ChatGPT

# Analyse der Agent-NN Codebasis

## Module: Vollständig implementiert und aktiv

Die grundlegenden **MCP-Microservices** sind weitgehend vorhanden und einsatzfähig. Dazu zählen insbesondere:

- **Task-Dispatcher** (Aufgaben-Koordinator) – verteilt eingehende Aufgaben an Worker-Agenten [1]. Der Dispatcher greift auf den **Agent Registry**-Service (Agentenregister) und den **Session Manager** (Kontextverwaltung in Redis) zu [2] [3]. Diese Kernmodule laufen bereits als eigenständige FastAPI-Services (siehe Docker-Compose) und bilden das Rückgrat der neuen Architektur [4] [5].
- **LLM-Gateway** – bietet eine einheitliche Schnittstelle zu Sprachmodellen [6] und ist implementiert (nutzt `LLMBackendManager` als Provider-Fabrik) [7]. Funktionen wie `generate`, `qa`, `translate` sind vorhanden und in Tests abgedeckt [7] [8].
- **Vector-Store** – als separater Service vorgesehen für dokumentenbasiertes Retrieval [6]. Eine einfache Implementierung existiert: Der neue Vektorservice speichert Texte in-memory mit HuggingFace-Embeddings und ermöglicht Similarity-Search [9] [10].
- **Worker-Services** – domänenspezifische Agenten-Dienste für **Dev**, **LOH** („Caregiver"), und **OpenHands**. Sie sind als eigene FastAPI-Services angelegt [11] [12] und verfügen über `/execute_task` Endpunkte [13]. Z.B. `worker_dev` generiert Python-Code-Snippets, `worker_loh` gibt kurze Pflege-Antworten, `worker_openhands` simuliert Docker-Operationen [14]. Diese Dienste antworten über den LLM-Gateway (d.h. rufen Modelle zur Ausführung auf) [15] [16].
- **CLI-Tool** – Es existiert ein installierbares CLI namens `agentnn`. In der aktuellen Version erlaubt es einfache Interaktionen: Fragen stellen (`ask`), Agenten/Sessions auflisten und Feedback geben [17] [18]. Dieses Kommandozeilentool greift über HTTP auf die API-Gateway/Dispatcher-Schnittstellen (`/chat`, `/agents` usw.) zu [19] [20]. Die CLI ist funktional für grundlegende Chats und wurde als Pip-Package bereitgestellt (siehe Installation im README) [21].

**Fazit:** Die oben genannten Kernmodule bilden ein lauffähiges Grundgerüst. Einfache End-to-End-Tests (z.B. ein Chat-Aufruf über CLI -> Gateway -> Dispatcher -> Worker) funktionieren bereits [22]. Diese Teile sind aktiv integriert und bilden die Basis für weitere Funktionen.

## Komponenten: Inkomplett, inkonsistent oder veraltet

Mehrere Bereiche des Repos sind derzeit unvollständig umgesetzt oder doppelt/inkonsistent vorhanden:

- **Task-Zuordnung & Routing:** Der Task-Dispatcher nutzt noch stark vereinfachte Logik. Aktuell wird z.B. nur der Task-Typ `"greeting"` explizit auf `worker_dev` gemappt [23]. Allgemeine Chat-Tasks (`"chat"`) fehlen in der Mapping-Tabelle, was darauf hindeutet, dass die dynamische Zuweisung unvollständig ist. Ein neu eingeführter *Routing-Agent* Service existiert zwar (mit statischen Regeln in `rules.yaml`), ist aber noch ein Platzhalter [24]. Die fortgeschrittene Routing-Logik (z.B. fähigkeitsbasiertes Matching) wurde begonnen, aber noch nicht produktiv eingebunden.

- **Meta-Learning und Agenten-Evolution:** Ältere Klassen wie `NNManager` und neuere wie `MetaLearner` koexistieren, was auf eine Übergangsphase hindeutet. Der `MetaLearner` (ein PyTorch-basiertes Modell zur Agentenauswahl) wurde zwar implementiert [25] [26], jedoch ist seine Integration noch offen. Im Codex-Tasks-Plan ist *"MetaLearner in NNManager integrieren"* ausdrücklich als Phase-2-Aufgabe aufgeführt [27]. Bislang verwendet der Supervisor/Dispatcher noch heuristische Verfahren (Embedding- und Regel-basiert im `NNManager`) [28] [29]. Ein automatisches Umlernen oder Agenten-Evolution (AutoTrainer, Feedback-Schleife) ist nicht sichtbar aktiv – entsprechende Platzhalter (z.B. `AdaptiveLearningManager`, `hybrid_matcher`) sind im Code vorhanden, werden aber scheinbar nicht vom aktuellen Laufzeitsystem genutzt.

- **Alte vs. neue Architektur (Monolith vs. Microservices):** Ein beträchtlicher Teil des Repos stammt aus der vormaligen Monolith-Architektur und überschneidet sich mit den neuen Services:

- Es existiert ein früher **Supervisor-Agent** (`agents/supervisor_agent.py`), der mit `AgentManager` und `NNManager` alle Aufgaben intern delegierte [30] [31]. Diese Logik wird nun durch den *Task-Dispatcher* Service abgelöst. Ähnlich wurden früher Agenten direkt im Prozess als Klassen instanziert (z.B. `CriticAgent`) [32], während jetzt *Worker-Services* über HTTP angesprochen werden. Die alten Manager-Klassen (`managers/agent_manager.py`, `managers/system_manager.py` etc.) erscheinen obsolet, da Registrierung und Sessions nun über Microservice laufen.
- Ein **monolithischer API-Server** (`api/server.py`) ist noch im Code, mit FastAPI-Routen unter `/api/v2` [33]. Gleichzeitig gibt es aber den neuen **API-Gateway** Service (`api_gateway/main.py`), der die eingehenden Anfragen an die Microservices weiterleitet [34] [35]. Dies ist redundant – langfristig sollte nur eine API-Schicht existieren. Die aktuelle Koexistenz führt zu Inkonsistenzen (z.B. erwartet die CLI einen `/agents`-Endpoint [36], der im API-Gateway aber in dieser Form evtl. fehlt oder anders realisiert ist).
- **Vector Store**: Hier zeigt sich ein Parallelismus alter/neuer Ansatz. Der ursprüngliche Code nutzte LangChain Chroma für den Vektorindex (`datastores/vector_store.py`) [37], während der neue Microservice eine eigene In-Memory-Lösung implementiert [38] [39]. Die Koexistenz deutet darauf hin, dass der Wechsel noch nicht konsolidiert ist.

