ChatGPT

TODO-Liste für Agent-NN Entwicklungsplan

Phase 1: Codebase-Konsolidierung & Architektur-Refaktor

1.1 Architektur-Grundlagen

- [ ] **Legacy-Code identifizieren und Struktur vorbereiten:** Analyse der bestehenden Codebase, um zu refaktorierende Module zu bestimmen. Verzeichnisstruktur für die neue Architektur anlegen (z.B. Ordner für Microservices) und einen `archive/` Ordner einrichten, in den alter Code verschoben statt gelöscht wird (Code bleibt als Referenz erhalten). *(Kein Legacy-Code löschen, nur umorganisieren.)*
- [ ] **ModelContext-Datentyp definieren:** Einen zentralen `ModelContext` Typ erstellen, der alle nötigen Informationen für KI-Modelle und den Kontext enthält (z.B. Konfigurationsparameter, Zustandsdaten) [1]. Dieser Datentyp bildet die Grundlage für die Kommunikation zwischen den neuen Services.
- [ ] **Grundservices als Microservices anlegen:** Die Kernkomponenten des Systems als eigenständige Services implementieren [1]. Dazu zählen insbesondere: Dispatcher-Service, Agent-Registry-Service, Session-Manager-Service, Vector-Store-Service und LLM-Gateway-Service. Jeder Service erhält ein eigenes Modul/Repository (gemäß geplanter Microservice-Architektur) und eine minimale API (z.B. via FastAPI oder gRPC) entsprechend seiner Zuständigkeit.
- [ ] **Docker-Umgebung einrichten:** Ein `docker-compose.yml` erstellen, das alle neuen Microservices konfiguriert und gemeinsam startbar macht [1]. Sicherstellen, dass jeder Service in einem Container läuft und die Services sich über definierte Netzwerke finden (inkl. Datenbank oder Redis falls benötigt).
- [ ] **Ersten End-to-End-Test durchführen:** Nachdem Grundservices stehen, einen einfachen End-to-End-Test über alle neuen Services aufsetzen [1]. Beispielsweise einen Dummy-Request durch den Dispatcher über das LLM-Gateway bis zu einem Worker-Agenten schleusen, um die Kommunikationswege und Integration der Microservices frühzeitig zu validieren.

1.2 Kernservices & Kontext-Integration

- [ ] **MCP-SDK integrieren & Basis-Routing implementieren:** Das vorhandene MCP-SDK ins System einbinden [2]. Die neuen Services so verbinden, dass Anfragen vom Dispatcher anhand des `ModelContext` an die richtigen Worker/Agenten weitergeleitet werden. Eine grundlegende Routing-Logik entwickeln, die z.B. anhand von Modell- oder Fähigkeits-Typ im `ModelContext` entscheidet, welcher Service die Anfrage bearbeitet.
- [ ] **LLM-Gateway-Anbindung umsetzen:** Den LLM-Gateway-Service implementieren, der als Schnittstelle zu externen Sprachmodell-APIs (OpenAI o.ä.) dient [2]. Sicherstellen, dass der `ModelContext` korrekt übergeben wird und das Gateway die Anfrage mit allen relevanten Parametern an das LLM weiterleitet. Die Rückgaben des LLM sollten an den entsprechenden Worker-Agent zurückgespielt werden.
- [ ] **VectorStore-Service integrieren (Wissensbasis):** Einen Vector-Store-Service aufsetzen, der Wissenseinträge als Embeddings speichert und Abfragen darauf erlaubt [2]. Dies ermöglicht es Agenten, in Dokumentationen/Knowledge-Bases zu suchen. Den Agenten so erweitern, dass bei Bedarf dieser Service genutzt wird (z.B. Embedden von eingehenden Fragen und Similarity Search im Vector Store).

- [ ] **Session-Manager mit persistentem Kontext:** Den Session-Manager-Service implementieren, der Benutzer-Sessions und Konversationskontexte verwaltet [2]. Zustand über Anfragen hinweg (z.B. Chat-Historie, vergangene Antworten) sollte hier gespeichert werden, ggf. in einer Redis- oder Datenbank, um eine persistente Unterhaltung zu ermöglichen. Dienste wie Dispatcher/Worker müssen den Session-Service konsultieren, um Kontext zu laden oder zu speichern.

