ChatGPT

Modernisierung von Agent-NN zur Modular-Control-Plane-Architektur

Aktuelle Architektur von Agent-NN (Ist-Zustand)

Agent-NN ist derzeit als monolithisches Multi-Agenten-System organisiert. Die Hauptkomponenten und ihre Rollen sind:

- **ChatbotAgent** – Frontend-Agent für Benutzerinteraktion (z.B. Chat-Oberfläche). Er nimmt Nutzereingaben entgegen und leitet Anfragen an den Supervisor weiter [1].
- **SupervisorAgent** – Zentraler Orchestrator, der eingehende Aufgaben analysiert und an passende Worker-Agents delegiert [2] [3]. Er wählt mithilfe des **NNManager** den besten Agenten für eine Aufgabe aus (basierend auf Embeddings/ML-Modell) und überwacht die Ausführung.
- **WorkerAgents** – Spezialisierte Agenten für verschiedene Domänen (z.B. Dev-Agenten für Softwareentwicklung, OpenHands-Agenten für DevOps/Docker-Aufgaben, geplante LOH-Agenten etc.). Jeder WorkerAgent besitzt domänenspezifisches Wissen und Fähigkeiten sowie eine eigene Wissensbasis [4]. Sie führen die vom Supervisor zugewiesenen Aufgaben aus.
- **AgentManager & NNManager** – Manager-Komponenten zur Verwaltung der Agenten. Der **AgentManager** verwaltet den Lebenszyklus der WorkerAgents und hält eine Registry verfügbarer Agenten bereit [5]. Der **NNManager** nutzt neuronale Netze/Embeddings, um basierend auf Aufgabenbeschreibung und Kontext den geeignetsten Agenten vorherzusagen [2] [6]. (In der aktuellen Implementierung überschneiden sich AgentManager und NNManager teilweise in ihrer Zuständigkeit für die Agentenauswahl.)
- **AgentCommunicationHub** – Ein zentrales Nachrichtensystem für die interne Kommunikation zwischen Agenten [2]. Es ermöglicht asynchronen Nachrichtenaustausch, Broadcasts und Wissensaustausch zwischen Agenten [7]. Derzeit ist dies als internes Hub-Modul realisiert.
- **Wissensverwaltung** – *DomainKnowledgeManager*, *VectorStore* und *WorkerAgentDB* verwalten das Wissen der Agenten [8]. Dokumente und deren Vektorembeddings werden in der VectorStore/WorkerAgentDB gespeichert, sodass Agenten semantisch nach relevanten Informationen suchen können [9]. Der DomainKnowledgeManager ermöglicht domänenübergreifende Abfragen und Retrieval-Augmented Generation (RAG).
- **LLM-Integration** – In `llm_models/` sind Schnittstellen zu verschiedenen LLM-Backends definiert (OpenAI, Azure, lokale Modelle via LM Studio usw.) [10]. Ein *LLMBackendManager* abstrahiert unterschiedliche Anbieter [11]. Aktuell werden LLMs synchron aus den Agenten heraus genutzt. Es existieren **SpecializedLLM**-Klassen für domänenspezifische Prompts [12]. Einige Agenten (z.B. *AgenticWorker*) integrieren bereits LangChain-Komponenten, um komplexere Werkzeugaufrufe und Ketten von LLM-Aktionen umzusetzen [13].
- **CLI und API** – Das Projekt bietet eine Kommandozeilenanwendung (im Verzeichnis `cli/`) für lokale Interaktion [14] sowie erste FastAPI-basierte Endpunkte (`api/` und `openhands_api/`) für externe Anfragen [15]. Diese Interfaces rufen intern die oben genannten Klassen auf.

Schwachpunkte der aktuellen Architektur: Alle Komponenten laufen im selben Prozess/Runtime. Es gibt derzeit keine **MCP (Modular Control Plane)**-Trennung – d.h. keine eigenständigen Services, die bestimmte Aufgaben isoliert übernehmen. Das System ist eng gekoppelt: Der Supervisor ruft direkt Methoden der Agenten auf, und die Kommunikation erfolgt in-process über den CommunicationHub.

Skalierung und Ausfallsicherheit sind dadurch begrenzt. Zwar ist die Architektur modular aufgebaut (viele Module/Klassen) [16], aber nicht in separate Dienste aufgeteilt. Außerdem fehlen noch einige Komponenten (LOH-Agent, Setup-Agent etc.) und umfassende Tests [17] [18]. Diese monolithische Struktur soll nun in eine verteilte MCP-Architektur überführt werden.

Ableitung modularer MCP-Services aus der bestehenden Architektur

Aus der Ist-Architektur lassen sich klare Verantwortlichkeiten herauslösen, die als unabhängige **MCP-Module** (Microservices) neu umgesetzt werden können. Folgende Komponenten bieten sich für eine Entkopplung an:

