ChatGPT

Entwicklungsplan für das Agent-NN Framework

1. Analyse der aktuellen Architektur

Übersicht: Das Projekt *Agent-NN* ist als Multi-Agenten-System mit eingebetteter KI-Funktionalität aufgebaut. Ein zentraler **SupervisorAgent** koordiniert eingehende Benutzeranfragen und delegiert sie an spezialisierte **WorkerAgents** [1]. Die **WorkerAgents** verfügen über eigene Wissensspeicher (Vektor-Datenbanken) und nutzen Sprachmodelle (LLMs) sowie interne neuronale Netze, um Aufgaben in ihrem Fachgebiet zu lösen [2] [3]. Wichtige Komponenten und ihre Rollen sind in der folgenden Tabelle zusammengefasst:

Komponente	Rolle im System	Relevante Klassen/Module
LLM-Integration	Anbindung an Sprachmodelle (LLMs) – ermöglicht Textgenerierung und Embedding entweder über externe APIs oder lokal. Bei vorhandenem OpenAI-Key wird die OpenAI-API genutzt, sonst ein lokales Modell (z.B. Llama 2) [4] [5].	`llm_models/base_llm.py` (Basisklasse für LLM-Zugriff) [4] [5]; `llm_models/specialized_llm.py` (domänenspezifische LLMs).
Vektor-Speicher	Wissensdatenbank für Dokumente pro Agent, ermöglicht semantische Suche via Embeddings (Chroma-Datenbank) [6]. Jeder WorkerAgent hat eine eigene Collection zur Ablage seines Wissens.	`datastores/vector_store.py` (Chroma Integration) [7]; `datastores/worker_agent_db.py` (Datenbank für Agentenwissen) [8].
WorkerAgent (Ausführungs-Agent)	Fachspezifischer Agent, der eine Anfrage im eigenen Domänenkontext bearbeitet. Er verfügt über einen Wissensspeicher, nutzt ein spezialisiertes LLM für sein Gebiet und ein internes neuronales Netz (*AgentNN*) zur Feature-Berechnung [2]. WorkerAgents können untereinander kommunizieren, um bereichsübergreifende Informationen auszutauschen.	`agents/worker_agent.py` (Implementierung der Worker-Agents) [2]; nutzt `SpecializedLLM`, `WorkerAgentDB` und `AgentNN`.

Komponente	Rolle im System	Relevante Klassen/Module
SupervisorAgent (Koordinator)	Verwaltungs-Agent auf Meta-Ebene. Nimmt Nutzeranfragen entgegen und entscheidet, welcher WorkerAgent die Aufgabe ausführen soll. Verwendet dazu einen **AgentManager** (führt Agenten-Verzeichnis) und einen **NNManager** (KI-Modell zur Agentenauswahl) [1]. Nach der Auswahl leitet der Supervisor die Aufgabe weiter und sammelt das Ergebnis ein.	`agents/supervisor_agent.py` (Logik des Supervisors) [1]; delegiert an `AgentManager` und `NNManager`.
AgentManager	Verwaltung aller bekannten WorkerAgents. Initialisiert Standard-Agents mit vorgegebenem Domain-Wissen und kann bei Bedarf neue Agents erstellen [9] [10]. Nutzt Embeddings, um aus einer Task-Beschreibung eine passende Domäne abzuleiten und wählt entsprechend den Agenten oder erzeugt einen neuen [11] [12].	`managers/agent_manager.py` (Registrierung und Erstellung von Agents) [9] [10].
NNManager (Agent-Auswahl-Modell)	Komponente zur automatischen Agentenauswahl. Implementiert heuristische und ML-basierte Kriterien, um für eine gegebene Aufgabe den am besten passenden Agenten zu finden [13] [14]. Nutzt Embedding-Vergleiche und Regelwerk (anfangs simple Schwellenwertlogik), protokolliert Entscheidungen mit MLflow.	`managers/nn_manager.py` (einfaches Auswahlmodell) [13]; künftig abgelöst/ergänzt durch `HybridMatcher` & `MetaLearner` (für komplexeres NN-Modell) [15].

Komponente	Rolle im System	Relevante Klassen/Module
Neurale Netze in Agents	Jede WorkerAgent-Instanz besitzt ein internes neuronales Netz (**AgentNN**), das aus der Aufgabenbeschreibung Merkmalsvektoren generiert und zur Optimierung der Agentenperformance dient [16] [17]. Dieses Netz kann lernend aktualisiert werden, um z.B. häufige Aufgaben besser zu bewältigen. Zudem existiert ein **Meta-Lernmodell** (MetaLearner) im Hybridsystem zur Verbesserung der globalen Agentenauswahl [18] [19].	`nn_models/agent_nn.py` (Definition des AgentNN-Modells) [16]; `managers/meta_learner.py` (Meta-Lernmodell zur Agentenbewertung) [20] [21]; `managers/hybrid_matcher.py` (Hybrid aus Embedding- und NN-Scoring) [22] [23].
Wissens- & Kommunikationsverwaltung	Unterstützungskomponenten für geteiltes Wissen und Agenten-Kommunikation. Z.B. **DomainKnowledgeManager** zum Verwalten von Wissensbasen über Domänen hinweg, sowie ein **AgentCommunicationHub** für Messaging zwischen Agents (zum Anfragen von anderen Domänen) [24] [25]. Diese erlauben, dass WorkerAgents Informationen von anderen Experten-Agents einholen, wenn eine Anfrage dies erfordert.	`agents/domain_knowledge.py`, `managers/domain_knowledge_manager.py` (Domain-Wissensverwaltung); `agents/agent_communication.py` (Nachrichtenhub zwischen Agents).
Logging & Monitoring	Querschnittssystem für Protokollierung und Auswertung. Aktionen und Leistungsmetriken der Agents werden ereignisgesteuert geloggt (u.a. in MLflow für Experimente) [26]. Systemmetriken (CPU, Speicher, Antwortzeiten) sollen über dedizierte Endpunkte abrufbar sein.	`utils/logging_util.py` (LoggerMixin für einheitliches Logging); `mlflow_integration/` Module für MLflow-Tracking; API-Endpunkt `/metrics` für Systemstatus [27].

Komponente	Rolle im System	Relevante Klassen/Module
Schnittstellen (API & CLI)	Zugriffsschicht für Nutzer und Entwickler. Eine REST-API (FastAPI) erlaubt das Einstellen von Tasks, Abfragen von Ergebnissen und Management von Agents [28] [29]. Zusätzlich existiert ein CLI-Toolset (und experimentelles Frontend), um das System lokal zu steuern.	`api/endpoints.py` (API-Routen für Tasks/Agents/A/B-Tests) [30]; `cli/` Verzeichnis (Kommandozeilenbefehle); `frontend/` (Web-Oberfläche in React).

