automatisch aus

Workflow:
  1. Code-Review
  2. Tests ausführen
  3. Backup erstellen
  4. Deployment durchführen
  5. Verifizieren
  6. Rollback bei Fehlern

Tools:
  - git_checkout
  - run_tests
  - create_backup
  - deploy_script
  - verify_deployment
  - rollback
```

---

3. Fine-tuning: KI lernt deinen Code-Stil

Was ist Fine-tuning?

Fine-tuning trainiert die KI auf deinen spezifischen Code-Stil, deine Konventionen und deine Projekte. Die KI wird dann besser darin, Code in deinem Stil zu generieren.

Vorbereitung: Trainingsdaten erstellen

Schritt 1: Code-Beispiele sammeln

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

Schritt 2: Trainingsdaten formatieren

```
{"prompt": "Erstelle ein Script, das die Netzwerk-Konfiguration prüft", "completion": "#!/bin/bash\n#
{"prompt": "Wie prüfe ich ob ein Service läuft?", "completion": "systemctl is-active service-name"}
{"prompt": "Erstelle eine Funktion für Logging", "completion": "log() {\n    echo \"[$(date '+%Y-%m-%
```

Fine-tuning mit Ollama

Schritt 1: Modelfile erstellen

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

# System prompt für deinen Code-Stil
SYSTEM """
Du bist ein Experte für Shell-Scripting und moOde Audio Konfiguration.
Dein Code-Stil:
- Verwendet bash
- Enthält Fehlerbehandlung (set -e)
- Hat ausführliche Kommentare
- Folgt den Projekt-Konventionen
- Verwendet absolute Pfade wenn nötig
"""

# Trainingsdaten
TEMPLATE """{{ .Prompt }}

{{ .Response }}"""
EOF
```

Schritt 2: Fine-tuned Modell erstellen

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

# Teste es
ollama run my-code-style "Erstelle ein Script für Netzwerk-Check"
```

Erweiterte Fine-tuning-Methoden

LoRA (Low-Rank Adaptation)

```
# Installiere LoRA-Tools
pip install peft transformers
```

```
# Trainiere LoRA-Adapter
python train_lora.py \
  --model llama3.2:3b \
  --data training_data.jsonl \
  --output my-code-style-lora
```

QLoRA (Quantized LoRA)

• Spart Speicher
• Schnelleres Training
• Gute Ergebnisse auch mit weniger Daten

---

4. Tools & Funktionen

Custom Tools erstellen

Tool: File Analyzer

```
# tools/file_analyzer.py
def analyze_file(file_path):
    """Analysiert eine Datei und gibt Metadaten zurück"""
    import os
    import subprocess

    stats = {
        'path': file_path,
        'size': os.path.getsize(file_path),
        'lines': sum(1 for _ in open(file_path)),
        'language': detect_language(file_path),
        'complexity': calculate_complexity(file_path)
    }
    return stats
```

Tool: Code Metrics

```
# tools/code_metrics.py
def get_code_metrics(directory):
    """Berechnet Code-Metriken für ein Verzeichnis"""
    metrics = {
        'total_files': count_files(directory),
        'total_lines': count_lines(directory),
        'functions': count_functions(directory),
        'complexity': calculate_complexity(directory)
    }
    return metrics
```

Tools in Open WebUI integrieren

Schritt 1: Tool definieren

```
# In Open WebUI Agent Settings
Tools:
  - name: analyze_project
    description: "Analysiert das gesamte Projekt"
    endpoint: "http://localhost:8000/analyze"
    method: POST
    parameters:
      - name: directory
        type: string
        required: true
```

Schritt 2: Tool-Server erstellen

```
# tool_server.py
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/analyze', methods=['POST'])
def analyze():
    directory = request.json.get('directory')
    # Analysiere das Projekt
    results = analyze_project(directory)
    return jsonify(results)

if __name__ == '__main__':
    app.run(port=8000)
```

---

5. Multi-Agent Systeme

Architektur

```
■■■■■■■■■■■■■■■■
■ Coordinator ■  ← Orchestriert alle Agenten
■■■■■■■■■■■■■■■■
        ■
        ■■■■■■ Code Reviewer Agent
        ■■■■■■ Test Runner Agent
        ■■■■■■ Documentation Agent
        ■■■■■■ Deployment Agent
```

Coordinator Agent

```
Name: coordinator-agent
Rolle: Orchestriert andere Agenten

Workflow:
  1. Analysiere Aufgabe
  2. Entscheide welche Agenten benötigt werden
  3. Weise Aufgaben zu
  4. Sammle Ergebnisse
  5. Konsolidiere Ergebnisse
  6. Erstelle finalen Report
```

Beispiel: Code-Review Pipeline

```
# multi_agent_pipeline.py
def code_review_pipeline(file_path):
    # 1. Code Reviewer Agent
    review = code_reviewer_agent.review(file_path)

    # 2. Security Agent
    security_issues = security_agent.scan(file_path)

    # 3. Performance Agent
    performance_tips = performance_agent.analyze(file_path)

    # 4. Documentation Agent
    docs = documentation_agent.generate(file_path)

    # 5. Coordinator konsolidiert
    final_report = coordinator.consolidate([
        review, security_issues, performance_tips, docs
    ])

    return final_report
```

---

6. Custom Prompts & Templates

Prompt-Templates erstellen

Template: Code-Review

```
# Code Review Template

**Datei:** {file_path}
**Autor:** {author}
**Datum:** {date}

**Review-Kriterien:**
- [ ] Funktionalität korrekt
- [ ] Fehlerbehandlung vorhanden
- [ ] Kommentare ausreichend
- [ ] Best Practices befolgt
- [ ] Tests vorhanden