- **Task-Dispatcher (Orchestrator)** – entspricht dem heutigen SupervisorAgent. Dieser Service nimmt Nutzeranfragen entgegen, entscheidet mittels KI/Regeln welcher Worker-Agent zuständig ist, und delegiert die Aufgabe an den entsprechenden Worker-Service. Durch Auslagerung in einen eigenen Dienst kann die Orchestrierungslogik zentral gebündelt und unabhängig skaliert werden.
- **Agent Registry Service** – übernimmt die Funktionen des AgentManager (Agentenverwaltung). Dieser Dienst hält Informationen über alle verfügbaren Agenten vor (Registrierung der Agent-Services, ihre Fähigkeiten, Zuständigkeiten und Endpunkte). Er ermöglicht dem Orchestrator, dynamisch verfügbare Worker zu finden. Die Registry kann auch Leistungsmetriken oder Health-Status der Agenten tracken.
- **Session Manager** – verwaltet Sitzungen und Kontext für laufende Interaktionen. In der aktuellen Architektur wird Kontext ggf. im ChatbotAgent oder in globalen Variablen gehalten; künftig übernimmt ein Session-Service dies zentral. Er speichert Gesprächsverläufe, Zwischenresultate oder Benutzerzustände (z.B. mit schnellem Zugriff via Redis). So bleibt Kontext erhalten, selbst wenn verschiedene Services beteiligt sind (Stateful Sessions [19] [20]).
- **Knowledge Vector Store Service** – kapselt die semantische Wissenssuche. Anstatt dass jeder Agent direkt auf `VectorStore`/Datenbank zugreift, bietet ein separater Dienst eine API für Einfügen von Dokumenten und Ähnlichkeitssuchen. Dieser Dienst kann intern eine Vektordatenbank (z.B. FAISS, Pinecone-Client oder Redis-Vector-Store) nutzen und ermöglicht Retrieval-Augmented Generation für alle Agenten. Die Auslagerung erlaubt unabhängige Skalierung des speicher- und rechenintensiven Vektor-Suchsystems und einfacheren Austausch der Technologie.
- **LLM Gateway Service** – zentralisiert die Anbindung an LLM-Backends. Dieser Dienst stellt ein einheitliches Interface bereit, um Prompt-Anfragen an OpenAI, lokale Modelle oder andere LLMs zu schicken [21]. Er kann LangChain nutzen, um komplexe Abläufe zu koordinieren – z.B. erst Wissensabruf vom Vector-Store und dann einen LLM-Aufruf (eine RAG-Kette). Durch einen LLM-Service können Caching, Rate-Limits und Fehlermanagement für Modellaufrufe an einer Stelle behandelt werden. Außerdem ermöglicht dies, die LLM-Last auf dedizierte Maschinen/GPU-Server auszulagern.
- **Worker Agent Services** – die einzelnen spezialisierten Agenten werden zu eigenständigen Microservices. Statt als Klassen im selben Prozess zu laufen, wird jeder Agent (oder jede Agenten-Kategorie) in einem separaten Dienst gekapselt. Z.B. ein **Dev-Agent-Service** für Softwareentwicklung, ein **OpenHands-Service** für Docker/Umgebungsaufgaben, ggf. ein **LOH-Agent-Service** etc. Jeder dieser Services hat eine klare API (etwa einen Endpoint wie `/execute_task`), nimmt vom Orchestrator delegierte Aufträge an und führt sie autonom aus. Sie beinhalten die jeweilige Geschäftslogik und können intern natürlich weiterhin auf gemeinsame Bibliotheken (z.B. LangChain, Tools) zugreifen – jedoch isoliert vom Rest. Diese Aufteilung fördert **Separation of Concerns**: Änderungen an einem Agenten betreffen nur

dessen Microservice. Zudem kann man kritischere Services getrennt absichern (z.B. den OpenHands-Service in einer Sandbox laufen lassen, da er potenziell systemnahe Kommandos ausführt).

- **Inter-Service Communication** – Anstelle des bisherigen in-process AgentCommunicationHub wird eine infrastrukturelle Nachrichtenvermittlung eingesetzt. Hier bieten sich **Message Queues** oder Publish/Subscribe-Systeme an (z.B. Redis Pub/Sub, RabbitMQ oder Apache Kafka) für die asynchrone Kommunikation zwischen den Diensten. Der Task-Dispatcher kann Aufgaben auf eine Queue legen, von der der zuständige Worker-Service sie abholt (Push-Prinzip), oder gezielt an einen Worker-Service senden (RPC-Prinzip). Ein Message-Broker ermöglicht entkoppelte, fehlertolerante Abläufe und bessere Lastverteilung (Stichwort **Asynchrone Verarbeitung** für parallelen Betrieb [22]). Falls geringere Latenz wichtig ist, kann ergänzend gRPC für synchrone RPC-Aufrufe zwischen bestimmten Services genutzt werden – z.B. der Orchestrator ruft den LLM-Service via gRPC auf und bekommt direkt die Antwort. In jedem Fall ersetzt eine solche Messaging-Schicht den monolithischen CommunicationHub durch eine robuste, verteilte Kommunikationsstruktur.

Zusätzlich zu diesen Hauptkomponenten können weitere Module entstehen, z.B.:

- **Monitoring/Logging Service** – Zentralisiert das Monitoring aller Agenten. Denkbar ist, Logs aller Dienste an einen zentralen Logger (ELK-Stack o.Ä.) zu senden. Auch ein **MonitoringManager** könnte als separater Dienst laufen, um Metriken von allen Instanzen einzusammeln (CPU, Durchsatz, Fehlerraten) [23]. Dies baut auf den bestehenden Ansätzen (MLflow-Tracking, LoggingUtil) auf, würde aber in der MCP-Architektur vereinheitlicht.
- **API Gateway** – Eine vorgeschaltete Schicht, die externe Anfragen (von UIs, CLI oder Dritt-Systemen) entgegennimmt und an die internen Services weiterleitet. Ein API-Gateway kann ein **einheitliches REST/GraphQL API** nach außen bereitstellen [24], Authentifizierung/Rate-Limits durchsetzen und die Aufrufe an den richtigen internen Dienst routen (z.B. an den Task-Dispatcher für normale Agenten-Queries, oder direkt an den Vector-Store-Service für Wissensmanagement-Endpunkte). In kleineren Systemen kann der Task-Dispatcher diese Rolle auch direkt mit übernehmen; bei wachsender Komplexität ist ein dediziertes Gateway aber sinnvoll für **Uniform Interface** nach außen.