1.3 Infrastruktur & Sicherheit

- [ ] **Monitoring- und Logging-Infrastruktur einführen:** Querschnittliche Überwachung einbauen [3]. Z.B. einen zentralen Logging-Mechanismus (über alle Microservices konsistente Log-Formate, ggf. ELK-Stack-Anbindung) und grundlegendes Monitoring (Health-Endpoints für Services, Metriken wie Request-Zeiten). Dadurch kann die Stabilität der verteilten Architektur beobachtet werden.
- [ ] **Sicherheits-Layer implementieren:** Einen Authentifizierungs- und Autorisierungsmechanismus vor die Services schalten [3]. Etwa API-Gateway oder jedes Service-API mit JWT-Überprüfung ausstatten, sowie Rate Limiting pro Endpoint einführen, um Missbrauch zu verhindern. Dieser Security-Layer soll alle neuen Microservices einheitlich schützen (ggf. mittels eines Gateway-Proxys oder gemeinsamer Middleware in jedem Service).
- [ ] **Persistente Speicherpfade & Konfiguration:** Festlegen, wo persistente Daten abgelegt werden (Datenbank-Schemas, Dateiverzeichnisse) und sicherstellen, dass diese in Docker-Volumes gemountet sind [3]. Pfade für Models, Logs, temporäre Dateien etc. standardisieren. Außerdem Konfigurationsdateien anlegen (z.B. `.env` oder YAML-Konfigurationen), damit sensitive Daten und Umgebungsvariablen zentral verwaltet werden können. Diese Struktur soll Portabilität ermöglichen, damit das System leicht auf andere Umgebungen übertragen werden kann (Stichwort portable Datenverzeichnisse).

1.4 Entwickler-Tools & Release-Vorbereitung

- [ ] **Developer-SDK und CLI bereitstellen:** Ein Entwickler-SDK (z.B. Python-Bibliothek unter `mcp/` Verzeichnis) erstellen, das grundlegende Funktionen zum Interagieren mit dem Agent-NN-System bietet [4]. Ebenso ein CLI-Tool entwickeln, mit dem Admin-Aufgaben oder Tests (z.B. ModelContext erzeugen, Services ansteuern) ausgeführt werden können. Diese Tools sollen die Entwicklung und Nutzung des Systems erleichtern und auf der neuen Architektur aufsetzen.
- [ ] **Erste Beta-Release vorbereiten:** Nachdem die obigen Schritte abgeschlossen sind, die Anwendung in einen ersten lauffähigen Zustand bringen und eine Beta-Version veröffentlichen [4]. Dazu gehört, eine Versionsnummer zu vergeben, Changelog zu skizzieren und ggf. ein Release-Branch aufzusetzen. Der Beta-Release dient intern dazu, Feedback zur neuen Architektur einzuholen. **Wichtig:** Der alte Monolith oder bisherige Code soll bis hierhin weiterhin im `archive/` Verzeichnis existieren, aber die Beta-Version stützt sich vollständig auf die neue Microservice-Architektur.

Phase 2: Feature-Integration & Harmonisierung

2.1 Meta-Learning-Integration

- [ ] **MetaLearner aktivieren:** Den Meta-Learning-Komponenten im System einschalten und integrieren, insbesondere den `MetaLearner` im `NNManager` modul aktivieren [5]. Dadurch sollen Agenten aus vergangenen Interaktionen lernen können, um ihre Antworten oder Aktionen zukünftig zu verbessern.
- [ ] **AutoTrainer und Feedback-Schleife implementieren:** Einen automatischen Trainingsprozess entwickeln [5]. Dieser AutoTrainer soll anhand von gesammeltem Feedback (z.B. Nutzerbewertungen der Antworten oder erreichte Ziele) Modelle nachjustieren. Entwerfen

einer Feedback-Schleife, in der Agenten-Outputs bewertet und als Trainingssignal zurückgeführt werden, um Modelle iterativ besser zu machen.