- **Services-Verzeichnis:** Neben dem neuen `mcp/` Ordner für Microservices existiert `services/` mit älteren Service-Implementierungen (z.B. `services/critic_agent`, `services/coalition_manager`). Diese sind teils unvollständig und duplicativ. Beispielsweise gibt es dort Routen und Logik für einen CoalitionManager (Team-Bildung mehrerer Agenten) [40], was in der aktuellen Roadmap (MCP-Phasen) noch gar nicht priorisiert ist – vermutlich Überbleibsel eines früheren Konzepts. Solche Verzeichnisse sind nicht aktiv in den MCP integriert und wirken veraltet.

- **Frontend / Monitoring:** Ein React-Dashboard im Ordner `monitoring/` ist vorhanden (basiert auf TypeScript, vermutlich zur Systemüberwachung). Es handelt sich um ein separates UI, das Metriken via Prometheus/Grafana bereitstellt [41] [42] und z.B. eine Dokumenten-Viewer-Komponente enthält. Allerdings ist dieses Frontend derzeit isoliert und optisch/architektonisch nicht in das Hauptsystem eingebunden. Ein einheitliches **Web-Frontend für Nutzerinteraktion** (Chat-UI, Admin-Oberfläche) existiert offenbar noch nicht oder nur rudimentär. Der Monitoring-Bereich dürfte aktualisiert oder neu aufgebaut werden müssen, um mit der revidierten Architektur (API-Gateway, neue Services) konsistent zu funktionieren.

**Fazit:** Die Codebasis enthält noch erhebliche Reste der alten Monolith-Struktur und halbfertige neue Komponenten. Es besteht Inkonsistenz und Redundanz zwischen Alt und Neu (siehe doppelte CLI, zwei API-Ebenen, parallele VectorStore-Implementierungen). Diese müssen harmonisiert werden, bevor neue Features implementiert werden, da sonst Komplexität und Wartungsaufwand steigen.

## Redundanzen zwischen CLI, SDK, Services und Agenten

Es zeigen sich konkrete Dopplungen, die bereinigt werden sollten:

- **CLI vs. SDK-CLI:** Der Befehl `agentnn` ist offenbar zweimal implementiert. Zum einen als **einfache CLI** im Verzeichnis `cli/` (Click-basierend) mit Grundfunktionen [17], zum anderen als Teil der **SDK** in `sdk/cli/main.py` (Typer-basierend) mit erweiterten Unterkommandos [43] [44]. Die SDK-Variante unterstützt deutlich mehr (Agent-Verwaltung, Modelle, Governance-Funktionen etc.), während die einfache CLI nur Chat und Listing kann. Diese Duplizierung führt zu Verwirrung (zwei unterschiedliche CLIs namens `agent-nn`). Hier muss konsolidiert werden – bevorzugt zu *einer* CLI mit allen Funktionen.

- **Doppelte Service-Implementierungen:** Wie oben erwähnt, existieren Services sowohl unter `mcp/` als auch `services/`. Beispielsweise gibt es `services/vector_store` (mit eigenem FastAPI-App inkl. Auth/Metrics-Middleware) [45] und den neuen `mcp/vector_store` (simpler Service ohne Extras) [46]. Ähnliches gilt für LLM-Gateway (`services/llm_gateway` vs. `mcp/llm_gateway`). Diese Parallelstrukturen deuten darauf hin, dass die Migration noch nicht abgeschlossen ist. Die *services/*-Variante scheint älter und komplexer (mit Logging, Auth etc.), während die *mcp/*-Variante minimalistisch ist. Vermutlich sollen die neuen Services langfristig die alten ersetzen; bis dahin besteht jedoch Redundanz.

- **Agentenlogik doppelt (intern vs. Service):** Bestimmte Agenten sind sowohl als Python-Klasse implementiert als auch als externer Service. Z.B. **OpenHands**: Es gibt eine Klasse `DockerAgent` [47], die Docker-Container via OpenHands-API steuert, und gleichzeitig den Microservice `worker_openhands`, der aktuell aber nur eine LLM-basierte Dummy-Antwort gibt [16]. Ähnlich gibt es einen `CriticAgent` als Klasse [32] und einen (unfertigen) Critic-Service. Diese Doppelungen sollten aufgelöst werden, indem die Agenten-Fähigkeiten eindeutig einer Implementierungsform zugewiesen werden (bevorzugt Microservice mit klarer API). Andernfalls laufen parallele Varianten auseinander (z.B. ob eine Aufgabe nun intern im Prozess oder via REST erledigt wird).

- **API-Endpunkte doppelt:** Die Koordination erfolgt teils über die alte `api/server` (z.B. `/api/v2` Routen) und teils über `api_gateway` (z.B. `/chat`, `/llm/generate`). So könnten ähnliche Funktionen zwei verschiedene Wege haben. Die CLI nutzt z.B. `/chat` und `/sessions` über das Gateway [19] [48], während der alte Server umfangreiche Monitoring-Funktionen bot, die im Gateway fehlen. Hier müssen Endpunkte vereinheitlicht werden, damit es **eine** offizielle API gibt.

**Fazit:** Die parallelen Strukturen erhöhen die Komplexität. Ein zentrales Ziel der nächsten Entwicklungsphase muss das **Eliminieren von Redundanz** sein – d.h. alte Pfade abschalten oder migrieren und eine einzige konsistente Codebasis schaffen (eine CLI, eine API-Schicht, ein Satz Services, usw.).

## Integration des OpenHands-Agents

Der **OpenHands-Agent** ist im Repository präsent, aber noch nicht vollständig integriert:

- Es existiert ein **OpenHands API Server** (`openhands_api/server.py`), der tatsächlich Docker-Container steuern, Code ausführen und Compose-Stacks deployen kann [49] [50]. Dieser Server verlangt JWT-Authentifizierung und nutzt Redis als Queue – also eine eigenständige Komponente, die außerhalb der Kern-Agent-NN-Services läuft. Die Agent-NN-Codebasis enthält eine *Client-Integration*: Die Klasse `OpenHandsAgent` ruft diese OpenHands-API asynchron auf (z.B. `submit_code`, `build_image`, etc.) [51] [52].

- **Aktueller Nutzungsgrad:** Der Worker-Service `worker_openhands` nutzt *derzeit nicht* die obige Klasse, sondern generiert eine Platzhalter-Antwort via LLM [16]. Das heißt, in der Standardkonfiguration wird ein Docker-Task nur simuliert („Bestätige das Starten eines Docker-Containers…"). Die echte Ausführung über OpenHands (also tatsächliches Container-Management) ist noch nicht in den Request-Flow eingebunden. Möglicherweise war dies aus Sicherheitsgründen oder mangels Fertigstellung zunächst deaktiviert.