Durch diese Zerlegung entsteht eine *Modular Control Plane*: Jedes Modul steuert einen Teil der Gesamtfunktionalität (Auftragsverteilung, Agentenverwaltung, Sitzungs-Handling, Wissensspeicher, LLM-Anbindung, etc.). Die Module kommunizieren über wohldefinierte APIs miteinander. Die **bestehenden Klassen** werden entweder auf die neuen Dienste verteilt oder ganz ersetzt. Beispielsweise ersetzt der Agent Registry Service intern die bisherigen AgentManager/ModelRegistry-Klassen; der Vector-Store-Service ersetzt direkte Aufrufe an `datastores/vector_store.py` usw. Der SupervisorAgent wird obsolet, da seine Logik im Task-Dispatcher zentral umgesetzt wird. Wichtig ist, dass diese Trennung **keine zusätzliche Schicht obendrauf** sein soll, sondern eine Neustrukturierung darstellt – das System wird vollständig auseinandergezogen und neu organisiert, anstatt den Monolithen nur um ein paar Microservices zu ergänzen.

Geplante MCP-Architektur (Soll-Zustand)

In der neuen Architektur arbeiten die genannten Dienste zusammen, um eine Benutzeranfrage von Anfang bis Ende zu bearbeiten. Ein möglicher Ablauf könnte so aussehen:

1. **Eingabe**: Ein Benutzer stellt eine Anfrage – etwa über die CLI oder eine Web-UI. Diese Anfrage gelangt an den **Task-Dispatcher** (ggf. via API-Gateway). Die Anfrage enthält eine Nutzerfrage oder Aufgabe sowie eine Session-ID, falls es sich um eine fortlaufende Unterhaltung handelt.

2. **Orchestrierung im Task-Dispatcher**: Der Dispatcher-Service validiert zunächst die Anfrage (Optionen für Authentifizierung/Filter durch Security-Manager könnten hier eingebunden werden). Anschließend nutzt er den **Session Manager**, um den bisherigen Kontext zu laden (z.B. frühere Dialognachrichten, Zwischenergebnisse), damit die Agentenauswahl und Antworterzeugung kontextbewusst erfolgen können.
3. Der Dispatcher formuliert nun einen *Auftrags-Plan*: Er bestimmt, welcher Agententyp oder welche Sequenz von Schritten benötigt wird. Dabei kann er auf den **Agent Registry Service** zugreifen, um die **verfügbare Agenten** und deren Fähigkeiten abzufragen. Falls ein ML-basiertes Auswahlmodell (der Nachfolger des AgentSelectorModel aus `training/`) eingesetzt wird, kann der Dispatcher dieses Modell anwenden (evtl. als Teil des Dispatchers selbst oder ausgelagert als kleiner ML-Servicedienst). Dieser Schritt entspricht funktional der heutigen Kombination aus AgentManager/NNManager: Es wird der passende Worker für die User-Query ermittelt [2] [6].
4. Optional: Sollte die Anfrage umfangreiche Wissensrecherchen erfordern (z.B. Nutzer fragt nach domänenspezifischen Informationen), kann der Dispatcher vorab den **Knowledge Vector Store Service** konsultieren. Über dessen API kann eine semantische Suche nach relevanten Informationen durchgeführt werden (z.B. relevante Dokumente oder FAQs abrufen). Die gefundenen Info-Snippets könnten dann dem gewählten Agenten mitgegeben oder direkt in die Prompt-Generierung einfließen (RAG).
5. Optional: Falls komplexe natürliche Sprachverständnis-Vorverarbeitung nötig ist, könnte der Dispatcher selbst den **LLM Service** nutzen (etwa um die Benutzeranfrage zu klassifizieren oder zu analysieren, bevor ein Agent gewählt wird). Denkbar ist hier der Einsatz von *LangChain*-Chains im Dispatcher, um den User-Input zu interpretieren (z.B. zu erkennen, welche Art von Aufgabe gestellt wird).
6. **Delegation an Worker-Agent**: Nachdem der Task-Dispatcher entschieden hat, welcher Agent zuständig ist, übergibt er die Aufgabe an den entsprechenden **Worker Agent Service**. Dies kann synchron per API-Call (z.B. REST/gRPC) oder asynchron via Message Queue erfolgen. Im asynchronen Fall würde der Dispatcher eine Nachricht mit Aufgabenbeschreibung und Kontext auf einen bestimmten Queue-Kanal legen (z.B. Topic = Agententyp oder spezifischer Agentenname), den genau dieser Worker-Service abonniert. Der Worker nimmt die Aufgabe von der Queue und sendet bei Fertigstellung eine Ergebnisnachricht zurück (oder schreibt in einen Response-Queue, welche der Dispatcher überwacht). – Für den MVP könnte zunächst ein synchroner Aufruf per HTTP/gRPC reichen, um Komplexität zu reduzieren, und bei steigendem Durchsatz später auf ein vollasynchrones Muster (Kafka o.ä.) umgestellt werden.
7. **Aufgabenbearbeitung im Worker-Service**: Der spezialisierte Agenten-Dienst empfängt den Auftrag und führt ihn aus. Intern kann er natürlich wieder LLM-Aufrufe tätigen oder Tools verwenden. Viele Agenten werden hier auf den **LLM Gateway Service** zurückgreifen: z.B. um eine Antwort zu formulieren, Code zu generieren oder einen Text zusammenzufassen, sendet der Worker eine Prompt-Anfrage an den LLM-Service (der wiederum das eigentliche LLM – lokal oder via API – anruft). Da der LLM-Service zentral ist, kann er auch LangChain Agents/Tools einsetzen – beispielsweise könnte ein Worker dem LLM-Service mitteilen, welche Tools (Websuche, Code-Ausführung, Datenbankabfrage) im Kontext genutzt werden dürfen, und der LLM-Service führt dann eine vordefinierte LangChain-Tool-Chain aus. (Dies knüpft an die bestehenden Ansätze an, einen AgenticWorker mit LangChain-Fähigkeiten zu haben [13], nur dass es als externer Dienst standardisiert wird.) Während der Ausführung kann der Worker-Service bei Bedarf auch selbst Wissensdaten vom Vector-Store-Service abfragen – etwa ein Dev-Agent, der Code-Dokumentation sucht.
8. **Resultat & Rückgabe**: Sobald der Worker-Agent die Aufgabe erledigt hat, sendet er das Ergebnis zurück an den Task-Dispatcher (bei synchronem RPC als Rückgabewert; bei asynchroner Queue als Ereignis mit Korrelations-ID). Der Task-Dispatcher empfängt das Resultat, aktualisiert ggf. den **Session-Context** (z.B. hängt die Antwort an den Gesprächsverlauf, oder