Architektur-Bewertung: Die modulare Struktur legt eine solide Basis, indem Kernfunktionen getrennt sind (Agentenlogik, Modell-Anbindung, Wissensspeicher). Bereits umgesetzt sind: - **LLM-Anbindung mit Fallback:** Die Klasse `BaseLLM` wählt je nach Konfiguration automatisch die OpenAI-API oder ein lokales Modell (z.B. Llama 2) zur Textgenerierung [4] [5]. API-Schlüssel werden über Umgebungsvariablen verwaltet, was Sicherheitsstandards erfüllt [31]. Für Embeddings wird analog OpenAI oder HuggingFace genutzt [32]. Dieses zweigleisige Setup ermöglicht Betrieb sowohl mit Cloud-Modellen als auch vollständig offline. - **Vektorstore & Wissen:** Die Wissensbasis jedes Agents wird in einer Chroma-Datenbank gehalten [6]. Neue Dokumente werden in Vektorform abgelegt und können über semantische Ähnlichkeit abgefragt werden [33] [34]. Diese Entkopplung von statischem Wissen (Dokumentspeicher) und logischer Agentenverarbeitung erleichtert Erweiterungen, z.B. Einspielen neuer Dokumente oder Austausch der Vektordatenbank. - **Agentenauswahl und -erstellung:** Aktuell wählt der Supervisor den Agenten für eine Aufgabe über den NNManager, der eine gewichtete Kombination aus Embedding-Ähnlichkeit und vordefinierten Regeln nutzt [35] [36]. Ist kein passender Agent vorhanden (Score unter Schwellwert), erstellt der AgentManager einen neuen Agenten basierend auf der erkannten Domäne der Anfrage [37] [38]. Dadurch kann das System dynamisch wachsen und sich an neue Themen anpassen. Verbesserungsfähig ist hier die Logik der Agentenauswahl – einfache Heuristiken wurden bereits durch einen Hybridansatz mit erstem Meta-Lernen ersetzt [15], jedoch besteht Potenzial für ein umfassender trainiertes Entscheidungsmodell (siehe Punkt 3 unten). - **Neurale Netzwerke in Agents:** Jede WorkerAgent-Instanz lädt bei Initialisierung ein eigenes kleines neuronales Netz (AgentNN), das Eingabebeschreibungen in einen Feature-Vektor projiziert [16] [17]. Diese Features können zur feineren Agentenauswahl (via MetaLearner) und zur Erfolgsmessung genutzt werden. Der Agent kann sein NN-Modell persistieren und wieder laden [39], was Raum für kontinuierliches Training pro Agent bietet. Bisher fließen Erfolgsmetriken (Antwortzeit, ggf. Feedback) in die AgentNN-Auswertung ein [40] [41], aber ein automatisiertes Nachtrainieren zur Leistungsverbesserung findet noch nicht statt (in Planung). - **Kommunikation & Kooperation:** Durch den *AgentCommunicationHub* können WorkerAgents sich untereinander Nachrichten schicken [24] [25]. Dies ermöglicht z.B., dass ein Agent aus Domäne A einen Agenten aus Domäne B konsultiert, wenn eine Anfrage domänenübergreifendes Wissen erfordert. Dieses Feature erhöht die Problemlösungsfähigkeit des Systems erheblich. Aktuell erfolgt die Kommunikation asynchron und mittels einfacher Message-Queues im Speicher. In Zukunft könnte hier ein skalierbareres Messaging-System (z.B. über einen Broker) nötig werden, falls viele Agenten verteilt arbeiten. - **Logging & Monitoring:** Das System nutzt MLflow, um wichtige Ereignisse und Parameter zu protokollieren – z.B. die Auswahlentscheidung des Supervisors (gewählter Agent, Score) [42] oder Performance-Updates nach Task-Ausführung [43] [44]. Dadurch entstehen Aufzeichnungen, die für spätere Auswertungen und zum Training eines verbesserten Entscheidungsmodells verwendet werden können. Allerdings sind Fehlerbehandlung und Warnungs-Logging (z.B. wenn ein Agent nicht reagiert) noch rudimentär [45]. Ebenso fehlt noch eine automatische **Überwachung** (Monitoring) der Systemressourcen zur Laufzeit – entsprechende Metrik-

Endpunkte sind zwar angedacht [27], müssen aber mit realen Daten gefüllt und ggf. in ein Dashboard integriert werden.

Zusammenfassend ist die aktuelle Architektur funktional, aber noch monolithisch verbunden. Viele Komponenten (LLM-Zugriff, Vektorstore, Agentenverwaltung, NN-Modelle) sind vorhanden, jedoch teils eng verflochten im selben Prozess. Die Basisfunktionen – vom Nutzer-Query bis zur Agentenantwort – sind als „Durchstich" implementiert [46]. Nun gilt es, diese Module besser zu **vereinheitlichen**, klarere Schnittstellen zu definieren und das System sowohl **robuster** (für den produktiven Einsatz) als auch **erweiterbarer** (für Forschungsexperimente) zu machen.

2. Vereinheitlichung der Module mit *ModelContextProtocol* (MCP)

Um die bestehenden Komponenten zu einem konsistenten Framework zu vereinen, wird eine übergeordnete Architektur namens **ModelContextProtocol (MCP)** eingeführt. Die Kernidee von MCP ist, **alle Module über ein gemeinsames Kontext-Protokoll miteinander kommunizieren zu lassen** und so eine lose Kopplung zu erreichen. Praktisch soll dies in einer *Modularen Control Plane* resultieren, in der einzelne Dienste getrennt, aber koordiniert arbeiten [47] [48]. Folgende Schritte und Design-Entscheidungen stehen dabei im Vordergrund:

- **Microservice-Aufteilung:** Die bisher im Monolith vereinten Funktionen werden als eigenständige Services aufgesetzt. Geplant sind u.a. ein **Task Dispatcher** (koordiniert eingehende Aufgaben), ein **Agent Registry** (verwaltet verfügbare Worker-Agents), ein **Session Manager** (hält Konversationskontext vor), ein **Vector Store Service** (bietet Vektor-Suche als separaten Dienst), ein **LLM Gateway** (vereinheitlichter Zugang zu LLM-Anbietern) sowie dedizierte **Worker Services** für jede Agenten-Domäne [49] [50]. Diese Dienste kommunizieren über definierte REST-APIs oder RPC-Aufrufe miteinander. Durch diese Entflechtung kann jeder Teil einzeln skaliert, neu gestartet oder weiterentwickelt werden, ohne andere zu beeinflussen.

