Dein WiFi-Skill: Startpunkt für einen funktionierenden OpenClaw Skill

Hey Alex,

das hier ist kein Rant und keine Kritik. Dein WiFi-Debugger hat echtes Potenzial — du hast Domain-Wissen über ASUS-Merlin-Router, die SSH-Befehle sind grösstenteils korrekt, und die Idee eines WiFi-Analyse-Skills ist genuinely nützlich.

Das Problem ist nur: der Code wird aktuell von OpenClaw nie aufgerufen. Der Skill sieht von aussen aus wie ein fertiges Produkt, aber es fehlt die Brücke zwischen "TypeScript-Klasse existiert" und "OpenClaw kann damit arbeiten".

Dieses Dokument beschreibt den Weg vom jetzigen Stand zu einem MVP — einem Minimal Viable Skill der tatsächlich läuft. Es geht erstmal nur darum, dass OpenClaw deinen Code überhaupt aufrufen kann. Es gibt danach noch einiges zu fixen (dazu am Ende mehr), aber der erste Schritt ist: überhaupt mal laufen.

Empfehlung wäre ehrlich gesagt, den Skill von Grund auf neu zu starten — kleiner, fokussierter, ein Feature das wirklich funktioniert statt 43 Actions die theoretisch existieren. Aber wenn du mit dem bestehenden Code weitermachen willst, hier ist was zu tun wäre.

Was du über OpenClaw Skills wissen musst

Bevor wir ins Konkrete gehen — das hier ist fundamental und erklärt warum dein Skill nicht funktioniert.

Ein Skill ist eine Anleitung, kein Programm

Ein OpenClaw Skill ist im Kern eine **Markdown-Datei** (`SKILL.md`) die dem LLM sagt was es tun soll. Wenn ein User "Mein WiFi ist langsam" schreibt, passiert folgendes:

1. OpenClaw scannt alle installierten Skills und wählt den passendsten aus
2. Das LLM liest die `SKILL.md` dieses Skills
3. Das LLM folgt den Instruktionen darin und führt **Shell-Befehle** aus (via `exec`-Tool)

Das ist direkt aus dem OpenClaw Source Code (`src/agents/system-prompt.ts`) — der System-Prompt den jeder Agent bekommt:

```
"## Skills (mandatory)"
"Before replying: scan <available_skills> <description> entries."
"- If exactly one skill clearly applies: read its SKILL.md at <location> with read, then follow it."
"- If multiple could apply: choose the most specific one, then read/follow it."
"- If none clearly apply: do not read any SKILL.md."
"Constraints: never read more than one skill up front; only read after selecting."
```

Zwei wichtige Punkte:

Erstens: Pro User-Anfrage wird **ein** Skill gewählt, nie mehrere gleichzeitig. Das heisst dein WiFi-Skill und deine Memory-Skills (Hippocampus, Amygdala, VTA) konkurrieren — und nur einer kann pro Nachricht aktiv sein.

Zweitens: Es gibt keinen Mechanismus der TypeScript-Klassen importiert oder npm-Module aufruft. Das LLM liest Text und führt Shell-Befehle aus. Das ist alles.

Wie Code in einem Skill ausführbar wird

Skills können eigene Tools mitbringen. Dafür gibt es den `bins/`-Ordner — Executables die dort liegen, werden automatisch in den PATH gelegt. Die SKILL.md sagt dem LLM dann "führe `mein-tool scan` aus" und das LLM macht `exec mein-tool scan`.

Alternativ: In der SKILL.md Metadata kann man deklarieren welches npm-Package installiert werden soll und welches Binary es bereitstellt:

```json
"install": [{ "kind": "node", "package": "mein-package", "bins": ["mein-binary"] }]
```

In beiden Fällen: **der Code muss ein CLI-Programm sein** das man in der Shell aufrufen kann. Nicht eine Library, nicht eine exportierte Klasse — ein Programm mit Entrypoint.

Wie TypeScript zu einem CLI wird

Falls du noch nie ein CLI aus TypeScript gemacht hast — der Ablauf ist:

1. Du schreibst eine `src/cli.ts` mit einem Shebang (`#!/usr/bin/env node`)
2. In `package.json` deklarierst du ein `"bin"`-Feld das auf die kompilierte Version zeigt
3. `npm run build` kompiliert TypeScript zu JavaScript (`dist/cli.js`)
4. `npm install -g .` oder `npm link` registriert das Binary im System-PATH
5. Danach kannst du `openclaw-wifi scan_network` im Terminal aufrufen

Dein bestehender TypeScript-Code bleibt dabei komplett wie er ist. Die `cli.ts` ist nur ein dünner Wrapper der die Klasse instanziiert und aufruft. Der Build-Prozess (`tsc`) den du schon hast, kompiliert alles zusammen.

