MoE Sovereign

Ein selbstgehosteter Multi-Model-Orchestrator mit template-basiertem Expert-Routing für souveräne KI-Infrastruktur

Whitepaper — Version 2.7, Juli 2026

Philipp Horn

Herausgeber, Initiator und Ideenhalter

Benediktus-Kirchplatz 15
59387 Ascheberg
Deutschland

kontakt@philipp-horn.dev
github.com/h3rb3rn/moe-sovereign

Repository	github.com/h3rb3rn/moe-sovereign
Dokumentation	docs.moe-sovereign.org
Website	moe-sovereign.org
LinkedIn	linkedin.com/in/philipp-horn-dev

Inhaltsverzeichnis

Abbildungsverzeichnis

Zusammenfassung

Diese Arbeit stellt MoE Sovereign vor, ein vollständig selbstgehostetes Multi-Agenten-Orchestrierungssystem[a] für datenschutzkonforme, domänenspezialisierte KI-Infrastruktur. Das System ist ein konkreter Versuch, *digitale Souveränität* als technischen Zustand herzustellen, nicht bloß als politisches Schlagwort: jede Komponente läuft lokal, jede Abhängigkeit ist benannt, jeder Datenfluss ist dokumentiert.
Der Orchestrator ist in Python geschrieben und nutzt LangGraph [2] als Ausführungsmodell. Anfragen werden nicht an einen gelernten Router übergeben, sondern über *versionierte Expert-Templates* an zwei bis drei lokal betriebene Open-Weight-Modelle pro Expertenkategorie weitergeleitet. Eine vierschichtige Cache-Hierarchie (ChromaDB [3] semantisch, Valkey [4] Plan-/Graph-/Performance-Score-Ebenen) überspringt redundante LLM-Aufrufe frühzeitig. Ein auf Neo4j [5] gestützter Wissensgraph wird im laufenden Betrieb durch einen Graph-basierten Akkumulationsmechanismus mit neuen Entitäten und Relationen angereichert, wobei domänenspezifische Filter Cross-Kontamination zwischen medizinischen, juristischen und technischen Kontexten verhindern. Deterministische Operationen (Arithmetik, Datumsberechnung, Subnetz-Kalkulation, juristische Paragraphensuche) werden an einen Model-Context-Protocol-Server [6] mit einem AST-Whitelist-Evaluator ausgelagert und damit der Halluzinationsneigung probabilistischer Modelle entzogen.
Das System unterstützt gleichzeitig drei API-Protokolle: das OpenAI-Chat-Completions-Interface (`/v1/chat/completions`), die Anthropic Messages API (`/v1/messages`) und die OpenAI Responses API (`/v1/responses`) – Claude Code, Claude Desktop und jeder OpenAI-kompatible Client können es als Drop-in-Gateway nutzen, ohne Code-Änderungen vorzunehmen. Es benötigt im Minimalbetrieb keine ausgehenden Internetverbindungen und unterstützt durch seine Architektur Privacy-by-Design-Anforderungen gemäß Artikel 25 der DSGVO [7]. Dasselbe OCI-Image läuft rootless in einem unprivilegierten Proxmox-LXC, als Docker-Compose-Stack, als Podman-Quadlet-Unit und als Helm-Release auf Kubernetes oder OpenShift. Ein Overnight-Stabilitätsbenchmark (36 Szenarien, 3 Epochen, null Fehler) zeigt, dass ein selbst gehostetes Ensemble aus acht domänenspezialisierten 7–9B-Modellen auf Legacy-Tesla-M10-Hardware **6,11 / 10** auf dem internen MoE-Eval-Benchmark erreicht (Likert-Skala, task-spezifisch gewichtet; kein kontrollierter Vergleich mit externen Frontier-Benchmarks) – bei vollständiger Datensouveränität. Wir diskutieren Architektur-Entscheidungen, berichten offen über Fehlschläge und Lernkurven der jüngsten Refactoring-Runden (einheitliche OCI-Artefakt-Schicht, Security-Hardening, Mermaid-Rendering-Regression sowie einen Benchmark-Integritätsvorfall, bei dem irrtümlich nicht-souveräne Frontier-Modelle in einem GAIA-Evaluationslauf verwendet wurden – vollständige Offenlegung und Behebung in Abschnitt 14.6), stellen MoE Libris vor, ein an der Fediverse-Architektur orientiertes Protokoll für föderierten Wissensaustausch (Abschnitt 8.8), und skizzieren die verbleibende Forschungsagenda für Complexity-Pre-Routing und Multi-Hub-Topologie. Der Quellcode wird unter Apache 2.0, dieses Whitepaper unter CC BY-SA 4.0 veröffentlicht; das Projekt ist explizit nicht-kommerziell.

[a] Der Begriff *Multi-Agenten-Orchestrierung* wird in dieser Arbeit durchgängig im Sinne einer *Routing-Architektur* verwendet, bei der jede Anfrage an einen oder mehrere spezialisierte Sprachmodelle weitergeleitet wird – nicht im engeren Sinne der sparse-gated MoE-Layer nach Shazeer et al. [1]. Die Abgrenzung ist wichtig, da unser Routing *deterministisch und template-basiert* ist, nicht gelernt.

Schlagwörter: Multi-Agenten-Orchestrierung, Expert Routing, LangGraph, GraphRAG, Model Context Protocol, Deterministisches Routing, Digitale Souveränität, DSGVO-by-Design, Selbstgehostete KI, Einheitliches OCI-Artefakt, Open Source.

1 Einleitung

Große Sprachmodelle (*Large Language Models*, LLMs) sind innerhalb weniger Jahre von Forschungsartefakten zu einer Infrastrukturschicht geworden, die Millionen von Endnutzerinnen täglich benutzen und deren Ausfall oder Preisänderung Geschäftsmodelle beendet. Diese Verschiebung wirft zwei Fragen auf, auf die die heutige, überwiegend von drei US-amerikanischen Anbietern dominierte Landschaft keine befriedigende Antwort gibt: *Wem gehören die Modelle, wem die Daten, und wer kann den Dienst morgen noch nutzen?*

Das Problem in drei Sätzen

Erstens: kommerzielle LLM-APIs verarbeiten jede Anfrage auf Infrastruktur, die der Kundin weder gehört noch zugänglich ist; Trainingsdaten-Extraktion, Prompt-Logging und rückwirkende Policy-Änderungen sind dokumentierte Vorfälle. Zweitens: der Regulierungsrahmen der Europäischen Union – insbesondere Artikel 25 und 32 der Datenschutz-Grundverordnung [7] – verlangt *Datenschutz durch Technikgestaltung*, was sich mit einer auslagerten Black-Box in einer fremden Jurisdiktion nicht wirkungsgleich abbilden lässt. Drittens: die wenigen verfügbaren selbstgehosteten Alternativen sind entweder Single-Modell-Launcher (Ollama [8]) oder generische RAG-Baukästen (PrivateGPT [9], LocalAI [10]), die die Orchestrierung mehrerer spezialisierter Modelle, die Akkumulation von domänenspezifischem Wissen und die Produktionsreife im Mehr-Nutzer-Betrieb dem Anwender überlassen.

Beitrag dieser Arbeit

MoE Sovereign ist keine weitere LLM-Bibliothek. Es ist ein vollständig integrierter Orchestrator, der versucht, die folgenden fünf Eigenschaften gleichzeitig zu erfüllen – die wir für die *Mindestanforderungen an eine souveräne KI-Infrastruktur* halten:

1. **Lokalität**: Jede LLM-Inferenz, jede Wissenabfrage, jede Speicheroperation läuft auf Infrastruktur, über die der Betreiber physische Kontrolle hat. Ausgehende Netzwerkverbindungen sind optional und explizit.