- **ModelContextProtocol Definition:** Zentral ist ein standardisiertes **Kontext-Objekt** (bzw. Datenstruktur), das alle relevanten Informationen einer Task durch das System trägt. Dieses *ModelContext* könnte z.B. folgendes enthalten: *Task-Beschreibung*, *Optionale Domänenangabe*, *Sitzungs-ID oder Nutzerkontext*, *ggf. Eingabedaten/Dateireferenzen*, *Status und Ergebnisse*. Jeder Service versteht und nutzt dieses Kontextobjekt. **Beispiel:** Der API-Gateway erzeugt aus einer Nutzeranfrage ein `TaskContext`-Objekt und übergibt es an den Dispatcher. Der Dispatcher ergänzt Routing-Informationen (welcher Agent ausgewählt wurde) und leitet es an den entsprechenden Worker weiter. Der Worker fügt das Resultat ins Kontextobjekt ein, und schließlich sendet der Dispatcher den ausgefüllten Kontext zurück an den Gateway, der dem Nutzer antwortet [51] [52]. Dieses Protokoll stellt sicher, dass alle Module einheitlich mit Tasks und deren Meta-Daten umgehen – ob es sich um einen lokalen Aufruf oder eine externe API handelt.

- **Zentraler Task-Dispatcher & Registry:** Im jetzigen Code übernimmt der SupervisorAgent intern das Routing. Unter MCP wird diese Funktion vom **Task-Dispatcher Service** übernommen. Er enthält die Logik, einen geeigneten Worker-Service für eine Anfrage zu finden. Dazu fragt er den **Agent Registry Service** nach verfügbaren Agents und ihren Fähigkeiten (dieser Registry-Service ersetzt den in-memory AgentManager) [50]. Die Registrierung von Agents kann dynamisch erfolgen – beim Start meldet sich jeder Worker-Agent-Service beim Registry an (z.B. via `/register` Endpoint) [53]. Der Dispatcher nutzt dann das im Registry hinterlegte Wissen (Domäne, Gesundheitsstatus, evtl. Performancewerte) sowie das NN-Auswahlmodell, um den Ziel-Agenten zu bestimmen. Die Verwendung eines Registry-Moduls gewährleistet

Erweiterbarkeit: künftige Agents (ggf. auf anderen Maschinen oder als neue Domänen) können leicht ins System integriert werden, ohne den Dispatcher-Code ändern zu müssen.

- **Entkopplung der LLM- und Vectorstore-Funktionen:** Anstatt dass jeder WorkerAgent direkt Bibliotheken für LLM-Aufrufe oder Datenbanksuche einbindet, werden **LLM Gateway** und **Vector Store** als Services gekapselt. Zum Beispiel würde ein WorkerAgent-Service für eine Wissensfrage den Vector-Store-Service über dessen API fragen: „Suche ähnliche Dokumente zu X in meiner Collection“ anstatt selbst die Chroma-Library aufzurufen. Ebenso schickt er Prompt-Anfragen an den LLM-Gateway (der die Anfrage dann an OpenAI, HuggingFace oder andere weiterleitet). Diese Trennung erhöht die Wiederverwendbarkeit – verschiedene Agents können denselben Vector-Store-Service benutzen – und vereinfacht das Austauschen von Implementierungen (z.B. Wechsel der Vektordatenbank von Chroma zu Weaviate in Zukunft, ohne die Agentenlogik zu ändern). Auch sicherheitsrelevante Aspekte (z.B. API-Schlüssel für OpenAI) sind so zentral im Gateway verwaltet, nicht verstreut im Code der Agents.

- **Gemeinsame Schnittstellen und Protokolle:** Alle Services werden schlanke REST/Socket-Schnittstellen mit klar definierten Endpunkten bereitstellen (z.B. `POST /tasks` am Dispatcher, `GET /status` am Registry, `POST /vector_search` am Vector Store etc.). Diese Endpunkte orientieren sich an dem ModelContextProtocol-Format. Durch ausführliche API-Dokumentation (im Repo existiert bereits ein API-Referenzentwurf [30] [54]) wird gewährleistet, dass Entwickler und auch automatisierte Tests die Services konsistent ansprechen können. Wichtig ist, dass Transaktionen über Service-Grenzen hinweg robust gestaltet werden – z.B. sollten Tasks im Dispatcher in eine Warteschlange gestellt werden können, falls Worker kurzzeitig nicht erreichbar sind, und Antworten asynchron zurückgeliefert werden (etwa via Polling oder WebSockets). Diese Mechanismen erhöhen die Fehlertoleranz im Vergleich zur bisherigen synchronen In-Prozess-Aufrufkette.

- **Phaseweiser Übergang:** Die Migration zur MCP-Architektur kann schrittweise erfolgen. In **Phase 1** könnte man die Module als getrennte Python-Threads/Prozesse innerhalb eines Containers laufen lassen (simulierte Microservices), um Overhead zu minimieren, aber schon klare Modulgrenzen durch APIs zu ziehen. In **Phase 2** können diese Komponenten auf tatsächliche Services (z.B. Docker-Container mit FastAPI-Anwendungen) verteilt werden. Wichtig ist, während der Umstellung sämtliche vorhandene Funktionalität beizubehalten und mit Integrationstests zu überwachen, ob eine Anfrage weiterhin von Anfang bis Ende korrekt bearbeitet wird. Das *ModelContextProtocol* dient dabei als verbindendes Element – Tests können prüfen, ob ein TaskContext vollständig durch alle Stationen läuft und am Ende die erwarteten Felder (Ergebnis, Metriken etc.) enthält.

Durch die Vereinheitlichung via MCP wird das Framework insgesamt **wartbarer und skalierbarer**. Neue Module lassen sich als weitere Services einhängen (z.B. ein *Knowledge Graph Service* als zukünftige Erweiterung), ohne den Kern zu verändern. Fehler in einem Teil (etwa LLM-Ausfall) können isoliert behandelt werden, während andere Teile weiterlaufen. Zudem erleichtert die klare Trennung auch das **experimentelle Austauschen einzelner Komponenten** für Forschungszwecke – z.B. könnte man einen alternativen Vector-Store-Service ausprobieren, indem man nur die Service-URL austauscht, ohne die restlichen Agents anzufassen.