- **Konzeptuell vorgesehen:** Laut Dokumentation ist `worker_openhands` als spezieller Microservice geplant, der Container-Operationen übernimmt [14]. Die Integration mit docs.all-hands.dev (vermutlich die OpenHands-Doku) impliziert, dass Agent-NN Docker- und Code-Execution-Fähigkeiten über OpenHands erhalten soll. Die Basiskomponenten dafür sind da (API-Server, Agent-Klasse, Testcases [53] [54]), aber **die endgültige Einbindung fehlt**.

**Bewertung:** Der OpenHands-Agent ist **teilweise integriert** – die Codebasis kann prinzipiell OpenHands ansprechen, jedoch geschieht dies noch nicht standardmäßig in laufenden Tasks. Es gilt zu entscheiden, ob der OpenHands-Service künftig fest in die Architektur aufgenommen wird (etwa als eigenständiger Worker-Service, der real Container startet) oder ob diese Funktion optional/experimentell bleibt. Für eine produktive Konsolidierung müsste `worker_openhands` so erweitert werden, dass er tatsächliche Docker-Operationen über den OpenHands-API-Server ausführt (und nicht nur LLM-Text).

## Rolle des Codex-Agents (Autonomer Entwicklungsagent)

Der **Codex-Agent** bezeichnet hier den KI-gesteuerten Entwicklungsprozess, der in Agent-NN zum Einsatz kam. Er ist **nicht als Laufzeitmodul** im System enthalten, sondern fungierte als externer Helfer bei der Codeentwicklung. In `AGENTS.md` wird detailliert beschrieben, wie ein autonomer Agent in Phasen (Architekt, Planer, Entwickler, Tester, Doku-Agent) den Modernisierungsprozess begleitet [55] [56]. Diese Rollen und Richtlinien richten sich an einen KI-Codex (vermutlich GPT-4), der Code analysiert, Pläne schreibt und Änderungen vornimmt – also das, was tatsächlich im Projekt genutzt wurde, um die MCP-Architektur voranzutreiben.

Wichtig ist: Dieser Codex-Agent ist **kein Bestandteil des ausgelieferten Agent-NN-Systems**. Er läuft nicht als Service mit, sondern hat via `.codex.json` und Skripte (Codex CLI/Workflow) die Entwicklungsschritte orchestriert. Beispielsweise listet `.codex.json` konkrete Phasen und Tasks [57] [58], die der Codex-Agent abarbeiten sollte. Viele dieser automatisierten Aufgaben (Phasen 1–4) wurden umgesetzt, was man an entsprechenden Commits/Änderungen sieht (z.B. Einführung von ModelContext, Microservice-Stubs etc.). Der Codex-Agent konnte also **sinnvoll Codeänderungen generieren und versionieren** – er hat Tests geschrieben, Code formatiert und die Architektur restrukturiert, ohne selbst Teil des Repos zu sein. In den Commits finden sich Hinweise auf

automatische Änderungen gemäß Codex-Plan (z.B. aktivierte Feature-Flags in `.codex.json` für Phasen).

**Zusammengefasst:** Ja, der Codex-Agent hat bereits effektiv an Agent-NN gearbeitet (Refactoring, neue Module, Testgenerierung) und kann dies weiterhin tun. Aber er ist ausschließlich ein Entwicklungswerkzeug (AI Pair Programmer) außerhalb der produktiven Software. Für den weiteren Verlauf ist er relevant, um die unten definierten Schritte eventuell erneut KI-gestützt umzusetzen. Im Auslieferungszustand bleibt Agent-NN jedoch frei von eingebetteten „selbstprogrammierenden" Agenten – alle KI-Funktionalität zur Laufzeit bezieht sich nur auf die Fachdomänen (z.B. LLM für Gespräche, nicht zur Selbstmodifikation).

# Priorisierter Entwicklungsplan

**Zielvision:** Ein konsolidiertes Agent-NN System, das klar strukturierte Module hat und frei von Altlasten ist. Insbesondere **eine einzige CLI** und **eine einheitliche SDK**, ein aufgeräumtes **React-Frontend**, sowie eine eindeutige Aufteilung in Module (`core/`, `services/`, `datastores/`, `frontend/`, `api/`, `agents/`). Keine doppelten Codepfade mehr, alle Komponenten nahtlos integriert.

Um dieses Ziel zu erreichen, wird ein mehrphasiger Plan vorgeschlagen. Die Phasen orientieren sich an aufeinander aufbauenden Meilensteinen. **Kurzfristig** steht die Bereinigung und Grundkonsolidierung im Vordergrund, **mittelfristig** die Erweiterung von Features und **langfristig** die Qualitäts- und Produktivitätsmaßnahmen (Tests, CI, Dokumentation). Ein ungefährer Ablauf könnte so aussehen:

## Phase 1: Codebase-Konsolidierung & Architektur-Refaktor (0–4 Wochen) 📦

**Ziele:** Entfernung veralteter Komponenten, Zusammenführen redundanter Teile und Etablierung einer sauberen Projektstruktur. Diese Phase schafft die Basis, auf der neue Features zuverlässig aufsetzen können.

- **1.1 Verzeichnis- und Modulbereinigung:** Inventar aller bestehenden Ordner und Klassen erstellen und entscheiden, was entfällt oder migriert wird. Konkret:
- **Alte Monolith-Relikte entfernen oder archivieren:** Das Verzeichnis `api/` (monolith Server) wird stillgelegt. Ggf. relevante Endpunkte (Monitoring) in den neuen API-Gateway übernehmen, den Rest streichen. Ebenso werden `managers/`-Klassen wie `AgentManager`, `NNManager`, `SystemManager` in Rente geschickt – ihr Code wird nicht mehr vom Hauptfluss genutzt und sollte ausgelagert (z.B. nach `archive/`) oder gelöscht werden. Gleiches gilt für `services/agent_worker`, `services/coalition_manager` etc., sofern nicht durch MCP-Services genutzt. **Ergebnis:** Der laufende Code basiert einzig auf der Microservice-Architektur, ohne tote Altmodule im Hintergrund.
- **Klare Struktur einführen:** Richten Sie die Ordner an der Zielstruktur aus. Z.B. verschieben Sie den Inhalt von `mcp/` nach `services/` (so dass alle Microservice-Module unter `services/` liegen: `services/dispatcher`, `services/registry` etc.). Das erleichtert die Übersicht „alle Services hier". Das `monitoring/`-Frontend könnte nach `frontend/monitoring` oder direkt `frontend/` verschoben werden. Das `agents/`-Verzeichnis wird überprüft: dort sollten nur noch **Agent-Interfaces oder -Definitions** verbleiben, die auch in der neuen Arch benutzt werden (z.B. vielleicht Basisklassen für Worker-Agents). Alles andere (Demo-Agents, alte Worker-Klassen) wird entfernt.