speichert gewonnene Erkenntnisse für folgende Fragen) und gibt die Antwort an den Benutzer zurück. Gegebenenfalls könnten hier noch **Post-Processing**-Schritte erfolgen, etwa Formatierung der Antwort oder Logging.

9. **Logging/Learning**: Parallel oder im Anschluss können verschiedene Hintergrundaktionen stattfinden: Der **Monitoring/Logging-Service** protokolliert die Interaktion (d.h. Anfrage, ausgewählter Agent, Antwortzeit, Erfolg/Fehler etc.) zentral [22] . Diese Daten können in MLflow oder einer Datenbank festgehalten werden. Ein **Feedback-Loop** kann implementiert werden, indem der Task-Dispatcher/Agent-Registry dem **Agent Selection Model** Feedback gibt: z.B. "Agent X war korrekt/falsch für diese Anfrage", um das Auswahlmodell online zu verbessern (über den NNManager-Nachfolger). Diese Architektur erleichtert zukünftige **A/B-Tests und Training**, da jede Entscheidung und Antwort getrennt erfasst wird und man verschiedene Versionen von Agenten/Modellen gegeneinander testen kann (z.B. mittels eines A/B-Test-Managers, wie im Code vorgesehen [25]).

Technologisches Fundament: Alle neuen Dienste bieten klar definierte **APIs**. Für externe Schnittstellen und leichte Entwicklung bietet sich **FastAPI** an, um schnell RESTful APIs für z.B. den Task-Dispatcher und Vector-Store-Service bereitzustellen. Intern kann man ebenfalls auf REST setzen; bei hoher Performance-Anforderung oder polyglotter Implementation ist **gRPC** jedoch eine gute Wahl (vertraglich festgelegte Schnittstellen, binäre Performance). Ein *Session Manager* könnte z.B. eine schlanke FastAPI haben oder einfach Redis nutzen – etwa indem der Task-Dispatcher direkt auf Redis (als Session-Cache) zugreift und der Session-Service eher eine Logik-Schicht darüber legt. Für das **Messaging** wird Redis bereits im Projekt genutzt [26] , was man für Pub/Sub oder Streams (Redis Streams) einsetzen könnte. Alternativ ist **Kafka** eine robuste Option für die Produktivphase, insbesondere wenn die Anzahl Events hoch ist oder mehrere Instanzen der gleichen Agent-Services im Cluster laufen (Kafka kann als zentraler Event-Bus fungieren). In frühen Phasen kann aber auch ein simpler in-Memory-Queue oder even polling ausreichen, um den Ablauf zu testen, bevor man die volle MQ-Integration vornimmt. **LangChain** wird im neuen Design eine größere Rolle spielen: als Orchestrierungs-Toolkit, um LLM-Aufrufe mit Tool-Nutzung zu kombinieren. So können wir bewährte LangChain-Ketten für Retrieval (Suche im VectorStore + LLM), für Code-Ausführung, Browser-Zugriff etc. einbinden, anstatt alles manuell zu codieren. LangChain erleichtert es z.B. einen Agenten zu bauen, der autonom entscheiden kann, ob er erst das Vector-Store befragt oder direkt ein LLM fragt, etc., was im MCP-Kontext als separater Service oder Library-Funktion genutzt wird.

Schließlich unterstützt die Aufteilung in Microservices die bereits angestrebten Qualitäten: **horizontale Skalierung** (kritische Komponenten wie LLM- oder Vector-Store-Service können bei Bedarf unabhängig skaliert werden) [22] , **Fehlertoleranz** (ein abstürzender Worker-Agent reißt nicht das ganze System mit; der Task-Dispatcher kann bei Timeout einen anderen Worker versuchen), und **klare Komponententrennung** für bessere Wartbarkeit [27] . Neue Agenten oder Modell-Backends können leichter hinzugefügt werden, indem man einfach einen weiteren Service registriert (die Erweiterungsmöglichkeiten wie zusätzliche LLM-Backends oder neue Domänen-Agenten sind explizit vorgesehen) [24] .

Entwicklungsplan: Umsetzung in Phasen

Um die Modernisierung Schritt für Schritt durchzuführen, folgt ein **phasenweiser Plan**, der sowohl die Implementierung der neuen MCP-Services als auch die schrittweise Ablösung des alten Monolithen vorsieht. Jede Phase umfasst konkrete Systemkomponenten, Integrationsschritte und berücksichtigt die Modernisierungsstrategie (inkrementell, mit laufenden Tests).

Phase 1: Grundlegende Architektur und Infrastruktur einrichten

Ziel dieser Phase: Vorbereitung der Umgebung für die Mikroservice-Architektur und Übergang einleiten, ohne sofort Funktionalität zu ändern.