**Gefundene Probleme:**
{issues}

**Verbesserungsvorschläge:**
{suggestions}
```

Template: Bug-Report

```
# Bug Report Template

**Beschreibung:** {description}
**Schritte zur Reproduktion:**
{steps}

**Erwartetes Verhalten:**
{expected}

**Tatsächliches Verhalten:**
{actual}

**Mögliche Lösung:**
{solution}
```

Templates in Open WebUI speichern

1. Gehe zu **"Prompts"** → **"Create Template"**
2. Füge dein Template ein
3. Speichere mit Tags (z.B. "code-review", "bug-report")
4. Verwende in Chats: `@template:code-review`

---

7. Code-Analyse & Auto-Fixes

Automatische Code-Analyse

AST-Analyse (Abstract Syntax Tree)

```
# code_analyzer.py
import ast
def analyze_code(code):
```

```
    tree = ast.parse(code)

    issues = []
    for node in ast.walk(tree):
        # Finde potentielle Probleme
        if isinstance(node, ast.Call):
            if is_dangerous_function(node):
                issues.append({
                    'type': 'security',
                    'line': node.lineno,
                    'message': 'Potentiell gefährliche Funktion'
                })

    return issues
```

Pattern-Matching

```
# pattern_matcher.py
import re

patterns = {
    'hardcoded_password': r'password\s*=\s*["\'][^"\']+["\']',
    'sql_injection': r'SELECT.*\+.*\$',
    'command_injection': r'subprocess\.call\(.*\+.*\)'
}

def find_patterns(code):
    issues = []
    for pattern_name, pattern in patterns.items():
        matches = re.findall(pattern, code)
        if matches:
            issues.append({
                'type': pattern_name,
                'matches': matches
            })
    return issues
```

Auto-Fix Agent

```
Name: auto-fix-agent
Beschreibung: Findet und fixt Code-Probleme automatisch

Workflow:
  1. Analysiere Code
  2. Finde Probleme
  3. Erstelle Fixes
  4. Teste Fixes
  5. Wende Fixes an (mit Backup)

Tools:
  - code_analyzer
  - pattern_matcher
  - test_runner
  - git_backup
  - apply_fix
```

Praktisches Beispiel

```
# Agent ausführen
agent: auto-fix-agent
task: "Analysiere scripts/deployment/DEPLOY.sh und fixe alle Probleme"

# Agent macht:
1. Liest die Datei
2. Findet Probleme:
   - Fehlende Fehlerbehandlung
   - Hardcoded Pfade
   - Fehlende Kommentare
3. Erstellt Fixes
4. Testet Fixes
5. Wendet Fixes an (mit Backup)
```

---

8. Praktische Workflows

Workflow 1: Neues Feature entwickeln

```
Workflow: feature-development
Schritte:
  1. Feature-Request analysieren
  2. Code-Struktur planen
  3. Code generieren
  4. Tests schreiben
  5. Code reviewen
  6. Dokumentation erstellen
  7. Deployment vorbereiten

Agenten:
  - planner-agent
  - code-generator-agent
  - test-generator-agent
  - reviewer-agent
  - documentation-agent
```

Workflow 2: Bug-Fix Pipeline

```
Workflow: bug-fix-pipeline
Schritte:
  1. Bug-Report analysieren
  2. Root Cause finden
  3. Fix entwickeln
  4. Tests schreiben
  5. Code reviewen
  6. Fix anwenden
  7. Verifizieren

Agenten:
  - bug-analyzer-agent
  - fix-generator-agent
  - test-runner-agent
  - reviewer-agent
```

Workflow 3: Code-Refactoring

```
Workflow: refactoring
Schritte:
  1. Code analysieren
  2. Verbesserungen identifizieren
  3. Refactoring-Plan erstellen
  4. Schrittweise refactoren
  5. Tests nach jedem Schritt
  6. Dokumentation aktualisieren

Agenten:
  - code-analyzer-agent
  - refactoring-planner-agent
  - refactoring-executor-agent
  - test-runner-agent
```

---

9. Integration mit deinem Projekt

moOde-Projekt Integration

Schritt 1: Projekt-Dokumentation hochladen

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

Schritt 2: Custom Agenten erstellen

```
# moode-network-agent
Name: moode-network-agent
Beschreibung: Spezialisiert auf moOde Netzwerk-Konfiguration

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

Tools:
  - check_network_config
  - fix_network_issues
  - deploy_network_config
```

Schritt 3: Workflows definieren

```
# network-troubleshooting-workflow
1. Analysiere Netzwerk-Problem
2. Prüfe Konfiguration
3. Finde Root Cause
4. Erstelle Fix
5. Teste Fix
6. Wende Fix an
```

---

10. Monitoring & Optimierung

Metriken tracken

```
# metrics.py
metrics = {
    'agent_success_rate': 0.95,
    'average_response_time': 1.2,
    'code_quality_improvement': 0.15,
    'bugs_found': 42,
    'bugs_fixed': 38
}
```

Performance optimieren

```
Optimierungen:
  - Model Caching
  - Response Streaming
  - Batch Processing
  - Parallel Agent Execution
```

Nächste Schritte

1. **Starte mit RAG** - Lade deine Dokumentation hoch
2. **Erstelle einen einfachen Agenten** - Code-Reviewer
3. **Experimentiere mit Fine-tuning** - Trainiere auf deinem Code

4. **Baue komplexere Workflows** - Multi-Agent Systeme

Viel Erfolg! ■