- **Konfigurationsdateien anpassen:** Dateien wie `mcp_services.json` und `config/services.yaml` sind redundant; vereinheitlichen Sie auf *eine* Quellen-of-Truth. Z.B. könnte `config/services.yaml` zum Master werden, wo Ports/Namen aller Services definiert sind. Die Docker-Compose und evtl. Helm-Charts sollten daraus generiert/abgeglichen werden. Entfernen Sie verwirrende Duplikate (z.B. `.codex.json` Marker, falls diese nur für KI-Entwicklung gedacht waren). Falls `.codex.json` als Projekt-Plan erhalten bleiben soll, dokumentieren Sie ihren Zweck klar (nur Dev-Zwecke). **Ergebnis:** Ein schlankes Set an configs (llm_config.yaml, services.yaml, etc.) spiegelt die tatsächliche Struktur konsistent wider.

- **1.2 CLI und SDK zusammenführen:** Die zwei existierenden CLIs werden zu einer verschmolzen. Vorgehen:

- Entscheiden, welche Codebasis als Kern dient – vermutlich die umfangreichere `sdk/cli/main.py` (Typer) wegen der zusätzlichen Funktionen. In diese überführen Sie die noch sinnvollen Features der einfachen CLI (z.B. interaktiver Chat-Modus). Stellen Sie sicher, dass der Befehl `agentnn` (oder `agent-nn`) eindeutig auf diese eine Implementation zeigt.
- **SDK-Aufteilung prüfen:** Falls das SDK noch generierten Code aus OpenAPI enthält, konsolidieren Sie dies. Möglicherweise ist der `AgentClient` bereits eine Wrapper-Klasse für HTTP-Aufrufe an Gateway [59] – validieren Sie, dass alle wichtigen API-Routen darüber abgedeckt sind. Erweiteren Sie den SDK-Client bei Bedarf, um alle Kernfunktionen (Chat senden, Session verwalten, Model wechseln, etc.) abzudecken, sodass sowohl CLI als auch eventuelle externe Nutzer über das SDK alles steuern können.

- Entfernen Sie schließlich die alte CLI-Datei und verweisen Sie in Doku/README auf die neue vereinheitlichte CLI. **Erfolgskriterium:** `agentnn --help` zeigt nun *alle* Befehle in einem Tool, keine Dubletten. Tests für CLI-Befehle (so vorhanden) laufen grün.

- **1.3 API-Gateway finalisieren:** Machen Sie den **API-Gateway** zum einzigen Einstiegspunkt. Dafür:

- **Fehlende Endpunkte hinzufügen:** Übernehmen Sie nützliche Routen aus dem alten API-Server (z.B. System-Metriken oder Health-Checks aller Dienste) in den Gateway. Es sollte Endpunkte geben, um verfügbare Agents/Tools aufzulisten (`GET /agents` analog früherem CLI-Befehl) und evtl. zum Abruf aller Sessions (`GET /sessions`). Implementieren Sie diese im Gateway, indem der Gateway die entsprechenden Microservices befragt (Registry für Agents, Session-Manager für Sessions). Damit deckt das Gateway mindestens die Funktionalität ab, die zuvor der Monolith bot.
- **Authentifizierung & Rate-Limits vereinheitlichen:** Der Gateway hat bereits rudimentäre API-Key/JWT-Prüfung und Rate-Limiter [60] [61]. Stellen Sie sicher, dass *alle* Routen durch konsistente Auth-Mechanismen geschützt sind, und dass konfigurierbar ist, ob Auth aktiviert ist (z.B. für lokale Tests aus). Dokumentieren Sie dies einheitlich. Entfernen Sie Auth-Logik aus einzelnen Services, falls vorhanden, damit die Verantwortung im Gateway zentral liegt.
- Nach diesen Anpassungen alle Aufrufe der CLI/SDK auf Gateway-Basis testen. Danach kann der alte `api/server.py` endgültig deaktiviert werden.

- **Output:** Aktualisierte OpenAPI-Dokumentation für den Gateway (in `docs/api`), die die neuen vereinheitlichten Endpoints zeigt.

- **1.4 Einheitliche Logging/Metrik-Infrastruktur:** Während der Konsolidierung auch gleich die Logging- und Monitoring-Aspekte standardisieren. Viele Services initialisieren Logging, Prometheus und ähnliches unterschiedlich (teils gar nicht). Führen Sie in `core/`

`logging_utils.py` und `core/metrics_utils.py` gemeinsame Helfer ein, die in jedem Service genutzt werden. So bekommt jeder Service konsistente Request-Logs und Metriken (wie bereits ansatzweise implementiert, z.B. LoggingMiddleware im Gateway [62]). Diese Phase ist ein guter Zeitpunkt, um *Observability* als Querschnittsthema geradezuziehen (entspricht auch "Phase 1.6: MCP Observability" aus dem Plan [63]).

**Teilziele & Exit-Criteria Phase 1:** - Projektstruktur ist bereinigt: keine offensichtlichen Duplikat-Module mehr. Entwickler können sich an der neuen Ordnerstruktur (core, services, agents, datastores, frontend, sdk, etc.) orientieren ohne Altlast-Verwirrung. - Alle Tests laufen grün (angepasst an neue Struktur). Mindestens ein End-to-End Test (z.B. Chat über CLI) funktioniert über das neue Gateway/Services-Setup. - Dokumentation (README, Installationshinweise, CLI-Doku) ist auf den neuen Stand gebracht. - **Blocker:** Mögliche Risiken hier sind, dass man versehentlich funktionale Logik entfernt. Daher schrittweise vorgehen und nach jedem größeren Löschvorgang testen. Auch muss sichergestellt sein, dass die Teammitglieder wissen, welche Teile entfernt wurden, um nicht parallel dran zu entwickeln.

## Phase 2: Feature-Integration & Harmonisierung (5–8 Wochen)

**Ziele:** Nun, da die Basis konsolidiert ist, werden die **wichtigen fehlenden Funktionen** fertiggestellt und Inkonsistenzen in der Funktionalität ausgebügelt. Diese Phase macht das System „feature complete" entsprechend der ursprünglichen Zielsetzung (MCP Phasen 2–3). Schwerpunkt: intelligente Agenten-Routing, OpenHands-Einbindung, konsistentes SDK und Domain-Funktionen.