3. Integration von Agenten und Neuronalen Netzen vertiefen

Eine nahtlose Verzahnung der Agentenlogik mit lernfähigen neuronalen Netzen (NN) ist entscheidend, um das System intelligenter und anpassungsfähiger zu machen. Der Entwicklungsplan sieht vor,

Agenten und NNs wechselseitig zu integrieren, sodass *Agenten die Verwendung von NN-Modellen steuern* und umgekehrt *NN-Modelle die Agentenleistung evaluieren und verbessern*. Konkret werden folgende Ansätze vorgeschlagen:

- **Meta-Lernmodell für Agentenauswahl ausbauen:** Aktuell entscheidet der Supervisor auf Basis eines einfachen Scoring (Embeddings + Regelgewichtung) [35] [36], teils schon ergänzt durch einen MetaLearner-Ansatz im Hintergrund. Zukünftig soll ein vollwertiges **Meta-Neuronales Netz** die Agentenauswahl übernehmen. Dieses Modell erhält als Eingabe z.B. den Embedding-Vektor der Nutzeranfrage und charakteristische Feature-Vektoren aller verfügbaren Agents (die von deren AgentNNs stammen) und sagt voraus, welcher Agent den höchsten Erfolg verspricht [55] [56]. Das Meta-Modell (z.B. in `MetaLearner`) wird mit historischen Daten trainiert: Aus vergangenen Tasks kennt es, welcher Agent gewählt wurde und ob die Aufgabe erfolgreich gelöst wurde – so kann es lernen, Muster zu erkennen (Stichwort *Meta-Learning für Agent-Auswahl* [57] [58]). Die Integration dieses gelernten Modells bedeutet, dass der **NNManager** refaktoriert wird: statt statischer Gewichtung nutzt er die Vorhersage des MetaLearners als Hauptkriterium. In der Praxis würde der SupervisorAgent dann `NNManager.predict_best_agent(...)` aufrufen, welcher intern den MetaLearner verwendet (mit Fallback auf Embedding-Only, falls das Modell unsicher ist). Damit werden Agentenentscheidungen **dynamischer und datengesteuerter** – das System passt sich im Lauf der Zeit an, welcher Agent für welche Art Aufgabe geeignet ist.

- **Feedback-Schleifen und kontinuierliches Lernen:** Damit das Meta-Entscheidungsmodell und die AgentNNs besser werden, ist ein automatisierter Feedback-Mechanismus einzurichten. Jeder Task-Durchlauf liefert einen Datenpunkt: welcher Agent wurde gewählt, was war das Ergebnis, wie lange dauerte es, gab es Fehler oder Nutzerfeedback? Diese **TaskMetrics** fließen bereits jetzt in Logs ein [59], sollen aber künftig genutzt werden, um die Modelle nachzujustieren. Geplant ist ein Hintergrundprozess oder eigener Service **Training Manager**, der periodisch die gesammelten Erfolgsdaten nimmt und **Online-Learning** betreibt [60]: z.B. alle X Aufgaben wird der MetaLearner mit neuen Daten weitertrainiert, und ebenso kann jeder WorkerAgent sein AgentNN mit den zuletzt bearbeiteten Fällen feinjustieren (Stichwort *Reinforcement Learning* für Agenten). Dieser Vorgang kann durch MLflow-Tracking begleitet werden, um zu überwachen, ob sich Metriken wie Erfolgsquote oder Antwortzeit verbessern. Wichtig ist, konservative Strategien zu verfolgen (z.B. nur kleine Lernraten beim Online-Lernen, A/B-Tests neuer Modellparameter), damit ein Lernen am produktiven System zu keinen Leistungseinbrüchen führt. Langfristig lernt so der Supervisor, immer treffender den optimalen Agenten zu picken, während die Agents selbst ihre Fähigkeiten ausbauen.

- **Agent steuert interne Modelle:** Die WorkerAgents sollen mehr Kontrolle darüber erhalten, *welche spezialisierten Modelle oder Tools* sie für eine Teilaufgabe einsetzen. Aktuell hat jeder Agent genau ein festes LLM und ein AgentNN. Perspektivisch kann ein WorkerAgent aber ein **Ganzes Set an Modulen** beinhalten (siehe *Model Pipeline* in der erweiterten Architekturplanung [61]): z.B. ein Finanz-Agent könnte neben dem generativen LLM noch ein **OCR-Neurales Netz** für das Auslesen von Tabellen, ein **Regressionsmodell** für Vorhersagen oder ein **Sentiment-Analysis-Modell** integriert haben. Der Agent bräuchte dann ein *Model Selection Module*, das je nach Task-Anforderung entscheidet, welches interne Modell genutzt wird [62] [63]. Diese Entscheidungslogik kann regelbasiert beginnen (z.B. wenn Eingabe ein Bild/PDF, dann OCR-Modell; wenn Sentiment-Frage, dann Sentiment-NN), lässt sich aber ebenfalls lernen (der Agent könnte aus vergangenen ähnlichen Aufgaben lernen, welches Modell zum besten Ergebnis führte). Dadurch **steuert der Agent sein NN-Arsenal selbstständig**. Entwickeln lässt sich dies modular: Man definiert pro Agententyp konfigurierbare *Capabilities* und hinterlegt, welche Modelle verfügbar sind. Die Agenten-API kann dann Aufträge intern routen, z.B.

`agent.execute_task(task)` prüft die Task-Beschreibung auf Schlüsselindikatoren und ruft dann entweder `self.llm.generate(...)` oder `self.domain_ocr_model.infer(...)` etc. auf. Damit einher geht die Notwendigkeit, das **Agent-Interface zu vereinheitlichen**, sodass von außen (Supervisor oder Dispatcher) nicht sichtbar ist, welches interne Modell gerade arbeitete – es zählt nur die finale Antwort im bekannten Kontextformat.

- **NN-gestützte Leistungsevaluierung der Agenten:** Neben der Auswahl des passenden Agents pro Task kann KI auch bei der **Bewertung der Agentenantworten** helfen. Geplant ist ein **Evaluation Manager** (ggf. als separater Service), der Ergebnisse automatisch analysiert. Dieser könnte z.B. ein LLM nutzen, um Antworten nach bestimmten Kriterien zu bewerten (Relevanz, Korrektheit) oder statistische Modelle, die Erfolgschancen vorhersagen. Auch könnte das Meta-Modell aus der Agentenauswahl im Nachhinein die Entscheidung bewerten: War die Wahl korrekt? Falls nicht, wird ein Fehlgriff registriert. Solche Daten fließen zurück in die Agenten-Metriken. Auf diese Weise **evaluieren NNs die Agenten**, indem sie z.B. Qualitäts-Scores vergeben. In der Umsetzung heißt das, bei jedem Task-Ende wird ein Evaluation-Workflow angestoßen – der WorkerAgent meldet Erfolg/Misserfolg und ggf. User-Feedback zurück an den Supervisor/Dispatcher, der gibt diese Info an den Evaluation Manager, der einen Score berechnet (0–1) und diesen Score dann dem MetaLearner-Training und dem AgentNN-Feedback zuführt. Dieses Konzept ist eng verwandt mit Reinforcement Learning: der Evaluation Manager liefert letztlich den Reward-Signal für die Agentenauswahl und -ausführung.