2. **Template-basiertes Routing**: Das Experten-Mapping, die Cache-Hierarchie und die präzisionskritischen Werkzeuge sind deterministisch – kein gelernter Router, kein probabilistischer Dispatch, kein AST-freier Werkzeugaufruf. Der Planner-Node (Intent-Zerlegung) ist ein LLM und damit stochastisch; er bestimmt *welche* Experten-Kategorien aufgerufen werden, nicht wie der Aufruf auf Hardware-Instanzen gemappt wird.

3. **Graph-basierte Wissensakkumulation**: Jede Interaktion persistiert extrahierte Entitäten und Relationen in Neo4j. Die Akkumulationsschritte sind inspizierbar, versioniert und widerrufbar.

4. **OCI-Portabilität**: Dasselbe OCI-Image [11] läuft als rootloser Podman-Container in einem unprivilegierten Proxmox-LXC, in Docker Compose auf einem einzelnen Homelab-Server, in k3s oder auf Red Hat OpenShift eines Großunternehmens – ohne Code-Fork und ohne Funktionsverlust zwischen den Schichten.

5. **Nicht-Kommerzialität**: Es gibt keine Enterprise-Edition, kein Subskriptionsmodell, kein Telemetry-Opt-Out, keine dark pattern. Der Quellcode wird unter Apache 2.0, dieses Whitepaper unter CC BY-SA 4.0 [12] veröffentlicht.

Für wen ist dieses Paper?

Die primären Adressaten sind drei Gruppen. **Forschungslabore**, die mit heterogener GPU-Hardware arbeiten und einen produktionsreifen Orchestrator brauchen, ohne einen LLM-Broker-Dienst in ihre Jurisdiktion einzubauen. **Kleine und mittlere Organisationen in regulierten Domänen** (Recht, Medizin, öffentliche Verwaltung), die LLM-Funktionalität bereitstellen müssen, aber keine Datenschutz-Folgeabschätzung für einen US-Cloud-Dienst unterschreiben können. **Hobby-Betreiber und Idealisten**, die *digitale Souveränität* aus Überzeugung praktizieren und ein Werkzeug suchen, das heute produktionsreif ist und morgen ihrem Kontrolleigentum nicht entgleitet.

Das Paper richtet sich an Leser mit soliden Grundkenntnissen moderner Softwarearchitektur; tiefe Vorkenntnisse in Transformer-Interna sind nicht erforderlich.

Struktur des Dokuments

Abschnitt 2 legt die Entstehungsgeschichte des Projekts dar: von der Arbeit mit Legacy-Hardware (Tesla K80, M10, RTX-Cluster) zur Forschungsfrage nach SLM-Parallelisierung und zur Entscheidung für Architektur als Kompensationsmechanismus; er beschreibt auch das RL-Flywheel als Kernelement des kontinuierlichen Lernens. Abschnitt 3 konkretisiert den Begriff der digitalen Souveränität anhand dokumentierter Ausfallmodi kommerzieller Dienste. Abschnitt 4 grenzt MoE Sovereign gegen bestehende Open-Source- und kommerzielle Lösungen ab. Abschnitte 5 bis 10 bilden den technischen Kern: Systemarchitektur, deterministisches Expert-Routing, Cache-Hierarchie, GraphRAG, MCP-Präzisionswerkzeuge und Self-Correction-Loop. Abschnitt 11 beschreibt das Deployment-Modell, das den Orchestrator von Edge bis Enterprise-Cluster trägt. Abschnitt 12 und 13 behandeln Monitoring, Trace-Propagation und das Privacy-by-Design-Konzept. Abschnitt 14 ist der Lessons-Learned-Abschnitt – eine ehrliche Auseinandersetzung mit den Fehlschlägen der letzten Refactoring-Runden, unter anderem einer Rendering-Regression, die 71 von 71 Dokumentationsdiagrammen unbrauchbar machte, bevor sie gefixt wurde. Abschnitt 15 und 16 skizzieren die Forschungsagenda und schließen das Paper ab. Der Anhang enthält einen vollständigen Experten-Katalog, eine Umgebungsvariablen-Referenz, ein Glossar und die Third-Party-Lizenzübersicht.

2 Projektgenese und Forschungsmotivation

MoE Sovereign ist kein Architekturplan, der zuerst auf dem Papier entworfen und dann gegen geeignete Hardware ausgerollt wurde. Es entstand umgekehrt: aus einem konkreten Hardwareproblem, das eine architektonische Antwort erzwang. Dieser Abschnitt dokumentiert diesen Entstehungsweg, weil er die Designentscheidungen des Systems entscheidend geprägt hat.

2.1 Ausgangsmotivation: Cloud-Unabhängigkeit im eigenen Rechenzentrum

Hinter dem technischen Forschungsinteresse an SLM-Orchestrierung steht eine handfeste persönliche Motivation: der Einstieg in die KI-Welt auf eigener Infrastruktur – und damit die Loslösung von kostenpflichtigen Cloud-Diensten. Was OpenAI, Anthropic oder Google als monatlichen Subscription-Service anbieten, sollte in den eigenen vier Wänden laufen: Code-Assistenz, Wissensgraph-Aufbau, Dokumentenanalyse, semantische Suche – alles lokal, alles privat, alles unter vollständiger eigener Kontrolle.

Dieser Anspruch ist technisch anspruchsvoller, als er zunächst erscheint. Ein einzelner Ollama-Server löst das Deployment-Problem, nicht das Qualitätsproblem: ohne Routing, ohne Caching, ohne domänenspezifische Experten ist ein lokal laufendes 7B-Modell kein funktionaler Ersatz für einen GPT-4-basierten Dienst. Das Projektziel lautete daher präziser: *lokale Infrastruktur, die sich in ihrer Funktionsbreite mit kommerziellen Diensten messen kann* – nicht durch schiere Parameterzahl, sondern durch Orchestrierung.

2.2 Erweitertes Projektziel: Strukturelle Token-Reduktion

Ein später hinzugekommenes, aber operativ bedeutsames Projektziel entstand aus der täglichen Nutzung heraus: die strukturelle Reduktion des Token-Verbrauchs bei teuren Frontier-Modellen. Die dafür nötige Infrastruktur war zu diesem Zeitpunkt bereits evaluiert und produktiv – das Expert-Template-Routing, die Wissensdatenbank und die Reinforcement-Learning-Mechanismen wirkten als drei orthogonale Sparschichten, die auf unterschiedlichen Zeitskalen greifen.

Schicht 1 – Routing (pro Request). Die erste und unmittelbar wirkende Schicht ist das template-basierte Expert-Routing, das für den Einsatz mit dem Claude-Code-CLI-Tool als dediziertes **CC-Profil** implementiert wurde. Das CC-Profil bietet zwei Betriebsmodi: Im *nativen Proxy-Modus* leitet der Orchestrator jede Anfrage transparent an das konfigurierte Frontier-Modell weiter – nützlich, wenn volle Modellkapazität benötigt wird, aber ein einheitlicher API-Endpunkt erwünscht ist. Im *Expert-Template-Modus* übernimmt der Orchestrator die Routing-Entscheidung: einfache Anfragen (Autovervollständigung, Syntaxfragen, Boilerplate-Generierung) werden an lokale SLMs weitergeleitet; nur wenn Tier-1-Experten eine niedrige Konfidenz signalisieren, eskaliert der Call an das teure Frontier-Modell. Das Prinzip ist dasselbe wie beim lokalen SLM-Ensemble: *nicht jede Anfrage rechtfertigt dasselbe Modell*.