- **Architektur-Blueprint finalisieren:** Auf Basis der obigen Soll-Architektur ein detailliertes Schnittstellen-Design erstellen. Definiere für jeden Service die Responsibilities und API-Endpunkte oder gRPC-Schnittstellen. Lege fest, welche Daten übergeben werden (z.B. Auftrags-JSON für Task-Dispatcher->Worker mit Feldern wie `task_type`, `payload`, `session_id`, etc.).
- **Repo-Struktur anpassen:** Den Code so organisieren, dass die Services getrennt entwickelt werden können. Z.B. Unterordner oder getrennte Python-Pakete für jeden Service (agent_registry/, dispatcher/, workers/, etc.), jeweils mit eigener `main.py` oder eigenem Startpunkt. Alternativ separate Repositories, falls gewünscht – für den Anfang kann es aber in einem Monorepo mit klarer Trennung bleiben, um Refactoring zu erleichtern.
- **Technologie-Entscheidungen treffen:** Wähle die Kommunikationstechnologien pro Schnittstelle. Zum Beispiel: REST (FastAPI) für externe Aufrufe an den Task-Dispatcher und evtl. für interne einfache Calls; gRPC für zeitkritische Pfade wie Dispatcher ↔ LLM-Service; Redis als Message-Bus für Dispatcher → Worker-Kommunikation (sofort vs. später aktivieren). Diese Entscheidungen hier festzurren und Basiskonfigurationen (z.B. Docker-Compose mit Redis/Kafka) bereitstellen.
- **Grundlegende Services als Platzhalter implementieren:** Starte mit dem **Task-Dispatcher-Service**, **Agent-Registry-Service** und **einem Beispiel-Worker-Service** als Skeletons. In dieser Phase können diese Dienste noch Mock-Logik haben. Beispielsweise: Der Task-Dispatcher nimmt einen Dummy-Request an und loggt ihn, fragt den Registry-Dienst (der eine statische Liste von Agenten zurückgibt) und ruft den Dummy-Worker (der immer eine Beispiel-Antwort liefert). Dies stellt sicher, dass die Services kommunizieren können.
- **Basis-Kommunikation testen:** Führe einfache End-to-End-Tests durch: kann man eine Anfrage an den Dispatcher senden und über alle neuen Schnittstellen hindurch eine Antwort erhalten? Zwar ist die Antwort noch Dummy, aber die **Struktur** (HTTP-Aufrufe, evtl. Queue-Nachrichten) soll schon funktionieren. Dieses Grundgerüst bildet die Plattform für die eigentliche Migration der Logik.

Phase 2: Core-MCP-Services implementieren (Agentenregister & Orchestrierung)

Ziel dieser Phase: Die wichtigsten Steuerungsdienste mit echter Funktionalität füllen, während das alte System im Hintergrund noch existiert.

- **Agent Registry Service ausbauen:** Implementiere die Kernfunktionalität der **Agenten-Registry**. Zunächst kann sie statisch konfiguriert werden (z.B. ein YAML/JSON mit verfügbaren Agenten und ihren Fähigkeiten). Später kann man dynamische Registrierung hinzufügen (d.h. Worker-Services melden sich beim Start selbst an). Die Registry sollte zumindest beantworten können: *Welcher Agententyp kann Aufgabe X ausführen?* Dazu evtl. Mapping von Aufgabenkategorien zu Agent-IDs pflegen. Auch Schnittstellen für **Health-Checks** (Worker melden periodisch ihren Status) können hier eingeplant werden.
- **Task-Dispatcher Logik migrieren:** Portiere die Kernlogik des **SupervisorAgent** in den **Dispatcher-Service**. Das umfasst: Analyse der Anfrage, Auswahl des Agenten. Diese Auswahl zunächst regelbasiert umsetzen (z.B. per simpler Wenn-Dann-Regeln oder Keywords der Anfrage), um Parität mit dem bisherigen Verhalten zu erreichen. Integriere den Aufruf an den Agent-Registry-Dienst: Anhand der Anfragekategorie liefert die Registry den zuständigen Agenten (bzw. dessen Service-Adresse).

- **NNManager-Integration (Agentenauswahl-ML):** Binde nun auch den **NNManager** bzw. das ML-Modell für Agentenauswahl ein. Falls bereits ein trainiertes Modell (`AgentSelectorModel`) vorliegt, kann es in den Dispatcher übernommen werden – z.B. als aufrufbare Funktion/Bibliothek innerhalb des Dispatchers (oder optional als separater Microservice *Agent Selection Service*). Ziel: Der Dispatcher entscheidet bei komplexeren Fällen aufgrund von Embeddings/ML (ähnlich wie bisher NNManager) den passenden Agenten [2]. Diese ML-Integration kann optional zunächst stumm geschaltet oder geloggt werden (d.h. Dispatcher entscheidet sowohl per Regel als auch per Modell und loggt beide, um das Modell zu validieren, bevor es "scharf" gestellt wird).
- **Session Manager bereitstellen:** Entwickle den **Session-Service** mit Anbindung an eine schnelle Datenbank (z.B. Redis). Implementiere Endpunkte wie `GET /session/{id}` (liefert Kontext/History) und `POST /session/{id}` (speichert Update, z.B. neue Nachricht). Im Dispatcher ersetze eventuelle globale Zustände durch Aufrufe an den Session-Service. Damit wird Mehrbenutzer- und Multisessionsupport vorbereitet.
- **CLI/API auf neuen Dispatcher umstellen (teilweise):** In dieser Phase kannst du testweise die bestehende CLI oder HTTP-API so modifizieren, dass Anfragen nicht mehr den alten SupervisorAgent aufrufen, sondern an den neuen Task-Dispatcher-Service weitergeleitet werden. Zum Beispiel kann die CLI intern einen HTTP-Call an den Dispatcher machen. So kann man das System bereits durch die neuen Pfade testen, auch wenn die Worker-Antwort noch Dummy oder begrenzt ist. Wichtig ist, parallel weiterhin die alte Logik verfügbar zu halten, bis die neuen Komponenten voll verlässlich sind – in dieser Phase also eher mit Schaltern oder separaten CLI-Kommandos arbeiten, die gezielt den neuen Pfad nutzen (für Testzwecke).