- [ ] **Capability-basiertes Routing prototypisieren:** In einem Worker-Agent einen Mechanismus erproben, der Anfragen je nach benötigter Fähigkeit an spezialisierte Sub-Modelle oder Agenten routet [5] . Z.B. erkennt der Agent, ob eine Anfrage programmieren, zeichnen oder planen erfordert, und leitet sie intern an das jeweils passend trainierte Modell weiter. Dieses Routing zunächst als Prototyp implementieren und testen.
- [ ] **Evaluationsmetriken sammeln:** Metriken definieren und erfassen, um den Erfolg der neuen Lernmechanismen zu messen [5] . Beispielsweise Genauigkeit der Antworten vor vs. nach Lernen, Konversationslängen, Erfolgsquote von Agenten bei Aufgaben etc. Diese Metriken regelmäßig loggen und für spätere Auswertung speichern (z.B. in einem Dashboard oder Log).

2.2 LLM-Provider-Integration & konfigurierbare KI

- [ ] **Provider-System erweitern:** Unterstützung für verschiedene KI-Modelle und Anbieter implementieren. Dazu Provider-Klassen und eine Factory im LLM-SDK entwickeln, um je nach Konfiguration unterschiedliche LLM-Backends (z.B. OpenAI, lokale Modelle) ansprechen zu können [6] . Dies ermöglicht es, das zugrundeliegende Modell austauschbar zu machen, ohne den Code der Agenten zu ändern.
- [ ] **Dynamische Konfiguration über `llm_config.yaml`:** Die Modelle und Provider sollen vollständig über externe Config-Dateien steuerbar sein [6] . Eine zentrale `llm_config.yaml` einführen/erweitern, in der Modellparameter, API-Keys, Provider-Typ etc. festgelegt werden. Der Code lädt diese Einstellungen zur Laufzeit, sodass Änderungen an der Datei (z.B. Wechsel des LLM-Anbieters) ohne Codeänderung wirksam werden.
- [ ] **SDK-Beispiele und Tests hinzufügen:** Für das erweiterte SDK mit Provider-Funktionalität Beispielanwendungen und Unit-Tests erstellen [6] . Dadurch wird sichergestellt, dass die Provider-Auswahl wie erwartet funktioniert. Mindestens einen Unit-Test pro Provider-Typ implementieren und ein kleines Beispielskript dokumentieren, das zeigt, wie ein Entwickler das SDK nutzt, um zwischen verschiedenen LLM-Providern zu wechseln.

2.3 System-Harmonisierung & Legacy-Migration

- [ ] **Bestehende Funktionen angleichen:** Alle noch nicht migrierten Alt-Funktionen oder Module aus dem vorherigen System in die neue Architektur überführen. Falls bestimmte Geschäftslogiken bislang nur im monolithischen Legacy-Code existieren, entsprechende Pendants in den neuen Microservices implementieren. Ziel ist, dass funktional nichts verloren geht. Beispielsweise sicherstellen, dass alle Agenten-Fähigkeiten oder Tools, die im alten System vorhanden waren, nun im neuen System (ggf. anders aufgeteilt) wieder verfügbar sind.
- [ ] **Konsistenz und Abwärtskompatibilität prüfen:** Überprüfen, dass das neue System alle Anwendungsfälle des alten Systems abdeckt. Tests mit Beispieldaten vom alten System durchführen und Ergebnisse vergleichen. Etwaige Unterschiede dokumentieren oder das neue System entsprechend nachbessern, sodass es sich konsistent verhält. Nutzer der alten Schnittstellen sollten (wenn möglich) keinen Funktionsverlust erleben.
- [ ] **Legacy-Code schrittweise stilllegen (archivieren):** Sobald eine alte Komponente durch eine neue ersetzt wurde, den betreffenden Legacy-Code in den `archive/` Ordner verschieben. Keine entfernten Features ohne Ersatz belassen. Die alten Module sollten nicht mehr aktiv im Build/Deployment genutzt werden, aber als Referenz im Archiv verbleiben. Dieser Schritt erfolgt fortlaufend für jede Funktion, sodass am Ende von Phase 2 kein alter Code mehr aktiv genutzt wird, sondern maximal noch im Archiv existiert.