- **2.1 Dynamische Task-Zuweisung & Meta-Learning:** Implementieren Sie die intelligente Agenten-Auswahl, sodass der Dispatcher nicht mehr mit festen Maps arbeitet, sondern kontextabhängig entscheidet:
- **MetaLearner einbinden:** Nutzen Sie den vorhandenen `MetaLearner` (oder vereinfachte Variante) um aus Task-Beschreibungen einen geeigneten Worker abzuleiten. Dies kann schrittweise erfolgen: zunächst regelbasiert (Mapping von Task-Typen zu Agent anhand `routing_agent/rules.yaml` [24]), dann ggf. erweitert um ML. Als Übergang kann der *Routing-Agent Service* so erweitert werden, dass `POST /route` eine Ziel-Worker-URL zurückgibt aufgrund von Regeln *oder* einfachen heuristischen Scores. Der Dispatcher ruft diesen Routing-Service dann für unbekannte Task-Typen auf.
- **Agent Registry nutzen:** Stellen Sie sicher, dass der Agent-Registry-Service aktuelle Informationen aller aktiven Worker enthält (Name, URL, Fähigkeiten). Möglicherweise muss die Registry beim Start aller Worker aktualisiert werden (z.B. Worker melden sich beim Registry an). Damit kann der Dispatcher bei einer Anfrage dynamisch verfügbare Agents abfragen statt statische Liste.
- **MetaLearner-Training:** Beginnen Sie einen Feedback-Loop aufzubauen: Der Dispatcher übergibt Ergebnisdaten (Success, Dauer, Feedback) zurück an einen Lern-Component. In einfachster Form kann dies geloggt werden; fortgeschritten könnte `MetaLearner.update_metrics()` genutzt werden [64], um die Leistung der gewählten Agenten zu verfolgen. Das Ziel ist, über wiederholte Ausführungen die Agentenauswahl zu optimieren (Phase 2 des ursprünglichen Plans).

- **AutoTrainer:** Falls genügend Daten vorhanden, kann ein Hintergrundprozess (AutoTrainer) implementiert werden, der den MetaLearner periodisch nachtrainiert. Dieser Schritt ist optional in dieser Phase, aber einzuplanen.

- **2.2 Vollständige OpenHands-Integration:** Jetzt soll der OpenHands-Agent wirklich genutzt werden:

- **OpenHands Worker aktivieren:** Ändern Sie `worker_openhands.service.execute_task`, so dass er nicht mehr eine statische LLM-Antwort baut, sondern den OpenHandsAgent benutzt. Beispielsweise könnte `execute_task(task)` den Befehl via `OpenHandsAgent.submit_and_wait(task)` an den OpenHands-API-Server schicken und dessen Ergebnis (Erfolg/Misserfolg, ID, Logs) zurückliefern [51] [65]. Damit würde ein Task vom Typ „docker“ (oder wie auch immer bezeichnet) tatsächlich einen Container starten/stoppen etc.
- **Sicherheit prüfen:** Da hier real Code ausgeführt wird, bedenken Sie Sicherheitsaspekte. Nutzen Sie die im OpenHands-API-Server vorgesehene JWT-Auth – d.h. Worker_OpenHands muss ein gültiges Token besitzen (via ENV `GITHUB_TOKEN` oder ähnliches) [66]. Eventuell definieren Sie in der Agent-NN-Konfiguration, ob OpenHands-Features aktiv sind (Feature-Flag), falls man das System auch ohne Docker-Anbindung betreiben will.
- **Testfälle erweitern:** Schreiben Sie Integrationstests, die einen Dummy-OpenHands-API-Server simulieren, um die End-to-End-Funktion zu prüfen (z.B. via aiohttp TestClient). So stellen Sie sicher, dass Worker_OpenHands korrekt mit dem Server redet und Fehlerfälle handhabt (Timeouts, Exceptions).

- **Dokumentation:** Ergänzen Sie die Doku (README oder separate MD), wie der OpenHands-Service zu starten/konfigurieren ist (z.B. `OPENHANDS_API_URL`, benötigte Docker/Redis Versionen). Klären Sie auch, welche Use-Cases damit abgedeckt werden (z.B. Code-Execution-Aufgaben).

- **2.3 Domain-spezifische Worker ausbauen:** Neben OpenHands sollten auch die anderen Worker intelligenter gestaltet werden:

- **Dev-Agent (worker_dev):** Momentan generiert dieser einfach Python-Code via LLM. Prüfen Sie, ob er erweiterte Fähigkeiten braucht – etwa das Kompilieren/Ausführen des generierten Codes oder Validierung. Ggf. könnte worker_dev z.B. einen kleinen Sandbox-Executor nutzen, um das Ergebnis des Codes zurückzugeben (derzeit scheint er nur den generierten Text zurückzugeben). Wenn das zu aufwändig ist, zumindest den Prompt verbessern (der LLM prompt kann Tests enthalten etc.).
- **LOH-Agent (worker_loh):** Dieser soll “kurze caregiving answers” geben [14]. Verifizieren Sie, dass `worker_loh.service` tatsächlich den LLM mit einem geeigneten Prompt ansteuert (vmtl. ähnlich wie OpenHands gerade implementiert ist). Gegebenenfalls Fein-Tuning: z.B. ein definierter System-Prompt für empathische Antworten hinterlegen. Das Ziel ist, dass jeder Worker-Service einen klar umrissenen Anwendungsfall vollständig erfüllt. Falls weitere Domain-Agents geplant sind (Dev, OpenHands, LOH sind 3 – vielleicht kommen noch FinanceAgent etc.), bereiten Sie die Infrastruktur dafür vor.

- **Einheitliche Response-Struktur:** Sorgen Sie dafür, dass alle Worker ihre Ergebnisse einheitlich zurückmelden. Z.B. ein JSON mit `result`, optional `confidence` oder Fehler. In Phase 1 wurde bereits begonnen, die Dispatcher-Antwort zu homogenisieren (`worker`, `response`, `duration`, `confidence`) [67]. Führen Sie das konsequent fort, sodass das Frontend/CLI nicht für jeden Agententyp eigene Output-Parsing benötigt.

- **2.4 SDK/CLI Feature-Completion:** Nachdem Backend-Funktionen erweitert wurden, passen Sie SDK und CLI an, um diese auszunutzen:

- **CLI-Befehle erweitern:** Ergänzen Sie z.B. einen Befehl `agentnn agent list` (der `/agents` aufruft, Liste aller registrierten Agenten) und `agentnn session list` (für Sessions) falls noch nicht vorhanden oder in Phase 1 erledigt. Stellen Sie sicher, dass `agentnn ask` nun auch verschiedene Agents ansprechen kann, nicht nur “dev” – möglicherweise erlauben Sie `--agent`

Switch für unterschiedliche Domain-Agents. So können Nutzer gezielt an z.B. LOH-Agent Fragen stellen.