Ergebnis von Phase 2: Das Kontrollzentrum (Control Plane) des Systems ist etabliert: Der Task-Dispatcher orchestriert Aufgaben mithilfe der Agenten-Registry und kann Session-Kontext berücksichtigen. Noch arbeiten die Worker ggf. rudimentär, aber wir haben einen funktionsfähigen Kern, der dem alten Supervisor/AgentManager funktional entspricht (nur als verteiltes System).

Phase 3: Worker-Services und Knowledge-Services implementieren

Ziel dieser Phase: Die ausführenden Teile (Daten und Aktionen) auf die neuen Dienste auslagern und die bestehende monolithische Logik ablösen.

- **Wissensdatenbank-Service (Vector Store) implementieren:** Realisiere den **Vector-Store-Service** basierend auf `datastores/vector_store.py`. Wähle eine geeignete Speicherung (z.B. FAISS in-memory Index oder Redis Vector auf dem vorhandenen Redis). Stelle API-Routen bereit: z.B. `POST /documents` (für neue Dokumente mit vektorweiser Indexierung) und `GET /query?text=...` (für Ähnlichkeitssuche). Verschiebe die Logik aus der bisherigen VectorStore-Klasse in diesen Service. Teste das separat, um sicherzustellen, dass semantische Suchen korrekt funktionieren. Danach integriere den Service ins System: z.B. der Task-Dispatcher nutzt ihn für initiale Wissensrecherche (bei allgemeinen Fragen), und Worker-Services können ihn ebenfalls anfragen, wenn sie domänenspezifische Infos brauchen.
- **LLM Gateway Service implementieren:** Errichte den **LLM-Service** mit Anbindung an mind. einen LLM-Provider (z.B. OpenAI API oder lokalen HuggingFace Model). Beginne mit einfachen Forwarding: der Service bietet z.B. `POST /complete` (nimmt Prompt + Parametern und liefert Completion vom Modell). Übernimm die Logik aus `llm_models/` – z.B. nutze den bestehenden LLMBackendManager und BaseLLM-Klassen innerhalb dieses Dienstes [11]. Anschließend erweitere mit LangChain: Integriere einen **Retrieval Chain** im LLM-Service, so dass dieser Service bei bestimmten Aufrufen selbstständig den Vector-Store-Service konsultieren kann. Beispielsweise könnte der LLM-Service einen Endpunkt `/qa` haben, der einen Query-Text entgegennimmt, relevante Doku über den VectorStore sucht und dann eine LLM-Antwort

generiert (klassisches RAG). So können Worker-Agents diese Intelligenz wiederverwenden, ohne es jeweils selbst zu implementieren. Auch Tools lassen sich hier einbinden: LangChain erlaubt das Definieren von Werkzeugen – der LLM-Service könnte ein "WebSearch-Tool" oder "CodeExec-Tool" kennen, die er bei Bedarf nutzt. (Diese Tools könnten intern wiederum Aufrufe an andere Microservices sein – z.B. ein CodeExec-Tool ruft einen Dev-Agent-Service auf. Solche Cross-calls gilt es aber sorgfältig zu gestalten, um Endlosschleifen zu vermeiden. In Phase 3 reicht es, Basisfälle wie reines QA abzudecken.)

- **Domain-spezifische Worker-Services entwickeln:** Für jede wichtige Agenten-Kategorie wird nun ein eigener Microservice umgesetzt. Priorität haben vermutlich: **Software-Dev-Agent-Service** (für Code-Generierung, Code-Analyse via LLM), **OpenHands-Agent-Service** (für Systemoperationen, z.B. Docker-Tasks) und der geplante **LOH-Agent-Service** (für den speziellen LOH-Bereich). Nimm den Code der bisherigen Agentenklassen (unter `agents/`) und refaktoriere ihn in diese Dienste. Jeder dieser Services benötigt im Grunde: einen Controller/Endpoint (`/execute_task`), der vom Dispatcher aufgerufen wird; und interne Logik, die dem früheren `run()` oder `handle_message()` des Agenten entspricht. Dabei müssen Aufrufe an Knowledge oder LLM nun über die neuen Services erfolgen (also z.B. nicht mehr direkt VectorStore durchsuchen, sondern Anfrage an Vector-Store-Service stellen).
- **Beispiel**: Der DevAgent-Service erhält vom Dispatcher den Auftrag "Schreibe eine Python-Funktion, die X tut". Intern formuliert er einen Prompt für das LLM (ggf. mit Code-Kontext) und ruft den LLM-Service an, bekommt den Code-Entwurf zurück, prüft ihn vielleicht (könnte z.B. via Tools den Code test-kompilieren oder auf Pylint prüfen), und sendet das Ergebnis zurück.
- **Beispiel 2**: Der OpenHands-Service erhält den Auftrag "Starte einen Docker-Container Y". Er könnte intern Shell-Kommandos ausführen (vorsichtig sandboxed) und den Output an den LLM-Service schicken, damit dieser bei Bedarf eine Zusammenfassung formuliert ("Container erfolgreich gestartet."). Dann Response zurück an Dispatcher.
- **Inter-Service Communication fertigstellen:** In dieser Phase sollte der Wechsel auf das echte Kommunikationsmuster erfolgen. Wenn bisher der Dispatcher noch synchron die Worker-Methoden aufgerufen hat (z.B. im selben Prozess zu Testzwecken oder per Direct-HTTP call), kann nun auf asynchrone Queues umgestellt werden, falls sinnvoll. Implementiere z.B. in jedem Worker-Service einen kleinen **Queue-Consumer** (hört auf ein Redis-Kanal oder Kafka-Thema). Der Dispatcher publiziert Aufgaben-Events mit Routing-Information (z.B. `task_type` oder spezifizierter Empfänger). Teste die End-to-End-Verkettung jetzt mit realen Worker-Aktionen:
- Sende über die CLI eine echte Frage/Aufgabe, lasse den Dispatcher via Registry -> Worker delegieren, der Worker nutzt Knowledge/LLM-Services und erzeugt Ergebnis, welches zum Dispatcher zurückfließt.
- Überwache Logs aller Dienste, um zu sehen, dass jede Nachricht korrekt fließt.
- Führe auch Fehlerszenarien vor: z.B. Worker-Service nicht erreichbar – reagiert Dispatcher korrekt (Fehlermeldung ans Frontend, vielleicht Fallback auf anderen Agenten)? Solche Robustheit soll nach und nach eingebaut werden (z.B. Retry-Mechanismen, wie im Konzept des FaultHandler vorgesehen [28]).