Was aktuell nicht stimmt

Dein `src/index.ts`:

```typescript
export default OpenClawAsusMeshSkill;
```

Das exportiert eine Klasse. Aber niemand ruft sie auf. Kein `main()`, kein CLI-Entrypoint. Wenn jemand das Package installiert, bekommt er eine Library — aber kein Programm.

Deine SKILL.md Metadata:

```
json

"install": [{ "kind": "node", "package": "openclaw-asus-mesh-skill", "bins": [] }]
```

`"bins": []` — leer. Selbst wenn das Package installiert wird, wird kein Binary registriert. OpenClaw hat nichts das es aufrufen könnte.

Deine SKILL.md nennt Actions wie `scan_network` und `full_intelligence_scan`, aber erklärt dem LLM nicht **wie** es diese aufrufen soll. Es fehlen konkrete Shell-Befehle. Das LLM liest "führe `scan_network` aus" und weiss nicht was damit gemeint ist — es gibt kein Binary namens `scan_network`, keinen Befehl den es in die Shell tippen kann.

Die `skill.json` referenziert `"$schema": "https://openclaw.io/schemas/skill.json"` — diese URL existiert nicht. Das Schema ist erfunden.

Was zu tun wäre (MVP)

1. CLI-Entrypoint erstellen

Eine neue Datei `src/cli.ts` die als Brücke zwischen Kommandozeile und deiner Klasse dient:

```
typescript

#!/usr/bin/env node
import { OpenClawAsusMeshSkill } from './skill/openclaw-skill.js';
import { SkillActionSchema } from './skill/actions.js';

const action = process.argv[2];
const paramsRaw = process.argv[3];

if (!action) {
  console.error('Usage: openclaw-wifi <action> [params-json]');
process.exit(1);
}

const params = paramsRaw ? JSON.parse(paramsRaw) : {};
const parsed = SkillActionSchema.parse({ action, params });

const skill = new OpenClawAsusMeshSkill();
await skill.initialize();
const result = await skill.execute(parsed);
console.log(JSON.stringify(result, null, 2));
await skill.shutdown();
```

Das ist die Mindest-Version. Fehlerbehandlung, sauberer Shutdown bei Fehlern etc. kommen idealerweise noch dazu.

2. package.json: `bin`-Feld hinzufügen

```json
{
  "bin": {
    "openclaw-wifi": "./dist/cli.js"
  }
}
```

Und im Build-Script sicherstellen dass das Binary ausführbar ist:

```json
{
  "scripts": {
    "build": "tsc",
    "postbuild": "chmod +x dist/cli.js"
  }
}
```

3. State-Persistenz

Das ist dein grösstes Architektur-Problem für den CLI-Modus: Deine Klasse hält State im RAM. `meshState`, `zigbeeState`, `pendingOptimizations` — alles weg sobald der Prozess endet. Aber ein typischer Workflow ist mehrstufig: erst `scan_network`, dann `get_optimization_suggestions`.

Ohne State-Persistenz macht jeder einzelne CLI-Aufruf einen neuen SSH-Scan auf den Router. Das dauert jedes Mal mehrere Sekunden und belastet den Router unnötig.

Lösung: nach jedem Call den relevanten State als JSON auf Disk schreiben, beim nächsten Call wieder einlesen. Beispiel-Pfad: `~/.openclaw/skills/asus-mesh-wifi-analyzer/state.json`. Mit einem Timestamp versehen und nach z.B. 5-10 Minuten als veraltet betrachten. Du hast bereits eine `NetworkKnowledgeBase` die nach `./data/network-knowledge.json` schreibt — gleiches Prinzip, aber für den Session-State.

4. SKILL.md komplett umschreiben

Statt abstrakter Action-Namen braucht das LLM echte Shell-Befehle. Nicht alle 43 Actions — nur die die tatsächlich Sinn machen. Hier ein Entwurf (Details müssen an den echten Code angepasst werden):

```markdown
```

---
name: asus-mesh-wifi-analyzer
description: ASUS Mesh WiFi network analysis and optimization via SSH
user-invocable: true
metadata: { "openclaw": { "emoji": "📡", "os": ["darwin", "linux"], "requires": { "bins": ["openclaw-wifi", "ssh"], "env": ["A

 # ASUS Mesh WiFi Analyzer

CLI-Tool für ASUS Mesh WiFi Netzwerk-Analyse und Optimierung via SSH.
Alle Befehle geben JSON zurück. Parse das JSON und fasse es menschenlesbar zusammen.