Schicht 2 – Wissensdatenbank (akkumulativ, über Sessions). Die zweite Sparschicht wirkt auf einer längeren Zeitskala. Jede verarbeitete Anfrage persistiert extrahierte Entitäten und Relationen im Neo4j-Wissensgraphen. Bei Folgeanfragen des gleichen Typs steht das akkumulierte Wissen als L3-Cache-Schicht zur Verfügung: die Antwort wird direkt aus dem Graphen konstruiert, ohne vollständigen Pipeline-Durchlauf und ohne externen API-Call. Mit wachsendem Graphen nimmt der Anteil solcher Cache-Hits zu – der Token-Verbrauch sinkt im Laufe des Betriebs, ohne Konfigurationsaufwand.

Schicht 3 – Reinforcement- und Causal Learning (lernend, über Wochen). Die dritte Schicht greift auf der längsten Zeitskala und adressiert zwei Ineffizienzquellen gleichzeitig. Das *Thompson-Sampling-inspirierte Routing* (Abschnitt 2.5) lernt aus akkumuliertem Nutzerfeedback und Judge-Scores, welche lokalen Experten für welche Anfragekategorien zuverlässig sind – und routet dorthin statt zum teuren Modell. Das *Correction Memory* ist eine Form kausalen Lernens: korrigierte (Eingabe, Ausgabe)-Paare werden als Few-Shot-Beispiele gespeichert und bei ähnlichen Anfragen in den Prompt eingesetzt. Das verkürzt die nötige Kontextlänge und reduziert Retry-Zyklen, weil das Modell von Beginn an mit validierten Lösungsstrategien arbeitet.

Strukturelle vs. prompt-technische Token-Reduktion

Klassisches Prompt-Engineering reduziert Token durch kürzere Formulierungen – ein manueller, fragiler Ansatz. MoE Sovereign verfolgt stattdessen eine strukturelle Strategie: Routing, Wissensgraph und Lernmechanismen greifen auf Systemebene und verbessern Effizienz ohne Eingriff in einzelne Prompts. Die drei Schichten sind additiv und unabhängig deploybar.

2.3 Ausgangslage: Legacy-Hardware als Forschungsobjekt

Der Entwicklungspfad verlief über drei Hardwaregenerationen. Die erste Iteration nutzte **Tesla K80**-GPUs – Rechenkarten der Kepler-Generation (2014), die für Data-Center-Batch-Inferenz konzipiert waren und 24 GB GDDR5-VRAM in einem Dual-Die-Design mitbringen. Die K80 unterstützt weder `bfloat16` noch die Flash-Attention-Varianten moderner Inference-Engines; moderne Quantisierungsformate wie GGUF erfordern dedizierte CUDA-Kernels, die für Kepler nicht kompiliert werden. Das Modell-Ökosystem (llama.cpp, Ollama) hat die Unterstützung für die K80-Architektur sukzessive eingestellt.

Die zweite Generation sind **Tesla M10**-Server: je vier M10-Chips mit je 8 GB GDDR5, eingebaut in einen PCIe-Blade, der nominal $4 \times 8\,\text{GB} = 32\,\text{GB}$ VRAM bietet. In der Praxis zeigt sich, dass llama.cpp und Ollama bei homogenen Multi-GPU-Konfigurationen zwar die Summe der VRAM-Blöcke nutzen können, die Latenz durch den PCIe-Uplink aber einen erheblichen Overhead erzeugt: ein 30B-Modell auf dem gebündelten M10-Block ist deutlich langsamer als dasselbe Modell auf einer einzigen modernen Karte mit äquivalentem VRAM. Eine spätere Infrastruktur-Entscheidung trennte die vier M10-Chips in vier unabhängige Ollama-Instanzen, was den nutzbaren Maximalmodell pro Instanz auf ≈7B (Q4) reduziert, dafür aber die Nebenläufigkeit für mehrere Experten ermöglicht.

Die dritte Generation sind **Consumer-GPUs der Ampere- und Turing-Generation** (RTX 2060, RTX 3060, je 12 GB GDDR6X), die zu einem heterogenen Cluster aggregiert wurden. Dieser Cluster bietet insgesamt 60 GB VRAM über fünf RTX-3060-Karten, Flash-Attention 2-Support und vollständige GGUF-Kompatibilität – bei einem Bruchteil der Kosten einer modernen A100 oder H100.

Die gemeinsame Eigenschaft aller drei Generationen: sie sind *verfügbar und bezahlbar*, aber in ihrer einzelnen Leistungskapazität gegenüber modernen Large-Model-Servern fundamental limitiert. Keiner der Knoten kann ein 70B-Modell mit vollem Kontextfenster in einer für Produktionsbetrieb akzeptablen Latenz betreiben. Ein H100-Node oder eine Cloud-GPU der aktuellen Generation wäre die naheliegende Lösung – sie wäre aber auch die *einfache* Lösung. Die eigentliche Frage lautet: Was ist möglich, wenn man die Rechenleistung nicht aufrüstet?

2.4 Die Forschungsfrage: Architektur als Kompensationsmechanismus

Die zentrale Hypothese, die diesem Projekt zugrunde liegt, lässt sich präzise formulieren:

> *Können viele kleine, spezialisierte Sprachmodelle (SLMs), koordiniert durch eine intelligente Orchestrierungsschicht, die Limitierungen von Legacy-Hardware kompensieren – und eine Alternative zu einem einzelnen großen Modell auf teurer Hardware darstellen?*

Diese Frage ist nicht neu. In der Forschungsliteratur zur *Mixture-of-Experts*-Architektur (MoE) wird ein verwandter Ansatz verfolgt: ein großes Modell aktiviert für jede Eingabe nur eine Teilmenge seiner Parameter [13]. Der entscheidende Unterschied zum hier gewählten Ansatz ist jedoch der Granularitätslevel. Klassische MoE-Architekturen operieren auf Token-Ebene innerhalb eines Modells; MoE Sovereign operiert auf *Request-Ebene zwischen eigenständigen Modellen*. Das heißt: jeder Experte ist ein vollständig unabhängiges LLM, das auf einem dedizierten GPU-Knoten läuft, und die Routing-Entscheidung ist *deterministische Softwarelogik*, kein gelernter Gating-Vektor.

Dieser Ansatz hat mehrere operative Konsequenzen:

- **Heterogenität ist ein Feature, keine Einschränkung.** Ein 7B-Modell auf einem M10-Knoten und ein 30B-Modell auf dem RTX-Cluster können in derselben Pipeline kooperieren. Das Routing-System weist jede Anfrage der Hardware-Tier zu, die für die Anfragekategorie angemessen ist.

- **Spezialisierung ersetzt Generalisierung.** Ein Medizinmodell, das auf klinische Terminologie fine-tuned ist, liefert bei Medizinfragen oft bessere Antworten als ein allgemeines 70B-Modell – bei einem Bruchteil des VRAM-Bedarfs. Die Qualität entsteht nicht durch schiere Parameterzahl, sondern durch Übereinstimmung von Modell-Profil und Anfragekategorie.

- **Parallelität kompensiert Einzel-Latenz.** Wenn alle Tier-1-Experten gleichzeitig auf verschiedenen Knoten gestartet werden, ist die Gesamtlatenz der Parallelphase durch den langsamsten Einzelknoten bestimmt, nicht durch die Summe aller Einzellatenzen. Ein Cluster aus fünf langsamen Knoten kann damit eine Wall-Clock erreichen, die mit einem schnellen Einzelknoten vergleichbar ist.