- **Verbesserte Zusammenarbeit und Wissensaustausch:** Die Inter-Agent-Kommunikation sollte weiter ausgebaut werden, damit Agenten komplexe Anfragen kooperativ lösen können. Hier kommen neuronale Modelle ebenfalls ins Spiel: Z.B. könnte ein spezieller **Routing-Agent** (selbst ein kleiner LLM) die Anfrage zunächst in Teilaufgaben zerlegen und an mehrere Worker verteilen – ein Ansatz aus der *Chain-of-Thought* Idee (Iteration 5 im ursprünglichen Plan sieht so etwas vor: komplexere Denkketten und LangChain-Tool-Integration [64]). Ein trainiertes Modell könnte lernen, welche Unteraufgaben typischerweise erforderlich sind und welche Agenten gut zusammenarbeiten. Außerdem sollen Agents künftig **Wissen teilen** können: Ein Knowledge-Graph oder gemeinsamer Vektorstore könnten als globale Ressource dienen, auf die alle zugreifen. Neuronale Netze könnten genutzt werden, um Muster in diesem geteilten Wissen zu erkennen (Stichwort *cross-agent knowledge sharing*). Diese Erweiterungen sind eher forschungsorientiert, sollten aber durch die modulare MCP-Architektur leicht anzudocken sein (etwa ein *Knowledge Manager Service*, der von allen Agents konsultiert wird).

Zusammengefasst zielt dieser Integrationsschritt darauf, das System **lernender und adaptiver** zu machen. Wo bisher fixe Zuordnungen oder einfache Heuristiken verwendet wurden, treten nun trainierbare Modelle an die Stelle: Das **Supervisor-Modell** lernt die beste Agentenwahl, **Agents** selbst lernen aus Erfahrungen (eigene und globale) und **Modelle bewerten Modelle** (Meta-Evaluation). Wichtig ist, diesen Prozess mit ausreichender **Transparenz** und **Kontrollmöglichkeiten** zu gestalten – z.B. via Dashboard für Agenten-Performance, justierbaren Parametern (Schwellenwerte, Lernraten) und ausführlichem Logging jeder automatischen Entscheidung, damit Entwickler das Verhalten nachvollziehen und bei Bedarf korrigieren können (insbesondere relevant, da das System produktiv eingesetzt wird, aber dennoch Experimente erlaubt sein sollen).

4. Entwickler-SDK für lokale und externe Modelle

Ein zentrales Ziel ist es, ein **SDK (Software Development Kit)** bereitzustellen, das Entwickler nutzen können, um beliebige Sprach- und KI-Modelle – ob lokal installiert oder via API von Drittanbietern – in das Agent-NN-Framework einzubinden. Dieses SDK abstrahiert die Details der jeweiligen Modell-

Anbindung und stellt **einheitliche Schnittstellen** zur Verfügung. Folgende Aspekte umfasst das SDK-Design:

- **Abstraktion der Modellprovider:** Es wird eine gemeinsame **Interface-Klasse** definiert (z.B. `LLMProvider`), die grundlegende Methoden bereitstellt, etwa `generate_text(prompt: str) -> str` für Textgenerierung, `embed_text(text: str) -> Vector` für Embeddings und ggf. `model_info()` für Metadaten. Für jeden konkreten Anbieter oder Modelltyp implementiert das SDK eine Unterklasse dieses Interface. Beispielsweise: `OpenAIProvider` ruft die OpenAI-API mit den entsprechenden Keys/Modellnamen auf, `AnthropicProvider` kümmert sich um Aufrufe an die Anthropic-API, `HuggingFaceProvider` nutzt entweder die HuggingFace Python-Bibliothek oder deren Hosted Inference API, `LocalModelProvider` deckt lokale Modelle wie GPT4All, Llama.cpp & Co ab. Intern hat das Projekt bereits einen Ansatz in diese Richtung: die Klasse `BaseLLM` entscheidet anhand der Konfiguration, ob ein OpenAI-Modell oder ein Llama-2-Modell (über LlamaCpp) genutzt wird [4] [5]. Dieses Konzept wird nun verallgemeinert: nicht nur OpenAI oder Llama, sondern beliebige *Pluggable Backends* können so konfiguriert werden. Entwickler sollen z.B. via einer Konfigurationsdatei oder Umgebungsvariable angeben können: `LLM_BACKEND=openrouter` oder `LLM_BACKEND=anthropic`, und das SDK lädt dann automatisch den passenden Provider mit den nötigen API-Keys.

- **Unterstützung von** LiteLLM **und** OpenRouter**:** Anstatt jeden Anbieter einzeln zu behandeln, kann das SDK auch Integrationen zu Meta-APIs bieten. **OpenRouter** etwa abstrahiert verschiedene LLM-APIs unter einer einheitlichen Schnittstelle – das SDK könnte einen `OpenRouterProvider` bereitstellen, bei dem Entwickler lediglich den OpenRouter-Key nutzen, um dann je nach Wunsch OpenAI, Anthropic etc. über diesen Router anzusteuern. **LiteLLM** wiederum könnte eine Bibliothek sein (der genaue Kontext ist unbekannt) – vermutlich ein leichtgewichtiger Client, um mit verschiedenen Modellen zu interagieren. Das SDK sollte solche Tools nutzen, um den Implementierungsaufwand gering zu halten. Für Developer Experience bedeutet das: Mit wenigen Zeilen können sie z.B. sagen *„Nutze Modell X von Anbieter Y“* und das SDK kümmert sich um Authentifizierung, Anfrageformat und Antwortparsing. Ein einheitliches Fehlerhandling (z.B. Timeout, Rate Limits) wird ebenfalls im SDK zentralisiert, damit nicht jede Komponente das neu behandeln muss.

- **Lokale Modelle und HuggingFace-Support:** Da produktiver Einsatz oft aus Datenschutz- oder Verfügbarkeitsgründen lokale Modelle erfordert, wird das SDK einen **LocalModelManager** enthalten. Dieser kann gängige vortrainierte Modelle aus der HuggingFace-Community laden (sofern ausreichend Hardware vorhanden ist) – etwa via `transformers` Pipeline – oder auf spezialisierte lokale Engines zugreifen (wie die bereits genutzte Llama.cpp in `LocalLLM` [65]). Entwickler sollen über das SDK entweder bereits konfigurierte lokale Modelle starten können oder eigene Modelle registrieren. Ein möglicher Ansatz ist, eine **Modell-Registrierung** in der Konfiguration bereitzustellen (ähnlich einer Model Registry in MLflow, aber hier zur Laufzeit): z.B. eine YAML-Datei, die Pfade zu lokalen Modelldateien oder Namen von HuggingFace-Modellen enthält, die das System laden darf. Das SDK könnte beim Start alle verfügbaren Modelle initialisieren oder dynamisch laden, sobald sie das erste Mal angefordert werden. Wichtig hierbei ist die **Performance-Optimierung**: Lokale Modelle sollen nach Möglichkeit auf geeigneter Hardware (GPU) laufen und, falls sie groß sind, als singleton Instanzen im Memory bleiben, damit nicht pro Anfrage neu geladen wird. Das SDK kann hierzu einen *Model Pool* verwalten.