Am Ende von Phase 3 sind **alle wesentlichen Funktionen** auf Microservices verteilt. Die alte monolithische Logik (SupervisorAgent, WorkerAgent-Klassen, Manager-Klassen) wird damit faktisch obsolet. Wichtig: Während dieser Implementationsschritte immer wieder Cross-Testing machen, um sicherzustellen, dass keine Leistungsabbrüche oder Fehlfunktionen auftreten. Gegebenenfalls stufenweise vorgehen (z.B. erst DevAgent-Service komplettieren und nutzen, dann den nächsten), statt alles auf einmal – so kann man nach und nach die Altimplementierung ersetzen.

Phase 4: Integration, Testen und Umschalten auf neue Architektur

Ziel dieser Phase: Vollständiger Wechsel auf das neue System und Sicherstellung von Stabilität, Performance und Kompatibilität.

- **Abschaltung des Monolithen:** Sobald alle wichtigen Agenten-Services stabil laufen, kann der alte Monolith stückweise deaktiviert werden. Ersetze intern Aufrufe: z.B. wenn die CLI bisher noch fallback-mäßig den alten Pfad hatte, nun komplett auf die neuen Service-Aufrufe umbiegen. Ebenso für API-Endpunkte – die alten FastAPI-Routen (`api/endpoints.py` etc.) können nun entweder entfernt oder umgestellt werden, dass sie als Proxy zu den neuen Services fungieren. Ziel: Die *gleiche Funktionalität* wird jetzt durch das verteilte System erbracht.
- **Umfassende Testsuite ausführen:** Führe alle vorhandenen Unit- und Integrationstests auf dem neuen System aus. Wahrscheinlich müssen Tests angepasst werden (z.B. weil Funktionen nun via Service-Aufruf statt Direktmethoden arbeiten). Wichtig ist, neue Tests für die verteilten Komponenten zu ergänzen: z.B. ein Integrationstest, der einen kompletten Task von Dispatcher bis Worker und zurück durchspielt [29] . Automatisiere die Tests soweit möglich (Continuous Integration), um bei künftigen Änderungen am verteilten System schnell Regressionen zu erkennen.
- **Leistungs- und Lasttests:** Da Microservices andere Performance-Charakteristika haben, gezielt messen: Wie ist die Latenz einer Anfrage durch alle Dienste hindurch? Wo entstehen Bottlenecks? Beispiel: Falls der LLM-Service häufig aufgerufen wird, stellt sicher, dass er ggf. mit mehreren Instanzen oder einem Async-Server laufen kann, um parallele Requests zu bedienen. Prüfe die **Skalierung**: man sollte nun probeweise einen Worker-Service mehrfach instanziieren können und beobachten, ob der Durchsatz linear steigt (ggf. erfordert das in Kafka-Themen Partitions oder in Redis Queue Namenskonventionen, aber das kann man einplanen).
- **Fehlerfälle und Recovery:** Simuliere den Ausfall eines Services (z.B. Worker stürzt ab). Der Dispatcher sollte nicht hängen bleiben: Implementiere Timeouts und Fehlermeldungen. Ggf. den Registry-Service nutzen, um erkannte Offline-Agenten aus dem Pool zu nehmen, und bei Anfrage einen Fehler zurückgeben wie "Service derzeit nicht verfügbar". Durch die lose Kopplung über Messaging kann ein Worker-Absturz das System nicht komplett stoppen – dieses Verhalten validieren.
- **Sicherheit überprüfen:** Da nun verteilte Komponenten übers Netzwerk kommunizieren, müssen Sicherheitsaspekte betrachtet werden. In dieser Phase z.B.: Authentifizierungs- und Autorisierungsmechanismen zwischen den Services (könnte simpel mit gemeinsamen Secrets/API-Keys oder JWT erfolgen, je nach Bedarf). Auch Input-Validation zentralisieren – eventuell als Teil des API-Gateways oder Security-Manager-Service, damit z.B. ein bösartiger Input nicht einen Worker zum Absturz bringt.
- **Optimierungen einbauen:** Jetzt, da alles läuft, können Optimierungen umgesetzt werden: z.B. **Caching** – der LLM-Service könnte letzte Ergebnisse cachen, um bei identischen Fragen schneller zu antworten (CacheManager nutzen [30]). Oder der Vector-Store könnte Ergebnisse zwischenpuffern. Ebenso Monitoring-Optimierung: Setup von Dashboards für die gesammelten Metriken (z.B. Grafana mit Daten aus dem Monitoring-Service).
- **Migration bestehender Daten:** Falls im alten System Wissensdaten schon in Dateien oder Datenbanken liegen (Vector-Store-Daten, Dokumente in WorkerAgentDBs), überführe diese in den neuen Vector-Store-Service/DB, damit kein Wissensverlust entsteht. Ähnlich, migriere MLflow-Experimente oder Modelle in den neuen Model/Registry-Strukturen (kann sein, dass ModelRegistry im neuen Setup einfach als Teil des LLM-Service fungiert oder separat bleibt).