Phase 3: Konsolidiertes Frontend & User Experience

3.1 Vollständige Frontend-Implementierung

- [ ] **Alle UI-Seiten/Panels fertigstellen:** Umsetzung aller geplanten Frontend-Komponenten und Seiten abschließen (zurzeit teilweise als Platzhalter oder Mock vorhanden). Insbesondere alle Dashboard-Panels implementieren, die im Konzept vorgesehen sind: z.B. *Knowledge*-Panel, *Monitoring*-Panel, *Security*-Panel, *Testing*-Panel, *Settings*-Panel, *Logs*-Panel, *Docs*-Panel usw., sodass die UI einen vollständigen Funktionsumfang bietet [7]. Jedes Panel soll die entsprechenden Daten anzeigen und Aktionen ermöglichen (z.B. Security-Panel für Rechteverwaltung, Monitoring-Panel für Systemmetriken etc.).
- [ ] **Frontend mit Backend-APIs verbinden:** Sämtliche bisher im Frontend verwendeten Mock-Daten durch echte API-Aufrufe ersetzen [8]. Die React-Frontend-App muss die neuen Microservice-Endpunkte nutzen (z.B. via REST-Aufrufe an den API-Gateway oder direkte Service-Endpoints, sowie WebSocket-Verbindungen für Echtzeitfunktionen). Sicherstellen, dass Chat-Nachrichten über den WebSocket-Kanal laufen und z.B. Agenten-Auswahl und Task-Management über entsprechende API-Routen gesteuert werden. Die Fehlerbehandlung und Response-Verarbeitung an diesen Stellen sollte robust sein.
- [ ] **Einheitliche Benutzeroberfläche konsolidieren:** Etwaige getrennte UI-Komponenten zusammenführen. Falls es zuvor separate Oberflächen (z.B. ein Admin-Tool oder eine einfache Chat-Webseite und ein separates Dashboard) gab, diese in die neue React-Anwendung integrieren, sodass Nutzer alles in einer konsolidierten App vorfinden. Dies kann bedeuten, ein zentrales Navigationselement (Sidebar/Header) bereitzustellen, über das alle Funktionen erreichbar sind (Chat, Verwaltung der Agenten, Monitoring, Einstellungen etc.). Damit wird die User Experience vereinheitlicht und der Pflegeaufwand reduziert, da nur noch eine Frontend-Codebase existiert.

3.2 UX-Optimierungen (Responsivität & Barrierefreiheit)

- [ ] **Authentifizierungs-Flow verbessern:** Den Login-/Authentifizierungsablauf im Frontend optimieren [9]. Falls noch nicht vorhanden, JWT-basierte Authentifizierung vollständig integrieren, inklusive Speicherung von Tokens im Frontend (secure) und automatischer Token-Erneuerung (Refresh-Token-Mechanismus). Zusätzlich sicherstellen, dass die UI rollenbasierte Zugriffskontrollen berücksichtigt (z.B. Admin-Bereich nur für Admins sichtbar/funktional).
- [ ] **Fehlerbehandlung und Ladeindikatoren:** Umfassende Fehler- und Ladeszenarien abdecken [8]. Globale Error Boundaries in React einführen, um unbehandelte Fehler abzufangen [10]. Nutzerfreundliche Fehlermeldungen anzeigen (z.B. Toasts oder Dialoge mit verständlichen Hinweisen, statt bloßer Console-Logs). Ebenso Ladeindikatoren (Spinner/Progress) überall dort einbauen, wo Daten vom Server geladen werden oder längere Aktionen laufen, damit der Nutzer Feedback über den Status erhält.
- [ ] **Responsive Design & Dark-Mode sicherstellen:** Überprüfen, dass die gesamte Anwendung responsiv gestaltet ist, also auf verschiedenen Bildschirmgrößen gut funktioniert. Falls nötig, CSS-Anpassungen vornehmen, Breakpoints testen usw. Zudem den Dark/Light-Mode konsistent umsetzen (ggf. über Tailwind CSS Klassen oder eine Theme-Switch Komponente) – sicherstellen, dass alle neuen Panels und Komponenten in beiden Themes lesbar und ansprechend sind. (Vorarbeit ist schon geleistet, z.B. dunkles/helles Theme vorhanden [11], aber neue Elemente daran anpassen.)
- [ ] **Zugänglichkeit (Accessibility) erhöhen:** Die Web-Oberfläche auf Barrierefreiheit prüfen und optimieren [12]. Dazu gehört: vollständige Tastatur-Navigation (alle interaktiven Elemente per Tab erreichbar), Screenreader-Unterstützung durch ARIA-Labels und semantische HTML-Strukturen, ausreichende Farbkontraste für Texte und Bedienelemente sowie aussagekräftige

Alternativ-Texte bei Icons/Bildern. Ziel ist es, WCAG-Konformität soweit wie möglich zu erreichen, um die Anwendung für alle Nutzer zugänglich zu machen.