- **Einbindung weiterer KI-Dienste:** Neben reinen LLMs sollte das SDK erweiterbar sein für andere KI-Services. In der Praxis gibt es APIs für z.B. **Speech-to-Text**, **Vision OCR**, **Translation** etc. Ein

flexibles SDK lässt Entwickler auch diese Dienste einhängen. Dafür kann man das gleiche Muster nutzen: eine abstrakte Klasse `AIServiceProvider` mit Implementationen wie `SpeechToTextProvider` (für z.B. AssemblyAI, Whisper API), `VisionOCRProvider` etc. Da das Agent-NN-Framework längerfristig vielseitige Agenten (auch mit visuellen oder auditiven Inputs) unterstützen soll, ist diese Erweiterbarkeit essenziell. Somit würde das SDK zum zentralen **Hub für KI-Modelle** jeglicher Art, und die Agents rufen lediglich die abstrahierten Methoden auf, ohne sich um die Details der jeweiligen API zu kümmern.

- **Developer Experience und Dokumentation:** Ein gutes SDK zeichnet sich durch einfache Benutzbarkeit und klare Dokumentation aus. Geplant ist, dem Repository einen **ausführlichen Leitfaden** hinzuzufügen (z.B. `docs/developer_sdk.md`), der anhand von Beispielen zeigt, wie man:

- einen neuen Modell-Provider implementiert (Schritt-für-Schritt-Anleitung, z.B. um einen neuen API-Anbieter einzubinden),
- zwischen vorhandenen Modellen wechselt (Konfigurationsbeispiele, etwa wie man von OpenAI zu eigenem Modell umstellt),
- lokale Modelle optimiert (Tipps zur Hardware-Nutzung, Batch-Processing etc.),

- und wie Agents oder externe Anwendungen über die API mit dem SDK interagieren. Außerdem wird das SDK selbst mit **Docstrings und Typannotationen** versehen, sodass ein `help()` oder IDE-IntelliSense Entwicklern schnell die Nutzung zeigt. Für gängige Anwendungsfälle könnten **Code-Beispiele** bereitgestellt werden, z.B. ein minimaler Python-Code, der das Agent-NN-System mit einem bestimmten Modell aufsetzt und eine Testanfrage stellt.

- **Unterstützung für Testing und Simulation:** Für Forschungsexperimente wollen Entwickler möglicherweise neue Modellvarianten ausprobieren, ohne gleich das ganze System aufzusetzen. Daher sollte das SDK es erlauben, Komponenten auch **isoliert zu testen**. Z.B. könnte man direkt den LLM-Gateway mit einem Dummy-Provider laufen lassen, um zu sehen wie Prompts formatiert werden, oder den Agent-Auswahlmechanismus mit einem Mock von Agent-Feature-Vektoren füttern. Hierzu wird das SDK mit einigen **Mock- oder Dummy-Implementierungen** kommen (etwa ein `EchoLLMProvider` der nur das Prompt zurückgibt, oder ein `RandomAgentSelector` als Platzhalter). Diese erleichtern Unit-Tests des Gesamtsystems und ermöglichen es Forschern, Teile der Pipeline zu ersetzen, um hypothetische Szenarien durchzuspielen.

Insgesamt fördert das Entwickler-SDK die **Wiederverwendbarkeit** der Plattform: Das Framework wird für verschiedenste Modelle offen, ohne dass tiefe Code-Eingriffe nötig sind. Gleichzeitig erhöht es die **Stabilität** im Produktivbetrieb, da alle Modelle über geprüfte Schnittstellen angebunden sind – das verringert Fehler durch inkonsistente Nutzung der unterschiedlichen APIs.

5. Implementierungsfahrplan: Prioritäten, Reihenfolge und Architekturänderungen

Angesichts der Analyse und Ziele werden im Folgenden die Entwicklungsschritte priorisiert und in eine sinnvolle Reihenfolge gebracht. Dabei fließen Überlegungen zur **Robustheit für den produktiven Einsatz** ebenso ein wie die **Offenheit für Erweiterungen** zu Forschungszwecken. Ein mehrphasiger Plan könnte wie folgt aussehen:

Phase	Schwerpunkte und Maßnahmen	Ergebnis (Zielzust
Phase 1: Grundlagen stabilisieren und Architektur vereinheitlichen (MCP)	*Modulare Neuordnung*: Aufteilung der Monolith-Architektur in definierte Services (Task-Dispatcher, Agent Registry, Session, VectorStore, LLM-Gateway, Worker-Agents) gemäß ModelContextProtocol [49] . *Kontext-Protokoll*: Einführung eines TaskContext-Objekts, das in allen Modulen verwendet wird, inkl. Anpassung der API-Endpunkte an dieses Schema. *Kommunikation*: Einrichtung der internen APIs/Endpoints zwischen den Services (z.B. REST-Aufrufe vom Dispatcher zum Worker). *Baseline-Tests*: Sicherstellen, dass eine Nutzeranfrage den kompletten Weg über neue Services erfolgreich durchläuft (Ende-zu-Ende-Test im lokalen Deployment).	**Modularisiertes System**: Klar getre Services, die über definierte Schnittstellen interagieren. Die Kernfunktionalität (Aufgabenrouting bearbeitung) läuft weiterhin vollständ durch, aber nun au einer Architektur, d skalier- und wartba Erste **Gesundheits Checks** (/health Endpoints) und Logging in jedem Service geben Transparenz über Systemzustand.