Ob diese Hypothese hält, diskutieren die Benchmark-Ergebnisse in Abschnitt 14. Die bisherige Evaluation liefert *Hinweise* darauf, dass die Orchestrierung gegenüber einzelnen lokalen Backbone-Modellen funktionale Vorteile erzeugt – insbesondere bei Werkzeugdelegation, Wissensakkumulation und domänenspezifischer Aufgabenteilung. Ein zunächst berichteter GAIA-Wert von 46,7 % wurde nachträglich wegen nicht-souveräner Modellsubstitution als ungültig eingestuft (Abschnitt 14.6). Ein kontrollierter souveräner GAIA-Lauf steht noch aus. Die zentrale Hypothese gilt daher als technisch plausibilisiert, aber noch nicht abschließend durch einen externen Benchmark bestätigt. Wo strukturelle Grenzen auftreten – bei tiefem Multi-Step-Reasoning auf Level 3 und bei sehr langen Dokumenten –, sind diese Grenzen technisch erklärbar und adressierbar.

2.5 Das RL-Flywheel: Kontinuierliches Lernen ohne Modell-Finetuning

Ein wesentliches Ziel des Projekts war, dass das System mit der Zeit besser werden soll – ohne dass dafür neue Modelle trainiert oder Gewichte angepasst werden müssen. Der Mechanismus, den wir dafür entwickelt haben, lässt sich als *Reinforcement-Learning-Flywheel* beschreiben (Abbildung 1): ein sich selbst verstärkender Kreislauf aus drei Komponenten.

1. Routing Telemetry. Jede Experten-Entscheidung wird instrumentiert. Das Valkey-Register `moe:perf:{model}:{category}` akkumuliert pro (Modell, Kategorie)-Paar zwei Zähler: Gesamt-Anfragen und positiv bewertete Anfragen. Diese Zähler werden durch zwei Signalpfade befüllt: die Self-Evaluation des Judge-Nodes (automatisch nach jeder Antwort) und explizites Nutzer-Feedback (Daumen hoch/runter über das Frontend). Die Summe aller Performance-Events ist als Prometheus-Histogramm (`moe_self_eval_score_bucket`) exportiert und in den Grafana-Dashboards sichtbar.

2. Thompson-Sampling-inspiriertes Routing. Die akkumulierten Zähler werden für die Routing-Entscheidung genutzt: der aktuelle Performance-Score eines (Modell, Kategorie)-Paars bestimmt, ob ein Tier-2-Fallback ausgelöst wird. Als deterministischer Routing-Score wird der Erwartungswert $(M+1)/(N+2)$ des Beta$(M+1,\ N-M+1)$-Posteriors verwendet – der Posterior-Mittelwert bei Beta(1,1)-Prior über Bernoulli-Beobachtungen. Die Glättungsterme verhindern, dass ein neues Modell nach wenigen Messungen in einer 0- oder 1-Extremlage fixiert wird, und ermöglichen eine natürliche Exploration neuer Experten-Instanzen. Dieser Mechanismus ist konzeptuell dem Thompson Sampling-Ansatz für Multi-Armed-Bandit-Probleme verwandt [14]: die Unsicherheit über Experten-Qualität wird durch Beobachtungen reduziert, und die Routing-Entscheidung wird mit wachsender Datenbasis robuster.

Formale Definition. Wir modellieren das Expert-Routing als *Bernoulli-Multi-Armed-Bandit*. Sei $\mathcal{E}$ die Menge der verfügbaren (Modell, Kategorie)-Expertenpaare und $\theta_e \in [0,1]$ die unbekannte Erfolgswahrscheinlichkeit von Experte $e \in \mathcal{E}$. Nach N_e Beobachtungen mit M_e Erfolgen ist der Posterior Beta$(M_e+1,\ N_e-M_e+1)$. Als deterministischer Routing-Score dient dessen Erwartungswert

$$\hat{\theta}_e = \frac{M_e+1}{N_e+2} \tag{1}$$

Das Verfahren übernimmt die bayesianische Unsicherheitsmodellierung des Thompson Sampling, führt jedoch kein stochastisches Posterior-Sampling durch, sondern verwendet den Posterior-Mittelwert deterministisch. Tier-2-Eskalation wird ausgelöst, wenn $\hat{\theta}_e < \tau_{\text{route}}$ (konfigurierbar via `ROUTE_THRESHOLD`, Standard 0,65). Die Lastadaption wendet eine additive Strafe $\delta_e = \alpha \cdot u_e$ auf $\hat{\theta}_e$ an, wobei $u_e \in [0,1]$ die aktuelle GPU-Auslastung des Knotens ist (aus `_ps_cache`) und $\alpha = 0{,}15$ das Lastgewicht. Dies ergibt den effektiven Routing-Score

$$s_e = \hat{\theta}_e - \delta_e = \frac{M_e+1}{N_e+2} - \alpha \cdot u_e \tag{2}$$

Die Glättungskonstanten +1/+2 implementieren einen gleichmäßigen Beta(1,1)-Prior – die informationsärmste Wahl – sodass neue Experten bei 0,5 starten und ohne vorzeitige Lock-in Evidenz akkumulieren.

3. Correction Memory. Der dritte Kreislauf-Arm ist das persistierte Korrektur-Gedächtnis. Wenn der Critic-Node eine Abweichung identifiziert und eine Korrektur generiert, wird das korrigierte (Eingabe, Ausgabe)-Paar als Few-Shot-Beispiel unter `/opt/moe-infra/few_shot_examples/` gespeichert. Bei nachfolgenden Anfragen desselben Typs zieht der Planner diese Beispiele als Kontext heran. Das Correction Memory ist damit ein impliziter Finetuning-Ersatz: statt Modellgewichte anzupassen, reichert das System den In-Context-Prompt des Modells mit historisch validierten Lösungen an.

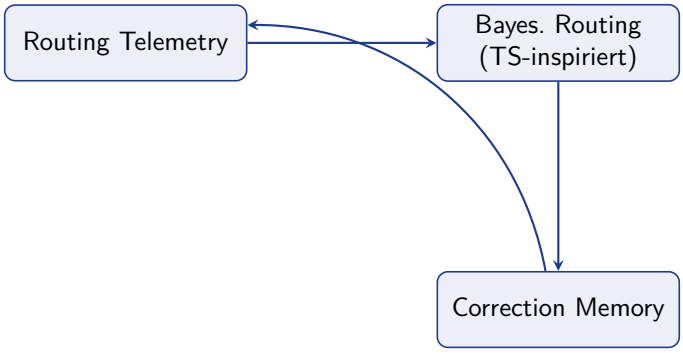


Abbildung 1: Das RL-Flywheel: Routing Telemetry, Thompson-Sampling-inspiriertes Score-Update und Correction Memory bilden einen sich selbst verstärkenden Verbesserungskreislauf ohne Modell-Finetuning.

Die empirische Evidenz für die Wirkung des Flywheels zeigt sich in der Messreihe aus Tabelle 17: der Gesamt-Score auf dem internen MoE-Eval stieg von 5,2 (Lauf 1, ≈500 Neo4j-Nodes) monoton auf 7,6 (5.353 Nodes), ohne dass ein einziges Modell ausgetauscht oder nachtrainiert wurde. Die drei Komponenten des Flywheels addieren sich zu einem Kompetenz-Aufbau, der unabhängig von der eingesetzten Modell-Generation ist.