Nach Phase 4 sollte **Agent-NN vollständig auf der MCP-Architektur laufen**. Alle Benutzer-Interaktionen passieren über die neuen Services, und die alten Module sind nicht länger im Einsatz. Das System ist nun modular, skalierbar und leichter wartbar.

Phase 5: Abschließende Modernisierungsschritte und Ausblick

Ziel dieser Phase: Feinschliff, Dokumentation und Vorbereitung für Weiterentwicklung.

- **Dokumentation aktualisieren:** Überarbeite die Projekt-Dokumentation, sodass sie die neue Architektur reflektiert. Architektur-Diagramme neu zeichnen (zeigen Services und ihre Interaktionen). In `docs/architecture/` neue Übersichten erstellen. Beschreibe die API der einzelnen Microservices (evtl. in einem zentralen **API-Dokument** oder mit OpenAPI-Spezifikationen aus den FastAPI-Services generiert). Auch die Entwickler-Dokumentation (`Code-Dokumentation.md`) anpassen, um die Verlagerung der Komponenten in Services zu erläutern.
- **Schulung/Teamübergabe:** Da dies eine große Umstellung ist, alle Beteiligten Entwickler mit der neuen Struktur vertraut machen. Möglicherweise **Contributing Guide** erweitern, z.B. wie man einen neuen Agent-Service hinzufügt oder wie Logging jetzt funktioniert.
- **Deployment vorbereiten:** Die Microservices sollten containerisiert sein (Dockerfiles für jeden Dienst). Erstelle eine Docker-Compose oder Helm Charts, um das gesamte System konsistent zu deployen. Teste das Deployment auf einer Staging-Umgebung. Hier zeigt sich die Stärke der neuen Architektur: einzelne Dienste können auf verschiedene Hosts verteilt oder skaliert werden [31].
- **Gradual Rollout (wenn nötig):** Falls das System in einer Produktionsumgebung genutzt wird, erwäge einen stufenweisen Rollout. Z.B. erst die neuen Services intern einsetzen und Ergebnisse mit dem alten System vergleichen (Shadow Mode), bevor man ganz umschaltet. Da laut Aufgabenstellung allerdings der alte Monolith komplett abgelöst werden soll, kann man nach erfolgreichem Staging-Test einen Big-Bang-Switch wagen – aber immer mit Fallback-Plan.
- **Weiterentwicklungsmöglichkeiten:** Mit der MCP-Architektur eröffnen sich etliche Möglichkeiten: Ein **Frontend-WebUI** kann nun relativ einfach das REST-API des Task-Dispatchers nutzen, um ein schönes Interface anzubieten [24]. Neue Agenten für weitere Domänen (z.B. Finance, Marketing) lassen sich als eigene Services anstöpseln, ohne das Kernsystem zu beeinflussen [24]. Man könnte einen **GraphQL-Gateway** einführen, um komplexe Queries an mehrere Services gleichzeitig zu ermöglichen. Ebenso kann an **federated learning** gedacht werden, bei dem mehrere Deployments Wissen oder Modelle austauschen [32]. In naher Zukunft wäre es sinnvoll, das Agent-Auswahlmodell kontinuierlich mit den gesammelten Interaktionsdaten zu verbessern (Online-Learning mit dem AdaptiveLearningManager-Ansatz [33]).

Modernisierungsstrategie zusammengefasst: Durch die Umsetzung in diesen Phasen wird das System Schritt für Schritt transformiert, ohne lange Zeit komplett offline zu sein. Kritische Teile werden zuerst dupliziert und getestet (Supervisor → Dispatcher, AgentManager → Registry), bevor alte Komponenten stillgelegt werden. Wichtig ist, in jeder Phase ein lauffähiges Teilsystem zu haben – so kann man bei Problemen leichter einen Schritt zurück. Die vorgeschlagenen Technologien (FastAPI/gRPC, Redis/Kafka, LangChain etc.) werden gezielt dort eingesetzt, wo sie am meisten Nutzen bringen – z.B. FastAPI für schnelles Implementieren von Service-APIs, Kafka für robuste Async-Kommunikation, LangChain für LLM-Orchestrierung – und sie ersetzen proprietäre Lösungen des alten Systems (etwa CommunicationHub) durch **bewährte Standard-Patterns**. Am Ende steht ein **vollständig MCP-basiertes Agent-NN**: Ein System, das aus modularen, lose gekoppelten Services besteht, welche zusammen eine intelligente, LLM-gestützte Multi-Agent-Plattform bilden – fit für zukünftige Erweiterungen und den produktiven Einsatz.

Quellen: Die Analyse basiert auf der bestehenden Code-Dokumentation und Projektstruktur von Agent-NN [34] [1], sowie bewährten Architekturmustern für verteilte KI-Systeme [22] [24].

---

1 4 9 14 15 overview.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/b9ab218b42c559c1734d695b67feb9e4a7ccc152/docs/architecture/overview.md

2 8 17 18 34 analysis.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/b9ab218b42c559c1734d695b67feb9e4a7ccc152/docs/architecture/analysis.md

3 5 6 7 10 11 12 13 16 21 22 23 24 25 26 27 28 30 31 32 33 Code-Dokumentation.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/b9ab218b42c559c1734d695b67feb9e4a7ccc152/docs/architecture/Code-Dokumentation.md

19 20 Gen-AI Powered Control for ECS, EKS & Lambda Workloads | by Hariharan Eswaran | May, 2025 | Medium
https://medium.com/@hariharan.eswaran/gen-ai-powered-control-for-ecs-eks-lambda-workloads-944af0309d99

29 ROADMAP.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/b9ab218b42c559c1734d695b67feb9e4a7ccc152/ROADMAP.md