- [ ] **Formularvalidierung und Usability:** In allen Benutzereingabeformularen (z.B. Einstellungen, neue Agenten anlegen, Task-Erstellung) eine Echtzeit-Validierung der Eingaben einbauen [13]. Fehlermeldungen bei ungültigen Eingaben sollen deutlich und nahe am entsprechenden Feld erscheinen. Darüber hinaus ggf. Hilfetexte oder Platzhalter ergänzen, um dem Nutzer das Ausfüllen zu erleichtern. Allgemein die Benutzerführung verbessern, z.B. durch verständliche Button-Bezeichnungen, Bestätigungsdialoge vor kritischen Aktionen und Fokus-Management nach Aktionen, um ein reibungsloses Nutzungserlebnis zu bieten.
- [ ] *(Optional)* **Erweiterte UI-Funktionen integrieren:** Falls zeitlich machbar, zusätzliche Komfortfunktionen in die Oberfläche aufnehmen. Zum Beispiel: Drag-and-Drop-Unterstützung für Datei-Uploads oder das Reihum-Anordnen von Komponenten, personalisierte Dashboard-Layouts, oder ein Benachrichtigungszentrum für Systemmeldungen. Diese Punkte sind optional und können auch in einem späteren Release folgen, würden aber das Nutzererlebnis weiter abrunden.

Phase 4: Testabdeckung, CI/CD und Dokumentation

4.1 Testabdeckung & CI-Pipeline

- [ ] **Umfassende Test-Suite entwickeln:** Eine vollständige automatisierte Test-Suite mit **pytest** erstellen, die alle Kernmodule des Systems abdeckt [14]. Dazu gehören Unit-Tests für einzelne Funktionen sowie Integrations-Tests über mehrere Microservices hinweg. Wo nötig, Mocking einsetzen (z.B. für externe API-Aufrufe oder Datenbankzugriffe), um realistische Testfälle zu gestalten. Angestrebte Code Coverage ist etwa **90%** der wichtigen Module [15] – wichtiger als die Prozentzahl ist, dass alle kritischen Pfade (Authentifizierung, Haupt-Features etc.) durch Tests abgesichert sind.
- [ ] **Regressionsprüfung & Bugfixing:** Bei der Einführung der Testsuite alle gefundenen Fehler beheben. Sicherstellen, dass neu geschriebene Tests zunächst fehlschlagen, wenn ein Bug besteht, und nach dem Fix grünes Licht geben. Kontinuierlich während der Entwicklung Tests ausführen, um Regressionen früh zu erkennen – insbesondere nach größeren Refaktorierungen oder dem Zusammenführen von Feature-Branches. Kein neues Feature abschließen, ohne entsprechende Tests, so dass die Stabilität des Systems bis zum Ende der Phase stetig steigt.
- [ ] **Continuous Integration einrichten:** Eine CI-Pipeline (z.B. via GitHub Actions) aufsetzen, die bei jedem Commit automatisch das Projekt baut und alle Tests ausführt [14]. In die Pipeline folgende Schritte integrieren: Linting (z.B. mit ruff oder Flake8, um Stilverletzungen zu finden), statische Code-Analyse/Type-Checking, Format-Überprüfung (Black), Ausführen der pytest-Suite und Messen der Testabdeckung. Die Pipeline soll bei Fehlern entsprechend failen und im Repository als Status sichtbar sein. So wird vor dem Merge von Änderungen die Codequalität sichergestellt.
- [ ] **Qualitätsmetriken beobachten:** Die Ergebnisse der CI (Linter-Warnungen, Testcoverage-Reports) regelmäßig prüfen und als Team besprechen. Falls bestimmte Module noch unzureichend getestet sind oder häufig Fehler auftreten, gezielt zusätzliche Tests oder Refaktorierungen einplanen. Das Ziel ist zum Ende von Phase 4 ein weitgehend fehlerfreies, wartbares System, das allen wichtigen Coding-Standards genügt.