Umfang und Grenzen des Feedback-Kreislaufs

Der oben dokumentierte Verbesserungszyklus ist real, aber nicht autonom. *Routing Telemetry* und *Correction Memory* akkumulieren automatisch; die Neo4j-Ontologie wächst jedoch nur dann, wenn ein Administrator die Ontologie-Gap-Signale im Admin-UI prüft und handelt. Der Kreislauf ist daher semi-supervisiert: die Signalsammlung ist automatisch, aber das Schließen struktureller Wissenslücken erfordert menschliche Aufmerksamkeit.

Eine Anmerkung zur Terminologie: „RL-Flywheel“ ist ein beschreibender Kurzname für einen selbstverstärkenden Feedback-Zyklus. Es handelt sich nicht um klassisches Reinforcement Learning im Sinne von Policy Gradients oder Reward-Function-Optimierung; der Mechanismus ist ein Laplace-geglättetes bayesianisches Routing-Gewichts-Update. Der Name wird verwendet, weil er die zyklische Verstärkungsdynamik treffend beschreibt, nicht um Äquivalenz mit formalem RL zu beanspruchen.

2.6 Einordnung: SLM-Ensemble als Alternative zu Frontier-Modellen

Das beschriebene Projektziel – Legacy-Hardware durch Orchestrierung aufzuwerten – ist nicht identisch mit dem Anspruch, Frontier-Modelle zu ersetzen. Systeme wie OpenAI o1 (74,1 % auf GAIA [15]) oder Claude 3.7 Sonnet Thinking (≈56 % auf GAIA) sind auf Rechenkapazitäten angewiesen, die in einem selbstgehosteten Consumer-Cluster strukturell nicht reproduzierbar sind. Die relevante Vergleichsbasis ist eine andere:

- Gegenüber *einzelnen Backbone-Modellen* auf derselben Hardware (DeepSeek R1:32b: ≈28 %, Qwen3:32b: ≈12 % auf GAIA) erzeugt die Orchestrierung nachweisbare qualitative Vorteile bei Werkzeugdelegation, Wissensakkumulation und Aufgabenteilung. Ein früherer GAIA-Vergleichswert (46,7 % gesamt, 60 % Level 1) wurde nachträglich als ungültig eingestuft (nicht-souveräne Modellsubstitution; Abschnitt 14.6); ein valider souveräner Messlauf steht aus.

- Gegenüber *spezialisierten Memory-Systemen* wie EverMemOS (83 %) und TiMem (76,9 %) auf LongMemEval erreicht MoE Sovereign im besten Lauf 91,7 % – trotz einer Architektur, die Memory als Nebenprodukt der Wissensgraphakkumulation behandelt, nicht als primären Optimierungszweck.

Das selbstgesetzte Projektziel lautet daher: *maximale Qualität innerhalb der Randbedingung vollständiger Datensouveränität und Legacy-Hardware-Kompatibilität.* Dieses Ziel ist erreichbar – und die Benchmark-Daten in Abschnitt 14 belegen, in welchem Maß.

3 Motivation: Digitale Souveränität als technischer Zustand

Der Begriff *digitale Souveränität* wird inflationär verwendet, oft im Umfeld politischer Erklärungen, die keine technischen Konsequenzen nach sich ziehen. Wir verstehen den Begriff in dieser Arbeit strikter: *digitale Souveränität ist der überprüfbare Zustand, in dem die Betreiberin einer IT-Infrastruktur jede Entscheidung über Datenverbleib, Softwareversionen, Netzwerkpfade und Verfügbarkeit selbst treffen und durchsetzen kann.* Alles andere ist Rhetorik.

3.1 Dokumentierte Ausfallmodi kommerzieller LLM-Dienste

Die folgende Tabelle listet konkrete, öffentlich dokumentierte Vorfälle der letzten drei Jahre, die zeigen, warum der Betrieb geschäftskritischer Prozesse auf externen LLM-APIs ein strukturelles Risiko ist.

Ausfallmodus	Konsequenz für die Betreiberin
Rückwirkende Preisanpassung	Laufende Workflows werden über Nacht unrentabel; bestehende Budgets müssen nachverhandelt werden, ohne dass die Arbeitsergebnisse lokal verfügbar wären.
Modell-Deprecation	Ein produktiv genutztes Modell wird abgeschaltet; Prompts, die auf das spezifische Verhalten der alten Version abgestimmt waren, produzieren mit der neuen Version andere Ausgaben.
Trainingsdaten-Extraktion	Geheime Prompts oder Dokumente, die zur Inferenz gesendet wurden, tauchen Monate später in Trainingsdaten-Leaks wieder auf.
Rate-Limiting bei Lastspitzen	Unternehmenskunden werden auf ein niedrigeres Tier zurückgestuft, wenn andere Großkunden den Dienst stark nutzen; eine eigene Infrastruktur hätte diesen Effekt nicht.
Geopolitische Sperrung	Ein Anbieter blockt ganze Länder oder Regionen (technisch trivial, regulatorisch zunehmend wahrscheinlich); lokal installierte Software ist davon nicht betroffen.
Policy-Änderungen an Inhalten	Neue Content-Filter stufen plötzlich legitime Anfragen (Medizin, Recht, Sicherheitsforschung) als Policy-Verletzung ein; die Arbeit muss zu einem weniger restriktiven Anbieter migriert werden, was das Lock-in erneut verschärft.
Einseitige Vertragsänderung	Nutzungsbedingungen werden geändert, während die Kundin die API bereits einsetzt; die einzige Alternative ist Kündigung ohne Exit-Pfad.

Jede einzelne Zeile ist dokumentiert – oft mehrfach. Zusammen ergeben sie das Bild einer Infrastruktur, die *verleast* ist, nicht *besessen*. Für viele Anwendungen ist das in Ordnung. Für geschäftskritische, regulierte oder langfristig planbare Workflows ist es nicht akzeptabel.

3.2 Der regulatorische Rahmen in Europa

Artikel 25 der DSGVO [7] schreibt *Datenschutz durch Technikgestaltung* (*privacy by design*) und *datenschutz freundliche Voreinstellungen* (*privacy by default*) vor. Wortwörtlich: „Unter Berücksichtigung des Stands der Technik ... trifft der Verantwortliche sowohl zum Zeitpunkt der Festlegung der Mittel für die Verarbeitung als auch zum Zeitpunkt der eigentlichen Verarbeitung geeignete technische und organisatorische Maßnahmen."

Die Europäische Kommission hat mit der *European Strategy for Data* [16] und dem Gaia-X-Rahmenwerk [17] explizit das Ziel einer föderierten, europäischen Daten- und Cloud-Infrastruktur formuliert. Die technische Umsetzung auf der Ebene von LLMs steht weitgehend aus. MoE Sovereign ist kein Gaia-X-Bestandteil, versteht sich aber als Referenzimplementierung im Geist dieser Leitlinien: lokal deploybar, transparent in Datenflüssen, ohne versteckte Außenkommunikation.