- **Model/Provider Management:** Falls Phase 3 des MCP (SDK & Provider-System) noch nicht umgesetzt: Implementieren Sie ein System, um verschiedene LLM-Provider zu nutzen. Das scheint schon angelegt (OpenAI, Anthropic, lokale Models in `llm_models/`). Hier sollten CLI/SDK-Befehle wie `agentnn model switch` funktionieren [68]. Verifizieren Sie den Status dieser Commands (Doku existiert bereits [69]). Implementieren Sie im SDK `ModelManager` das Laden verschiedener Modelle (OpenAI via API Key, Local via Huggingface etc.), basierend auf Konfiguration (`llm_config.yaml`). Ziel: Der Nutzer kann per CLI zwischen GPT-4, Llama-2 etc. wechseln, und LLM-Gateway nutzt entsprechend den richtigen Backend.

- **Skill-System und Tools:** Im Code gibt es Konzepte wie `core.skills` und `core.agent_profile` (Profile mit Traits, Skills). Diese könnten relevant werden, falls Agenten eigenständig Fähigkeiten laden können. Entscheiden Sie, ob dieses Feature in MVP benötigt wird. Wenn ja, harmonisieren Sie es mit dem Worker-Service-Ansatz (evtl. können Worker beim Start ihre Skills dem Registry melden). Wenn nicht, dokumentieren Sie es als zukünftige Erweiterung und belassen es passiv.

- **2.5 Frontend-Überarbeitung (Teil 1 – Grundfunktionen):** Starten Sie die Arbeiten am React-Frontend:

- **Monitoring-Dashboard aktualisieren:** Passen Sie das vorhandene Monitoring-UI an die neue Backend-API an. Das Dashboard sollte mindestens den Status der Microservices anzeigen (Health-Checks) und wichtige Metriken (Anzahl Sessions, letzte Tasks, Performance) visualisieren. Nutzen Sie hierzu die Prometheus-Metriken, die Phase 1 eingeführt hat, und bauen Sie ggf. einfache Übersichts-Endpoints (z.B. Gateway route `/metrics/summary`).
- **Einfache Chat-Oberfläche integrieren:** Falls noch nicht vorhanden, implementieren Sie im Frontend ein einfaches Chat-Interface, das über den API-Gateway mit dem Dispatcher kommuniziert. Damit kann man direkt über den Browser mit Agenten interagieren (ähnlich der CLI-Funktion). Dies erhöht die Zugänglichkeit enorm. Fokus zunächst auf Funktionalität (eine Textarea, Senden-Button, Ausgabe des Agent-Antworttextes). Die Session-ID kann im Hintergrund vom Gateway generiert werden und in der UI gehalten werden, um Konversationen fortzusetzen.
- **Feedback und Verlauf:** Bauen Sie eine Ansicht, die vergangene Chats/Sessions anzeigt, und ermöglichen Sie pro Nachricht Feedback (good/bad) wie per CLI möglich [70]. Dieses Feedback fließt bereits in Session-Manager (siehe `/session/{id}/feedback` im Gateway) [71] [72] – das Frontend sollte es auch anbieten, damit Endnutzer bewerten können.
- Diese Frontend-Aufgaben können parallel von UI-Entwicklern durchgeführt werden, sollten aber mit der Backend-API-Entwicklung synchronisiert werden.

**Exit-Kriterien Phase 2:** Das System bietet nun alle Kernfunktionen an: intelligente Aufgabenverteilung, echte Docker/Code-Execution über OpenHands, Multi-LLM-Support und ein rudimentäres Web-UI. Die CLI/SDK sind vollständig und konsistent mit der Backend-Logik. Interne Inkompatibilitäten aus Phase 1 (z.B. fehlende Endpunkte) sind behoben. Mindestens ein Agent (z.B. Dev-Agent) nutzt tatsächlich das neue Lernmodell oder Routing-Mechanismus erfolgreich (per definiertem Erfolgskriterium aus `.codex.json` Phase 2) [73].

Mögliche **Blocker/Herausforderungen:** Feinabstimmung des MetaLearners benötigt genügend Feedback-Daten – evtl. ist kurzfristig ein einfacherer Schwellwert-Ansatz pragmatischer. Die OpenHands-Integration könnte durch Umgebungsprobleme (Docker erforderlich, Sicherheitsbedenken) erschwert werden; hierfür evtl. Sandbox-Umgebung einplanen. Außerdem muss man darauf achten,

dass parallel viele Änderungen in verschiedenen Teilen gemacht werden – sauberes Versionieren (Branches pro Feature) und Code-Review sind wichtig, um nichts zu brechen.

## Phase 3: Konsolidiertes Frontend & User Experience (9–12 Wochen) 💻

**Ziele:** In dieser Phase wird das **Frontend professionalisiert** und die Module weiter konsolidiert, um ein benutzerfreundliches, einheitliches Produkt zu formen. Zudem werden letzte veraltete Verzeichnisse entfernt und alles auf Release-Qualität gebracht.

- **3.1 Unified React Frontend ("All-in-One" UI):** Aufbauend auf Phase 2 erweitern und optimieren Sie die Web-Oberfläche:
- **Modulares Frontend:** Strukturell sollte das Frontend in Sektionen gegliedert werden: *Chat/Interaktionsbereich*, *Monitoring-Dashboard*, *Konfiguration*. Nutzen Sie React-Router oder ein Tabs/Menu-System, damit Nutzer zwischen **Chat UI** und **Admin UI** wechseln können. So haben Endanwender eine Chat-Maske, während Entwickler/Operatoren Einblick in Metriken und Logs erhalten.
- **Design & Usability:** Arbeiten Sie am Look-and-Feel. Vielleicht das Corporate Design von All-Hands/OpenHands übernehmen, um Konsistenz zu haben. Wichtig: responsives Design, Eingabeflüsse (z.B. beim Stellen einer Aufgabe Auswahl, welcher Agent/Skill genutzt wird).
- **Erweiterte Funktionen:** Integrieren Sie fortgeschrittene Features in die UI, z.B.:
    - Management-Ansichten für Agents (Liste registrierter Worker mit Status „online/offline", Möglichkeit neue Worker per API hinzuzufügen oder bestehende neu zu starten).
    - Anzeige des VectorStore-Inhalts: ggf. eine einfache Liste von Dokumenten, die im Wissensspeicher liegen, und eine Upload-Funktion um neue Dokumente hinzuzufügen (dies könnte über einen neuen Endpoint `/vector_store/document` realisiert werden, und per Frontend-Form).
    - Modellverwaltung: UI-Controls um das aktive LLM-Modell zu wechseln oder API-Keys einzugeben (diese könnten in `llm_config.yaml` gespeichert werden – das Frontend könnte dafür einen Safe-Upload vorsehen).