Schnelldiagnose (immer damit starten)
```bash
openclaw-wifi get_quick_diagnosis
```

Netzwerk scannen
```bash
openclaw-wifi scan_network
```

Geräteliste
```bash
openclaw-wifi get_device_list
openclaw-wifi get_device_list '{"filter":"problematic"}'
```

Optimierungsvorschläge
```bash
openclaw-wifi get_optimization_suggestions
```

Zum Anwenden (erst User fragen!):
```bash
openclaw-wifi apply_optimization '{"suggestionId":"ID","confirm":true}'
```

Workflow

1. `get_quick_diagnosis` als Einstieg
2. Bei Problemen: `get_optimization_suggestions`
3. User fragen ob anwenden → `apply_optimization`
4. `scan_network` zum Verifizieren

Wichtig

```
- Vor `apply_optimization` IMMER den User fragen
- `restart_wireless` trennt alle WLAN-Clients — nur nach Bestätigung
```

Beachte: `"bins": ["openclaw-wifi"]` statt `"bins": []`. Und `win32` ist raus — dein Code nutzt `ssh` via `child_process.spawn` und `sshpass`, das gibt es auf Windows nicht nativ.

5. skill.json aufräumen

Die `skill.json` mit dem erfundenen Schema entfernen oder ersetzen:

```json
{
  "name": "asus-mesh-wifi-analyzer",
  "description": "ASUS Mesh WiFi analysis and optimization",
  "env": {
    "ASUS_ROUTER_HOST": "",
    "ASUS_ROUTER_SSH_USER": "",
    "ASUS_ROUTER_SSH_PASSWORD": ""
  }
}
```

Keine erfundenen Schemas, keine Permissions die nirgends geprüft werden.

So testest du ob es funktioniert

Nach dem Umbau:

```bash
# Build
npm run build

# Ohne Argumente → sollte Usage-Info zeigen
openclaw-wifi

# Ohne Router-Credentials → sollte saubere Fehlermeldung geben
openclaw-wifi scan_network

# Mit Credentials (wenn Router erreichbar)
ASUS_ROUTER_HOST=192.168.178.1 \
ASUS_ROUTER_SSH_USER=admin \
ASUS_ROUTER_SSH_PASSWORD=deinPasswort \ openclaw-
wifi scan_network
```

Wenn der letzte Befehl JSON auf stdout ausgibt — herzlichen Glückwunsch, dein Skill ist zum ersten Mal tatsächlich gelaufen.

Was danach noch ansteht (die wichtigsten Baustellen)

Der MVP oben macht den Skill aufrufbar. Aber im bestehenden Code gibt es einige Probleme die danach angegangen werden sollten. Hier die Top 5:

1. SSH-Fehler werden verschluckt (`src/infra/asus-ssh-client.ts`, `executeRaw()`, ca. Zeile 228-239): Wenn ein SSH-Befehl mit Exit-Code 1 oder 127 fehlschlägt, wird das als Erfolg gewertet. Die Parsing-Kette danach arbeitet mit leeren oder kaputten Daten, ohne dass jemand es merkt.

2. Interface-Namen sind hardcoded (`eth6`, `eth7`): Die echten Interface-Namen auf ASUS-Routern variieren je nach Modell. Müsste dynamisch ermittelt werden, z.B. via `nvram get wl0_ifname`.

3. SNMP-Parser wird in der Praxis scheitern (`src/infra/snmp-client.ts`): Der BER/ASN.1-Parser nutzt hardcodierte Byte-Offsets. BER-Encoding hat variable Längen — bei echten SNMP-Geräten wird der Parser an der falschen Stelle lesen.

4. Heatmap bekommt keine Daten (`src/core/heatmap-generator.ts`): Die Klasse wird ohne Argumente instanziiert und bekommt keine Referenz auf Scan-Daten oder Geräte-Positionen. Das Ergebnis ist eine leere Heatmap.

5. actionCount wird nie hochgezählt (`src/skill/openclaw-skill.ts`): `this.actionCount` bleibt immer 0. Die Metrik-Funktionen geben Fantasie-Zahlen zurück.

Zusammenfassung

Was du hast: ~17.000 Zeilen TypeScript die kompilieren und Tests bestehen.

Was fehlt: Die Brücke die OpenClaw tatsächlich nutzen kann.

Der Kern-Umbau ist überschaubar:

- Eine neue Datei (`cli.ts`) als Entrypoint
- Ein `bin`-Feld in `package.json`
- State-Persistenz auf Disk
- Eine SKILL.md mit echten Shell-Befehlen
- Eine bereinigte `skill.json`

Die bestehende Klasse, die Module, die Zod-Schemas — das alles bleibt. Es wird endlich benutzt statt nur exportiert.