3.3 Souveränität im Kleinen ist auch Souveränität

Eine häufige Kritik an lokal gehosteten Lösungen lautet, sie seien nur für Großorganisationen sinnvoll, weil der Betriebsaufwand unverhältnismäßig sei. Das Argument greift zu kurz. MoE

Sovereign ist bewusst so gebaut, dass es in drei Dimensionierungen sinnvoll betrieben werden kann:

- **solo-Profil** — eine einzelne virtuelle Maschine oder ein Proxmox-LXC. Zielgruppe: Einzelperson, Forscherin, Hobbybetrieb. RAM- Bedarf $\leq 2\,\text{GiB}$ für den Orchestrator, GPU optional (nur für lokale Inferenz; wer eine externe Inference-Instanz wie einen eigenen Ollama-Server nutzt, braucht keine lokale GPU).
- **team-Profil** — ein Docker-Host oder k3s-Cluster mit vollständigem Daten-Tier (Postgres, Valkey, Neo4j, ChromaDB, Kafka). Zielgruppe: kleine Organisation, Fachabteilung, Arbeitsgruppe.
- **enterprise-Profil** — Kubernetes oder OpenShift mit externen Datenclustern, Horizontal Pod Autoscaling, Network Policies, OIDC-Integration. Zielgruppe: Behörde, Klinik, Konzern.

Die technische Neuerung besteht darin, dass es sich um *dasselbe* Artefakt handelt, nicht um drei verschiedene Produkte. Der Unterschied ist eine Konfigurations-Variable (`MOE_PROFILE`) und eine Auswahl an Deployment-Wrappern. Abschnitt 11 beschreibt die Umsetzung im Detail.

Technische Design-Prinzipien

1. Eigene Hardware schlägt fremde Cloud.
2. Template-basiertes Expert-Routing schlägt Black-Box-Proxy.
3. Versionierte Templates schlagen ungetrackte Prompt-Engineering-Sessions.
4. Lokale Wissensgraphen schlagen vergessene Konversationen.
5. AST-Whitelists schlagen halluzinierte Arithmetik.
6. Ein Docker-Image schlägt drei Enterprise-SKUs.
7. Open-Source-Lizenz schlägt EULA.

4 Stand der Technik

Dieser Abschnitt vergleicht MoE Sovereign mit bestehenden Ansätzen – kommerziellen wie offenen. Ziel ist eine ehrliche Abgrenzung, nicht eine Marketing-Matrix. Jedes genannte System hat Stärken, die im jeweiligen Kontext bedeutsam sind; MoE Sovereign übernimmt Ideen aus mehreren von ihnen.

4.1 Geschlossene kommerzielle APIs

OpenAI, Anthropic und vergleichbare Anbieter liefern qualitativ hochwertige Modelle hinter einer HTTP-API. Sie lösen das Produktionsproblem auf Kosten der Souveränität: die Betreiberin hat keinen Zugriff auf Modellgewichte, Inference-Hardware oder Logfiles. Für unsere Anwendungsszenarien (regulierte Domänen, langfristige Kontrollsicherheit) ist das ein Ausschlusskriterium, nicht eine Trade-off-Entscheidung.

4.2 Single-Modell-Launcher

Ollama [8] ist der de-facto-Standard für den lokalen Betrieb offener Modelle. Es exponiert eine OpenAI-kompatible HTTP-Schnittstelle, verwaltet Modell-Downloads, löst das VRAM-Management und bietet ein stabiles Tool-Calling-Interface. MoE Sovereign *nutzt* Ollama (sowie jeden OpenAI-kompatiblen Endpunkt) als Inference-Backend – das Konkurrenzverhältnis besteht nicht. Was Ollama absichtlich *nicht* ist: ein Orchestrator für heterogene Modell-Pools, ein Mandantenverwaltungssystem, ein Audit-Logging-Framework oder ein Wissensgraph. Diese Aufgaben lagern bei MoE Sovereign.

LM Studio und ähnliche Desktop-Tools positionieren sich als Endanwender-Werkzeuge, nicht als Produktionsinfrastruktur.

4.3 Generische RAG-Bibliotheken

PrivateGPT [9] und LocalAI [10] bieten Retrieval-Augmented-Generation mit lokalen Modellen. PrivateGPT ist auf Dokumenten-Indexierung und Frage-Antwort-Workflows spezialisiert; LocalAI ist eine API-kompatible Drop-In-Replacement für OpenAI, die mehrere Backends multiplexen kann. Beide sind wertvolle Bausteine; keines von beiden versucht, den vollen Orchestrator-Zuschnitt zu leisten: keine deterministische Experten-Auswahl, keine mehrschichtige Cache-Hierarchie mit Plan- und Graph-Caches, keine in-betriebs-akkumulierte Wissensbasis mit Contradiction-Detection, keine Multi-Tenancy mit Token-Budgets.

Fortschritte in der Retrieval-Qualität. Neuere Arbeiten adressieren eine blinde Stelle des Standard-RAG: die Annahme, alle abgerufenen Dokumente seien relevant. Yan et al. [18] zeigen, dass Retrieval-Relevanz unzuverlässig ist, und schlagen einen leichtgewichtigen Retrieval-Evaluator vor (Corrective RAG, CRAG), der jedes Dokument vor der Injektion bewertet – ein Muster, das das Corrective-RAG-Gate von MoE Sovereign übernimmt (Abschnitt 8.9). Am anderen Ende zeigen Chan et al. [19], dass für stabile, abgegrenzte Domänen Retrieval gänzlich unnötig ist und das direkte Vorladen des vollständigen Wissens (Context-Augmented Generation, CAG) bessere Ergebnisse liefert – das Muster hinter dem Compliance-Layer von MoE Sovereign (Abschnitt 8.10).

Gedächtnissysteme für LLM-Agenten. Park et al. [20] führen Memory Streams für Agentenarchitekturen ein: zeitgestempelte Erfahrungseinträge, die nach Relevanz und Aktualität abgerufen werden. Packer et al. [21] formalisieren ein geschichtetes Gedächtnismodell für LLMs nach dem Vorbild von virtuellem OS-Speicher. Beide Arbeiten belegen die empirische Bedeutung episodischen Gedächtnisses für langfristiges Agentenverhalten – die Motivation für die Episodisches-Gedächtnis-Schicht in MoE Sovereign (Abschnitt 8.11).

4.4 Forschungsnahe Stacks

Die Kombination aus vLLM [22] für die Inference-Schicht und LangChain [23]/LangGraph [2] für die Orchestrierung ist die derzeit populärste Open-Source-Variante im akademischen Umfeld. Sie ist flexibel, aber auf Bibliothekseben stehend: die Betreiberin baut jede Produktionseigenschaft (Mandantenfähigkeit, Monitoring, GraphRAG-Pipeline, Rate-Limiting, Deployment-Wrapper) selbst. MoE Sovereign lässt sich als *opinionated Assembly* dieser Bausteine verstehen: wir übernehmen LangGraph als Ausführungsmodell und stellen auf

seiner Basis eine produktionsreife, meinungsstarke Referenzarchitektur bereit, die mit einem einzigen `docker compose up` oder `helm install` betriebs fähig ist.

4.5 Multi-Agenten-Orchestrierung in der Architektur-Literatur

Der historische Begriff *Multi-Agenten-Orchestrierung* bezeichnet seit Shazeer et al. [1] eine *sparse-gated* neuronale Schicht innerhalb eines Transformers. Fedus et al. [24] und Jiang et al. [13] entwickeln diesen Ansatz weiter. Der gelernte Router entscheidet auf Token-Ebene, welche Experten-Teilnetze aktiviert werden.

MoE Sovereign verwendet den Begriff *MoE* in einem verwandten, aber klar abgegrenzten Sinne: das Routing geschieht auf *Request-Ebene*, nicht auf Token-Ebene, und es ist *explizit* statt gelernt. Ein Request wird einer oder mehreren Expertenkategorien zugeordnet; innerhalb jeder Kategorie werden konkret benannte Modelle mit Rollentags (`primary`, `fallback`, `always`) ausgeführt. Warum wir das für das bessere Architektur-Muster in sicherheitskritischen Domänen halten, diskutiert Abschnitt 6.