- **Testing:** Schreiben Sie End-to-End-Tests für die UI (z.B. mit Selenium oder Puppeteer) insbesondere für kritische Pfade: Chatten mit Agent, Feedback geben, Metriken live updaten. Diese Tests stellen sicher, dass das Frontend wirklich mit dem Backend harmoniert.

- **3.2 Backend Finetuning & Cleanup:** Parallel zum UI-Fokus werden letzte Backend-Baustellen abgeschlossen:

- **Bereinigung überflüssiger Verzeichnisse:** Nach Phase 2 sollten die meisten Altlasten weg sein. Nun endgültig entfernen: jegliche *archive/* Ordner, nicht mehr gebrauchte Demos (z.B. `archive/sample_agent` falls vorhanden), alte Doku-Dateien mit überholtem Inhalt. Was nicht gelöscht wird, aber nicht im Kernsystem ist (z.B. experimentelle `agents/nn_worker_agent.py`), klar als *deprecated* markieren. Das Repo soll einen eindeutigen Satz aktiver Module präsentieren.
- **Konfigurationsabgleich:** Überprüfen Sie die **Trainingspfade und gespeicherten Daten**. Z.B.: Wo werden Modelldateien abgelegt ( `models/agent_nn/*.pt` von AgentNN) [74] [75] ? Wo liegen Agentenprofile (PROFILE_DIR)? Stimmen diese Pfade mit dem neuen Layout überein (ggf. alles unter einem zentralen `data/` Verzeichnis sammeln). Definieren Sie in einer zentralen Config (etwa `agentnn.yaml`) alle relevanten Pfade und laden Sie diese in den Modulen aus der

Config statt hartcodiert. Das verhindert, dass Trainingsdaten verloren gehen oder an unbekannten Orten liegen.
- **Performance & Scalability:** Führen Sie Lasttests auf den Microservices durch. Optimieren Sie falls nötig: z.B. Uvicorn-Worker-Anzahl erhöhen, Timeout-Werte justieren (aktuell oftmals 10s fest kodiert [76] [77]). Implementieren Sie ggf. Retries für kritische externe Aufrufe (LLM-API, OpenHands-API) im SDK, um Robustheit zu steigern.
- **Security Review:** Phase 3 ist ein guter Zeitpunkt für einen kurzen Security-Audit. Überprüfen Sie, ob keine sensiblen Infos im Repo sind (API-Keys in config? – sollten via ENV Variablen kommen). Die Privacy-Filter (`core/privacy_filter.py`) und Governance/Contract-Mechanismen sollten auf Konsistenz geprüft werden – falls diese Features (AccessControl, Verträge) noch nicht aktiv genutzt werden, entscheiden ob MVP-relevant oder in späteres Release verschoben.

**Exit-Kriterien Phase 3:** Die Anwendung präsentiert sich nun als **gerundetes Produkt**: Benutzer können über ein Web-Interface und CLI das System nutzen, alle geplanten Agenten-Fähigkeiten sind vorhanden. Die Verzeichnisstruktur ist final konsolidiert, jede Komponente hat ihren Platz (core, services, frontend, etc.), nichts "schwebt" veraltet herum. Die *Zielstruktur* mit eindeutigen Modulkategorien ist erreicht. Interne Konsistenz: Konfigurationen greifen, Pfade stimmen, Logging und Monitoring laufen übergreifend.

## Phase 4: Testabdeckung, CI/CD und Dokumentation (ab Woche 13)

**Ziele:** In der Schlussphase geht es darum, die Qualität zu sichern und das Projekt auf einen wartbaren Stand zu heben. Dazu gehören umfassende **automatisierte Tests**, ein sauberer **CI/CD-Workflow** und vollständige **Dokumentation** für Nutzer und Entwickler. Diese Phase mündet in einen Release-Candidate der neuen Version.

- **4.1 Testsuite erweitern:** Schreiben Sie fehlende Tests, bis kritische Komponenten >90% Coverage haben [78]. Schwerpunkte:
- **Unit-Tests für jeden Service:** z.B. Dispatcher-Logik (Routing mit verschiedenen Inputs), Session-Manager (Speichern/Abrufen von Verlauf), LLM-Gateway (Mock LLMs und prüfen, dass `generate`, `qa` korrekt funktionieren inklusive Fehlerpfade) [79] [80].
- **Integrationstests End-to-End:** Simulieren Sie einen vollen Request via API-Gateway bis zum Worker und zurück. Nutzen Sie hierfür am besten die FastAPI TestClient in Python für jeden Microservice, oder Docker-Compose in CI, um tatsächlich Container hochzufahren. Ein Test könnte sein: Starte alle Services (z.B. per `scripts/start_mcp.sh`), sende einen Chat-Request, warte auf Antwort, prüfe Format und Inhalt. Ebenso Tests für OpenHands-Flow (ggf. mit stubbed OpenHands API).
- **Negative Tests & Randfälle:** Wie verhält sich das System bei ungültigen Eingaben? (z.B. sehr lange Prompts, unbekannter Agenten-Name, fehlendem Token). Schreiben Sie Tests, die sicherstellen, dass Fehler ordentlich zurückgemeldet werden (HTTP 400/500 mit sinnvollen Messages, nicht z.B. unhandled exceptions).

- **Performance-Tests (optional):** Wenn möglich, integrieren Sie einen Test, der eine Reihe paralleler Anfragen schickt, um sicherzustellen, dass die Rate-Limits greifen und das System stabil bleibt unter Last.

- **4.2 CI/CD Pipeline einrichten:** Konfigurieren Sie GitHub Actions oder eine ähnliche CI:

- **Linting & Static Analysis:** Führen Sie Format-Prüfer (Black, isort) und Linter (ruff/Flake8) in der Pipeline aus, so dass jeder PR automatisch auf Stil-Konformität geprüft wird [81]. Fehler in Typisierungen (mypy) oder Sicherheitslinter (Bandit) können optional hinzu.
- **Testausführung:** Die gesamte Pytest-Suite sollte in der CI laufen (inklusive ggf. Spin-up von Services via Docker). Richten Sie auch Coverage-Berichte ein. Ein Kriterium kann sein, dass ein PR nur gemerged werden darf, wenn z.B. Coverage > 80% und kein Test fail.
- **Build & Deployment:** Erstellen Sie in CI auch automatisch Docker-Images für die Services. Ein Stage sollte `docker-compose build` ausführen und im Erfolgsfall Images mit Tags (z.B. `:latest` oder Git-Hash) ins Container-Registry pushen [82]. Ebenso den Python-Package-Build für das SDK (über `setup.py`/Poetry). So ist sichergestellt, dass jederzeit ein Deployment der neuesten Version möglich ist.