Phase	Schwerpunkte und Maßnahmen	Ergebnis (Zielzust
Phase 2: Vertiefte Agent-NN-Integration und Lernmechanismen	*MetaLearner aktivieren*: Den bestehenden Meta-Lernalgorithmus für die Agentenauswahl produktiv nehmen – d.h. SupervisorAgent nutzt nun standardmäßig den trainierbaren Pfad (HybridMatcher/MetaLearner) [15]. Trainingsdaten aus der Vergangenheit werden aufbereitet, um einen initialen Model-Stand zu haben. *Feedback-Loop*: Implementierung der automatischen Rückkopplung: Nach jeder Task speichert der Dispatcher/Supervisor einen Datensatz (Kontext, gewählter Agent, Erfolg) und ein Hintergrund-Thread oder Service aktualisiert in regelmäßigen Abständen die Modelle (AgentNNs, MetaLearner) auf Basis dieser Daten. *Agent-internes Modellrouting*: Für ausgewählte Domänen-Agenten (z.B. *SoftwareDev* vs. *Marketing*) Prototyping von je einem zusätzlichen Spezialmodell und Logik, die entscheidet, wann es genutzt wird. Z.B. der *SoftwareDev-Agent* könnte ein statisches Code-Analyse-Modul bekommen und bei Programmierfragen zunächst dieses nutzen. Das dient als Proof-of-Concept für agenteninterne Modellauswahl. *Evaluationsmetriken*: Entwicklung erster automatisierter Erfolgsmessungen – etwa Vergleich der vom Agenten gegebenen Antwort mit einer erwarteten Lösung (wo verfügbar) oder simple Heuristiken (Antwort enthält bestimmte Stichworte = vermutlich relevant). Diese Metriken fließen als numerisches Feedback in die Update-Schleife ein (z.B. in Form eines `success_score`).	**Intelligenteres System**: Die Agentenauswahl erfolgt adaptiv und datengesteuert, w höherer Treffgenauigkeit fü Die Agenten zeige erste **Lernfähigke** da ihre internen Modelle und Auswahlkriterien angepasst werden (kontinuierliches Lernen). Das Syste loggt umfassende Metriken (Erfolgsra Antwortzeiten pro Agent etc.), wodur ein **Performance-Dashboard** entste kann. Risiken wie Fehlentscheidunge des Meta-Modells werden durch Monitoring abgefangen (im Zw kann auf das einfachere Auswahlverfahren zurückgeschaltet werden).

Phase	Schwerpunkte und Maßnahmen	Ergebnis (Zielzust
Phase 3: Entwickler-SDK und Multi-Provider-Unterstützung	*LLM SDK fertigstellen*: Implementierung der abstrahierten Provider-Klassen für OpenAI, Anthropic, HuggingFace und lokale Modelle. Integration von OpenRouter als optionaler Pfad. Sicherstellen, dass Konfiguration (Keys, Modellnamen) bequem über `config.llm_config` oder Env-Vars verwaltet wird. *Einbindung ins System*: Austausch der bisherigen direkten LLM-Aufrufe im Code durch das SDK. Z.B. der LLM-Gateway-Service nutzt intern nur noch das SDK: `LLMProviderFactory.get_provider(...).generate_text(prompt)`. Damit kann ein Wechsel des Backends ohne Codeänderung erfolgen. *Erweiterung Vectorstore-SDK*: Falls nicht schon in Phase 1 erfolgt, ähnliches Prinzip für den Vektor-Speicher – z.B. ob lokal Chroma oder ein Cloud-Vektorservice (Pinecone o.ä.) verwendet wird, sollte konfigurierbar sein. *Dokumentation & Beispiele*: Ausarbeitung der SDK-Dokumentation mit Anleitungen. Parallel Implementierung von Unit-Tests für jeden Provider (z.B. mit gemockten API-Antworten), um Zuverlässigkeit sicherzustellen. *Community-Feedback*: Falls möglich, Einbeziehung erster Nutzer, die das SDK ausprobieren, um API-Design und Fehlermeldungen zu verbessern (Developer Experience Feinschliff).	**Flexibles Entwicklungs-Too** Entwickler können Framework leicht verschiedenen KI-Modellen betreibe Das *Agent-NN* ist d sowohl in Cloud-Umgebungen (Op Anthropic) als auch isolierten On-Prem Umgebungen (nur lokale Modelle) einsetzbar. Durch klare Dokumentati und die Vereinheitlichung reduziert sich die Einarbeitungszeit neue Entwickler. N Modell-Anbieter können über das S mit minimalem Aufwand integrier werden, was die **Zukunftssicherhe** des Projekts erhöh

Phase	Schwerpunkte und Maßnahmen	Ergebnis (Zielzust
Phase 4: Produktreife, Optimierung und Erweiterbarkeit sichern	*Umfassende Tests*: Ausbau der Test-Suite – alle neuen Services erhalten Integrationstests (Simulation mehrerer Agents, Ausfall eines Services etc.). Fehlerfälle werden gezielt getestet (Netzwerkausfall zum LLM-API, ungültige Eingaben, Timeout eines Agenten). *Fehlerbehandlung*: Implementierung globaler Error-Handler in den Services, sodass bei Ausnahmen informative Antworten/Logs erzeugt werden statt unkontrollierter Abbrüche. *Performance-Tuning*: Identifikation von Engpässen (Profiling). Einführung von Caching für häufige Vektor-Queries oder LLM-Antworten, optional Nutzung von **GPU-Beschleunigung** für lokale Modelle (Integration von CUDA falls verfügbar). Load-Tests mit simulierten Anfragen, um die Skalierbarkeit der Microservices (z.B. mehrere gleichzeitige Worker) zu evaluieren. Ergebnisse fließen in Optimierungen (Threading, Batch-Processing, Datenbank-Indizes). *CI/CD Pipeline*: Einrichtung von Continuous Integration (z.B. GitHub Actions), die bei jedem Commit Tests und Linter ausführt [66] . Optional Continuous Deployment für eine Staging-Umgebung, um neue Versionen automatisiert bereitzustellen. *Dokumentation & Guides*: Finalisierung der Nutzer-Dokumentation (README, Getting Started), Architekturbeschreibung und eines *Contributing Guide* für externe Beitragende. Speziell die Erweiterungspunkte (Plugins, neue Agents, neue Modelle) werden dokumentiert, damit die Forschungscommunity eigene Ideen einbringen kann. *Sicherheits- und Produktiv-Features*: Implementierung von Authentifizierung/Autorisierung für die API (Token-basierter Schutz) [67] , Request-Limits [68] und Audit-Logging für kritische Aktionen. Diese Dinge sind unerlässlich für einen stabilen produktiven Betrieb. Gleichzeitig können sie bei Bedarf für Forschungstestläufe abgeschaltet werden, um keine Entwicklung zu behindern. *Optional: Deployment-Templates*: Bereitstellen von Docker-Compose oder Helm-Charts, um das gesamte System (inkl. aller Microservices) leicht auf Kubernetes oder VMs auszurollen – dies erleichtert sowohl Produktivdeployments als auch verteilte Experimente.	**Robustes, wartba Release**: Das Framework erreich Produktionsreife – ist stabil, skalierba und gut abgesiche Durch die Priorisie von Wartbarkeit (T CI, Doku) ist es langfristig betreibl Gleichzeitig ist es **für Forschung**: Modulare Architek Plugins und ausführliche Doku ermöglichen es, ne Ideen auszuprobie seien es eigene Agenten, spezielle Modelle oder alternative Auswahlverfahren Dank der isolierten Services kann ein Forschungs-Team an einem *MetaLea 2.0* arbeiten, ohne ganze System zu gefährden, und es Erfolg einfach austauschen. Insgesamt entsteh ein **flexibles KI-Agenten-Framew** das im täglichen Produkteinsatz Mehrwert liefert u zugleich als Experimentierplatt dient.