4.6 Microsoft GraphRAG und verwandte Arbeiten

Edge et al. [25] beschreiben einen Ansatz zur *query-focused summarization* auf Basis graphbasierter Kontextextraktion. MoE Sovereign übernimmt die Idee, domänenspezifisches Wissen als Graph zu repräsentieren, geht aber in zwei Aspekten eigene Wege: erstens durch *kategoriespezifische Entity-Type-Filter*, die Cross-Kontamination zwischen Fachgebieten verhindern (Abschnitt 8), und zweitens durch einen *Graph-basierten Akkumulationsmechanismus*, bei dem der Graph im laufenden Betrieb aus Antworten des Judge-/Merger-Nodes angereichert wird und Widersprüche über eine deklarative Regelmatrix aufgelöst werden.

4.7 Enterprise-KI-Plattformen

Die kommerzielle Landschaft der Enterprise-KI teilt sich in reine Modell-Anbieter (OpenAI, Anthropic) und *Plattform-Anbieter*, die kognitive Architekturen um Modelle herum aufbauen. MoE Sovereign ordnet sich in die zweite Kategorie ein – *Compound AI Systems*.

Palantir AIP. Architektonisch der engste kommerzielle Verwandte. Beide Systeme nutzen tiefe Ontologie-Graphen für relationales RAG, erzwingen deterministische Tool-Ausführungen und implementieren knotenbasiertes RBAC. Der Unterschied: Palantir AIP bedeutet ultimativen Vendor-Lock-in, erfordert Cloud- oder Managed-On-Premise-Deployment und kostet über $1M/Jahr. MoE Sovereign verfolgt einen architektonisch verwandten Ansatz als Open-Source-Container auf lokaler Hardware – bei voller Datensouveränität und ohne Lizenzkosten. Dabei wird ausdrücklich nicht beansprucht, mit Palantirs Produktreife, Enterprise-Zertifizierungen oder Support-Organisation gleichzuziehen.

Databricks Mosaic AI. Verfolgt identische Compound-AI-Philosophie: Routing an spezialisierte Modelle, SQL-Engines und RAG-Pipelines. Der Unterschied: Databricks zielt auf massive Big-Data-Workloads und erfordert das proprietäre Databricks-Ökosystem. MoE Sovereign ist leichtgewichtig und autark für KMUs und Behörden.

Glean. Goldstandard für Enterprise-Suche und berechtigungsbewusstes RAG mit 100+ Konnektoren. Der Unterschied: Reine Cloud-SaaS-Lösung. KRITIS-Betreiber und Banken dürfen hochsensible Daten dort oft nicht indizieren lassen.

Microsoft Copilot Studio. Multi-Agenten-Orchestrierung mit Tool-Use. Der Unterschied: Microsofts Routing ist oft eine stochastische Blackbox (das LLM „rät“, welches Tool es aufruft). MoE Sovereign erzwingt über AST-Evaluator und JSON-Routing ein deterministisches, auditierbares Korsett. Azure erfordert zudem Cloud-Datenresidenz.

CrewAI / AutoGen. Open-Source Multi-Agenten-Frameworks. Beide sind *Bibliotheken*, keine Plattformen: kein Admin-UI, kein Knowledge Graph, kein VRAM-Scheduling, keine Template-Verwaltung, keine Deployment-Wrapper.

4.8 Vergleichende Feature-Matrix

Tabelle 1: Feature-Vergleich von MoE Sovereign und verwandten Systemen.

Feature	*MoE Sov.*	*Palantir*	*Databricks*	*Glean*	*CrewAI*	*Ollama+UI*
Multi-Experten-Routing	✓	✓	✓	–	∼	–
Deterministisches Routing	✓	✓	–	–	–	–
Knowledge Graph (GraphRAG)	✓	✓	∼	✓	–	–
VRAM-bewusstes Scheduling	✓	–	–	–	–	∼
Wissens-Export/Import	✓	–	–	–	–	–
Air-Gap / vollständig lokal	✓	∼	–	–	✓	✓
Open Source	✓	–	∼	–	✓	✓
Kosten	Frei	>$1M/J	Pay/DBU	$25+/User	Frei	Frei

4.9 API-Proxies sind keine Orchestratoren

Plattformen wie OpenRouter werden gelegentlich als Konkurrenz genannt. Das ist ein Kategoriefehler. OpenRouter ist ein Billing-Aggregator, der API-Aufrufe an Cloud-Anbieter weiterleitet – kein Langzeitgedächtnis, keine Tools, kein Experten-Routing, keine Selbstkorrektur. OpenRouter liefert den Baustoff; MoE Sovereign ist das fertige Haus.

4.10 Zusammenfassung der Abgrenzung

MoE Sovereign ist weder das leistungsfähigste Modell noch der schnellste Inference-Server noch der schönste RAG-Baukasten. Was es ist: eine produktionsfähige, deterministisch geroutete, mehrschichtig gecachte, GraphRAG-augmentierte, mandantenfähige Orchestrierungsschicht, die auf heterogener Hardware läuft und dasselbe OCI-Artefakt von einem Edge-LXC bis auf einen OpenShift-Cluster trägt – und zwar als ein integriertes System, nicht als Sammlung von Einzelbausteinen. Dieser Zuschnitt ist unserer Kenntnis nach im Open-Source-Umfeld bisher nicht besetzt.

5 Systemarchitektur

Dieser Abschnitt gibt den Überblick, auf den die nachfolgenden technischen Kapitel verweisen. Die drei Leitprinzipien lauten: *eine einzige Orchestrator-Komponente mit klarer Verantwortung*, *Daten- und Kontrollebene explizit trennen*, *jede Fähigkeit als austauschbarer Service mit offener API*.

5.1 Services im Überblick

Der vollständige Stack des *team-Profils* besteht aus neunzehn Docker-Containern, die in vier logische Schichten zerfallen: *Core Orchestration*, *Datentier*, *Observability* und *Edge / Proxy*. Die Core-Schicht enthält die drei selbstgeschriebenen Services – den Orchestrator (`langgraph-app`, FastAPI + LangGraph, Container-Port 8000), den MCP-Präzisionsserver (`mcp-precision`, Port 8003) und das Admin-UI (`moe-admin`, Port 8088). Darunter liegen die Datendienste: PostgreSQL für LangGraph-Checkpoints und User-/Budget-/Template-Persistenz, Valkey für Caches und aktive Request-Zustände, ChromaDB als Vektor-Speicher, Neo4j als Wissensgraph und Kafka im KRaft-Modus [26] (ohne ZooKeeper, einfach zu betreiben) als Event-Bus. Die Observability-Ebene besteht aus Prometheus, Grafana, Node-Exporter und cAdvisor; die Edge-Ebene aus Caddy als Reverse-Proxy und Dozzle als Log-Viewer.