- **Continuous Deployment (falls gewünscht):** Denkbar ist, einen Workflow aufzusetzen, der bei Push auf `main` automatisch einen dev-Server aktualisiert (z.B. via SSH Docker pull, oder Kubernetes Rolling Update, falls Infrastruktur vorhanden). Dies ist optional, aber einen "Release"-Workflow (mit Versions-Bump, Tagging, Changelog generieren) sollte man vorsehen.

- **4.3 Dokumentationsabschluss:** Überarbeiten Sie alle Dokumente, sodass sie den neuen Stand reflektieren:

- **README:** Sollte Installationsanleitung (für Docker- und dev-Setup) enthalten, kurze Komponenten-Übersicht (Mermaid-Diagramm updaten) [83], sowie Beispiel, wie man das System nutzt (CLI und Web).
- **User Guide:** In `docs/` einen ausführlichen Benutzerleitfaden, der auf Deutsch die Nutzung aller Kernfeatures erklärt (basierend auf der bisherigen CLI-Doku [84] [68] etc.). Auch eine Sektion "Troubleshooting" (Häufige Probleme) ergänzen, damit Nutzer sich selbst helfen können bei Setup-Problemen.
- **Developer Docs:** Halten Sie fest, wie die Module strukturiert sind, eventuell mit Diagrammen für die neue Architektur (ähnlich `overview_mcp.md` aber aktualisiert). Beschreiben Sie auch die *Agent Lifecycle* – z.B. wie ein neuer Worker-Service hinzugefügt werden kann, falls jemand das System erweitern will.
- **Inline-Dokumentation:** Gehen Sie den Code durch und ergänzen fehlende Docstrings, insbesondere für öffentlich nutzbare SDK-Funktionen und core-Klassen. Einheitliche Terminologie ist wichtig (z.B. durchgehend "Task-Dispatcher" statt mal "Supervisor" verwenden) [85].

- **Überprüfung:** Zum Abschluss einen "frischen" Durchlauf simulieren: neues Repository klonen, Doku Schritt für Schritt befolgen – klappt alles? Diese Prüfung stellt sicher, dass keine Diskrepanzen zwischen Anleitung und Realität bestehen [86].

- **4.4 Vorbereitung Release:** Fassen Sie zum Abschluss alle Änderungen in einem Changelog zusammen (siehe RELEASE.md Vorlage) und heben Sie wichtige Neuerungen hervor [87]. Erhöhen Sie die Versionsnummer entsprechend (v1.0 finale oder v2.0, je nachdem). Stellen Sie sicher, dass die Container-Images und das SDK-Package versioniert und veröffentlicht werden (z.B. Docker Hub, PyPI, falls vorgesehen).

**Exit-Kriterien Phase 4:** Die Testabdeckung erreicht ~90% in kritischen Bereichen [88], der CI-Badge ist grün für Lint/Tests. Die Dokumentation ist vollständig und auf dem aktuellen Stand – keine "TODO" Platzhalter mehr, alle öffentlichen Seiten sind lesbar und hilfreich [85]. Ein neuer Nutzer kann das System laut Anleitung installieren, starten und verwenden, ohne Insider-Wissen. Damit ist Agent-NN in einer **produktionsreifen Architektur** fertiggestellt [89].

**Blocker & Risiken:** Hauptgefahr in dieser Phase ist Zeitdruck und “Test-Fatiguen” – das Testschreiben darf nicht vernachlässigt werden. Ebenso muss man achtgeben, dass Lasttests mit echten LLM-APIs Kosten verursachen (API-Keys, Rate Limits) – hier simulieren oder stubben, um kein Budget zu sprengen. Dokumentation sollte idealerweise von jemandem gegen-gelesen werden, der nicht im täglichen Projekt war, um Verständlichkeitslücken aufzudecken.

---

**Abschließende Empfehlung:** Durch diese vier Phasen wird aus dem derzeit noch fragmentierten Agent-NN-Projekt ein schlankes, einheitliches System. Die Priorität liegt klar auf **Konsolidierung vor Erweiterung** – erst wenn alles an seinem Platz ist (Phase 1), sollten neue Features richtig aktiviert werden (Phase 2). Dies verhindert technische Schulden. Der vorgeschlagene Zeitrahmen (~3 Monate) kann je nach Teamgröße variieren, aber die Reihenfolge ist entscheidend. Mit Abschluss dieses Plans hätte Agent-NN ein robustes Fundament und könnte in Zukunft leichter um weitere Agenten oder Fähigkeiten (z.B. Phase 4+ Ideen: agent networks, Reputation-System) erweitert werden [90], ohne am Kern rütteln zu müssen. Viel Erfolg bei der Umsetzung!

---


[1] [6] [21] [22] [83] README.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/README.md

[2] [3] [23] [67] [76] service.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/mcp/task_dispatcher/service.py

[4] [5] [11] [12] docker-compose.yml
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/mcp/docker-compose.yml

[7] service.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/mcp/llm_gateway/service.py

[8] [79] [80] test_llm_gateway_service.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/tests/test_llm_gateway_service.py

[9] [10] [38] [39] [46] service.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/mcp/vector_store/service.py

[13] api.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/mcp/worker_dev/api.py

[14] worker_services.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/docs/api/worker_services.md

[15] [16] [77] service.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/mcp/worker_openhands/service.py

[17] [18] [19] [20] [36] [48] [70] agent_nn_cli.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/cli/agent_nn_cli.py

24 service.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/mcp/routing_agent/service.py

25 26 64 meta_learner.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/managers/meta_learner.py

27 codex.tasks.json
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/codex.tasks.json

28 29 nn_manager.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/managers/nn_manager.py

30 31 supervisor_agent.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/agents/supervisor_agent.py

32 critic_agent.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/services/agent_worker/demo_agents/critic_agent.py

33 server.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/api/server.py

34 35 41 42 60 61 62 71 72 main.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/api_gateway/main.py

37 vector_store.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/datastores/vector_store.py

40 routes.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/services/coalition_manager/routes.py

43 44 59 main.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/sdk/cli/main.py

45 main.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/services/vector_store/main.py

47 52 docker_agent.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/agents/openhands/docker_agent.py

49 50 server.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/openhands_api/server.py

51 65 66 base_openhands_agent.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/agents/openhands/base_openhands_agent.py

53 54 test_openhands_agents.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/tests/test_openhands_agents.py

55 56 78 85 86 88 AGENTS.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/AGENTS.md

57 58 63 73 81 89 90 .codex.json
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/.codex.json

68 69 84 CLI-Dokumentation.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/docs/cli/CLI-Dokumentation.md

74 75 nn_worker_agent.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/agents/nn_worker_agent.py

82 87 RELEASE.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/RELEASE.md