Prioritäten & Architekturänderungen: Die obige Roadmap priorisiert zuerst die **Architektur** (Phase 1), weil eine saubere Trennung der Module die Basis für alle weiteren Verbesserungen legt. Damit einher geht bereits eine Verbesserung der Wartbarkeit (klare Verantwortlichkeiten pro Service) und eine Vorbereitung für Skalierung. In Phase 2 folgen die **KI-bezogenen Verbesserungen** – hier steht die *Intelligenz* des Systems im Vordergrund (Lernmechanismen, bessere Entscheidungen). Phase 3 adressiert die **Flexibilität nach außen** durch das SDK, was wichtig ist, um das System in verschiedenen Umgebungen einzusetzen und für Entwickler attraktiv zu machen. Phase 4 bündelt alle restlichen Aspekte, um das Projekt auf ein hohes Qualitätsniveau zu heben (Tests, Doku, Performance, Sicherheit).

Während dieser Entwicklung sind eventuell **Architekturänderungen** nötig, etwa das Austausch des In-Memory-Message-Hubs durch eine robustere Message Queue (z.B. RabbitMQ oder Redis pub/sub) wenn Agents über Prozessgrenzen hinweg kommunizieren. Auch könnte die Persistenz der Agenten (ihre Wissensdatenbanken, Modelle) neu gedacht werden – z.B. Ablage in einer zentralen DB oder Registry, damit neu gestartete Agent-Container ihren Zustand rekonstruieren können. Solche Änderungen sollen stets mit dem Ziel erfolgen, **Wiederverwendbarkeit** und **Klarkeit** zu erhöhen. Jede Schnittstelle, die eingeführt wird, sollte hinreichend generisch sein, damit sie auch zukünftigen Anforderungen standhält. Beispielsweise lohnt es sich, das ModelContext-Protokoll so zu gestalten, dass es leicht erweiterbar ist (etwa durch zusätzliche optionale Felder für neue Features, ohne die alten zu brechen).

Abschließend wird durch diesen Plan sichergestellt, dass *Agent-NN* sowohl den **praktischen Anforderungen** eines Produktivbetriebs (Stabilität, Effizienz, Sicherheit) gerecht wird, als auch die **Forschungsfreiräume** bietet, um neue KI-Methoden im Agentenverbund auszuprobieren. Durch die priorisierte Umsetzung von Modularität und sauberer Schnittstellen entsteht ein Framework, das **dauerhaft anpassbar** ist – ein Entwickler kann in einigen Jahren ein ganz neues Modell integrieren oder einen Agenten für eine neue Domäne hinzufügen, ohne das Rad neu zu erfinden. Dieser Entwicklungsfahrplan führt schrittweise zu einem voll funktionsfähigen Agenten-Framework, in dem LLMs, spezialisierte Neural Nets und Agenten-Logik zu einer **kohärenten, leistungsfähigen Plattform** verschmelzen.

Quellen: Die Analyse und Planung stützen sich auf die vorhandene Codebasis und Dokumentation des Projekts (siehe referenzierte Ausschnitte), insbesondere auf das Roadmap-Dokument [69] [15] und die Architekturentwürfe für das ModelContextProtocol/MCP [47] [49]. Diese wurden mit den vorgeschlagenen Erweiterungen verknüpft, um einen konkreten Umsetzungspfad zu formulieren. Die genannten Änderungen und Erweiterungen sind darauf ausgerichtet, die im Repository bereits erkennbaren Ziele – z.B. Embeddings-basierte Agentenauswahl, kontinuierliches Lernen, Domänenspezialisierung und Skalierung – zu erreichen und das System in eine produktionsreife wie auch forschungsfreundliche Form zu bringen. [1] [4]

---

[1] [37] [38] supervisor_agent.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/8fed0725188c00b66c66f86a6f4426f6a2ca4ded/agents/supervisor_agent.py

[2] [3] [24] [25] [39] worker_agent.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/8fed0725188c00b66c66f86a6f4426f6a2ca4ded/agents/worker_agent.py

[4] [5] [65] base_llm.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/8fed0725188c00b66c66f86a6f4426f6a2ca4ded/llm_models/base_llm.py

[6] [15] [26] [31] [45] [46] [57] [58] [60] [61] [62] [63] [64] [66] [67] [69] Roadmap.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/8fed0725188c00b66c66f86a6f4426f6a2ca4ded/Roadmap.md

[7] [32] vector_store.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/8fed0725188c00b66c66f86a6f4426f6a2ca4ded/datastores/vector_store.py

[8] [33] [34] worker_agent_db.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/8fed0725188c00b66c66f86a6f4426f6a2ca4ded/datastores/worker_agent_db.py

9 10 11 12 agent_manager.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/8fed0725188c00b66c66f86a6f4426f6a2ca4ded/managers/agent_manager.py

13 14 35 36 42 43 44 59 nn_manager.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/8fed0725188c00b66c66f86a6f4426f6a2ca4ded/managers/nn_manager.py

16 17 40 41 agent_nn.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/8fed0725188c00b66c66f86a6f4426f6a2ca4ded/nn_models/agent_nn.py

18 19 22 23 hybrid_matcher.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/8fed0725188c00b66c66f86a6f4426f6a2ca4ded/managers/hybrid_matcher.py

20 21 55 56 meta_learner.py
https://github.com/EcoSphereNetwork/Agent-NN/blob/8fed0725188c00b66c66f86a6f4426f6a2ca4ded/managers/meta_learner.py

27 28 29 30 54 68 api_reference.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/8fed0725188c00b66c66f86a6f4426f6a2ca4ded/docs/api/api_reference.md

47 48 49 51 52 overview_mcp.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/8fed0725188c00b66c66f86a6f4426f6a2ca4ded/docs/architecture/overview_mcp.md

50 53 mcp_components.md
https://github.com/EcoSphereNetwork/Agent-NN/blob/8fed0725188c00b66c66f86a6f4426f6a2ca4ded/docs/architecture/mcp_components.md