4.2 Deployment & Dokumentation

- [ ] **Deployment-Skripte finalisieren:** Die Docker-Compose-Konfiguration aus Phase 1 finalisieren und für den Produktionseinsatz optimieren [14]. Volle Überprüfung, ob alle Services korrekt verlinkt sind, Start-Reihenfolge passt, Persistenz-Volumes definiert sind und

Sicherheitsaspekte (z.B. nur notwendige Ports geöffnet) berücksichtigt wurden. Gegebenenfalls Dockerfiles der einzelnen Services optimieren (kleinere Images, Security-Best-Practices). Falls ein Deployment auf Cloud oder Kubernetes geplant ist, entsprechende Helm-Charts oder Manifeste vorbereiten. Ziel: Ein Klick/ Befehl zum Aufsetzen der gesamten Umgebung.

- [ ] **Dokumentation aktualisieren:** Alle Projektdokumente auf den neuesten Stand bringen [14]. Dazu zählt die README.md (kurze Anleitung und Verweis auf weitere Doku), detaillierte Installations- und Usage-Guides, sowie Developer-Dokumentation für die Architektur. Insbesondere ein aktualisiertes **Architektur-Diagramm** erstellen, das die neue Microservice-Struktur (Dispatcher, Registry, etc.) und deren Zusammenspiel zeigt [14]. Dieses Diagramm in die Doku einbinden. Außerdem **CHANGELOG** oder Migrationshinweise schreiben, damit Entwickler die Änderungen nachvollziehen können.
- [ ] **SDK- und API-Dokumentation ergänzen:** Spezifische Dokumentationsteile, die in vorherigen Phasen vorbereitet wurden, nun fertigstellen. Zum Beispiel eine Dokumentation der LLM-Provider-SDK Nutzung unter `docs/sdk/` verfassen [6], die erklärt, wie man das SDK in eigenen Projekten verwendet. Ebenso die API-Dokumentation (z.B. Swagger/OpenAPI Doku unter `/docs`) überprüfen und an neue Endpunkte anpassen. Alle "TODO" oder "Coming soon" Platzhalter in der Doku müssen entfernt werden – vollständige Beschreibungen an deren Stelle.
- [ ] **Abschließender Usability-Test:** Ein Teammitglied (oder neuer Nutzer) soll das System frisch aufsetzen nur anhand der Dokumentation. Dazu Repository klonen, Installation durchführen und eine Ende-zu-Ende Nutzung (z.B. einen Beispiel-Chat mit dem Agenten) ausprobieren. Etwaige Lücken in der Anleitung oder Fehler in der Startprozedur dokumentieren und beheben. Dieser Schritt stellt sicher, dass ein Außenstehender das Projekt anhand der Dokumentation verstehen und nutzen kann, was wichtig für Onboarding und Open-Source-Veröffentlichung ist.
- [ ] **Release-Vorbereitung:** Die Versionierung des Projekts für einen Stable Release anheben (z.B. v1.0.0 markieren, falls vorher Beta-Stände waren). Einen letzten Check aller LICENSE- und CONTRIBUTING-Hinweise durchführen. Anschließend einen Release-Tag in Git setzen und eine Release-Beschreibung verfassen (wichtige Änderungen, neue Features aus diesen Phasen hervorheben). Optional: Bekanntgabe oder Ankündigung des Releases intern/extern vorbereiten.
- [ ] **Legacy-Code endgültig archivieren:** Abschließend sicherstellen, dass sämtlicher alter Code, der durch die neue Implementation ersetzt wurde, im `archive/` Ordner liegt und nicht mehr Teil des aktiven Deployments ist. In der Dokumentation vermerken, welche Module als deprecated gelten. Der Legacy-Code bleibt im Repo zwar erhalten (für historische Zwecke), wird aber nicht mehr weiterentwickelt. Damit ist die Transition zum neuen System vollständig abgeschlossen.

---

[1] [2] [3] [4] [5] [6] [14] ROADMAP.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/ROADMAP.md

[7] [8] [9] [10] [11] [12] [13] Roadmap.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/monitoring/Roadmap.md

[15] AGENTS.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/904819970351796a2fda272015e499a703926007/AGENTS.md