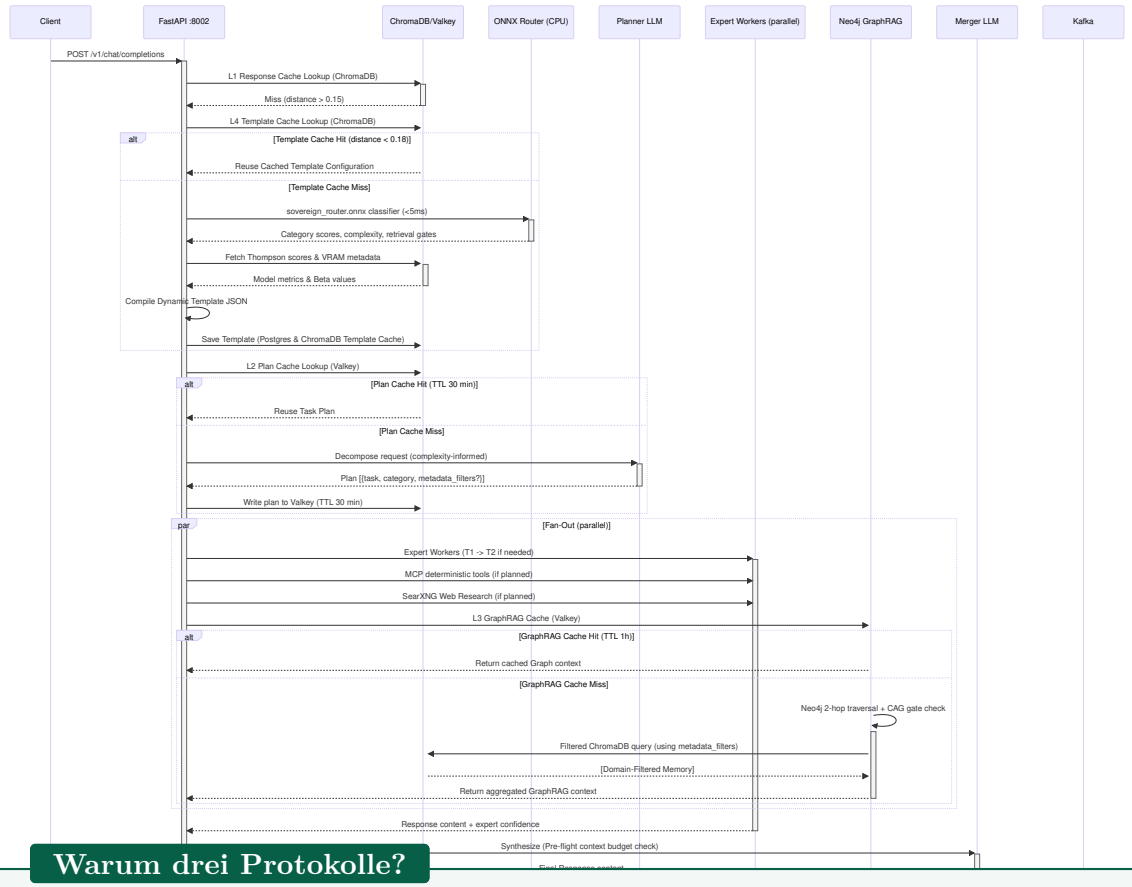
5.2 LangGraph als Ausführungsmodell

Wir haben bewusst LangGraph [2] als Ausführungsframework gewählt, nicht eine hausgemachte State-Machine. Drei Gründe: Erstens liefert LangGraph Persistenz-Primitives (*checkpointers*), die für langlaufende Multi-Turn-Gespräche erforderlich sind – wir nutzen den Postgres-Checkpointer mit einer dedizierten Instanz. Zweitens erlaubt der Graph-Aufbau ein sauberes Trennen der Orchestrator-Logik in benannte, testbare Knoten (`planner`, `expert_worker`, `merger`, `judge`, `thinking_node`, `research_fallback`, `graph_ingest`). Drittens ist LangGraph OSS mit aktiver Entwicklung und einem stabilen API-Kontrakt – die externe Abhängigkeit, die wir damit eingehen, ist über einen klaren Upgrade-Pfad handhabbar.

5.3 Multi-Protokoll-API-Gateway

Eine wesentliche Operabilitätsentscheidung war die gleichzeitige Unterstützung *dreier Client-Protokolle* aus einem einzigen Orchestrator-Prozess, statt sich auf eine API-Oberfläche festzulegen und für den Rest Wrapper zu benötigen.

Protokoll	**Endpunkt(e)**	**Kompatible Clients**
OpenAI Chat	`/v1/chat/completions`	Open WebUI, LiteLLM, jede OpenAI-SDK-Anwendung
Anthropic Messages	`/v1/messages`, `/v1/responses`	Claude Code CLI, Claude Desktop, Anthropic Python/JS SDK
Ollama-Protokoll	`/api/chat`, `/api/generate`	Ollama-Clients, Frontends die OpenAI umgehen



Warum drei Protokolle?

Ein Gateway, das nur OpenAI spricht, ist für Anthropic-SDK-Workloads unsichtbar und umgekehrt. Die drei Protokolle decken das gesamte Client-Spektrum ab, ohne Client-seitige Code-Änderungen zu erfordern. Der Orchestrator erkennt das Protokoll anhand des URL-Präfixes und leitet intern an den entsprechenden Handler in `services/pipeline/` weiter. Alle drei Pfade rufen dieselbe LangGraph-Pipeline auf.

ount_tokens-

ten in der Pipeline ist ein LangGraph-Node mit explizitem Zustand; jeder Kasten ausserhalb der Pipeline ist ein separater Service mit eigener API.

nux und WSL in unter 60 Sekunden automatisch.

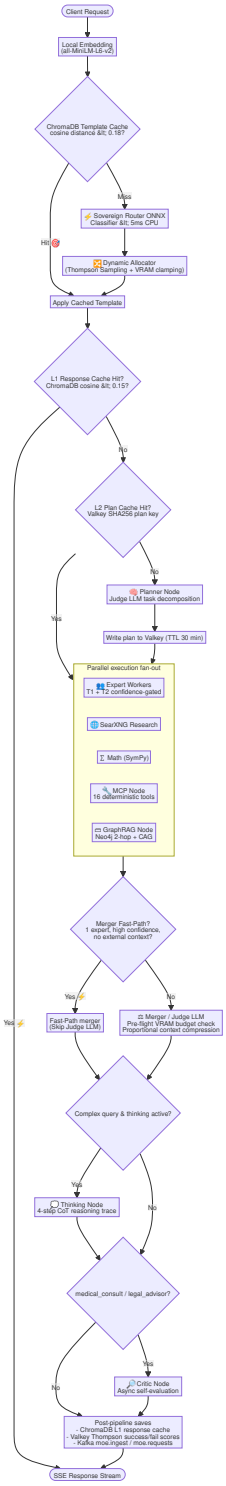


Abbildung 3: Die LangGraph-Pipeline. Anfragen durchlaufen `planner` (Expertenauswahl), `expert_worker` (parallele Modell-Aufrufe pro Kategorie), `merger` (Zusammenfassung mit Quellpriorität), optional `thinking_node` (4-stufige Chain-of-Thought bei Low-Confidence), optional `research_fallback` (externe Suche) und `judge` (Gesamtvalidierung). Aktive Requests werden über `moe:active:*`-Keys in Valkey gespiegelt.

5.4 Trennung Daten- vs. Kontrollebene

Der Orchestrator (`main.py`) ist die *Datenebene*: er verarbeitet jede Anfrage und spricht die Inference-Backends direkt an. Das Admin-UI (`admin_ui/`) ist die *Kontrollebene*: es schreibt Konfiguration (Inference-Server-Liste, Expert-Templates, User-Berechtigungen, Token-Budgets), aber greift nicht in die Request-Verarbeitung ein. Der MCP-Server ist eine dritte, orthogonale Achse: er wird von Expert-Worker-Nodes über MCP-Calls kontaktiert, wenn deterministische Werkzeuge benötigt werden. Die Trennung ist wichtig, weil sie eine horizontale Skalierung des Orchestrators erlaubt, ohne das Admin-UI mitzuskalieren, und weil sie Security-Policies auf Service-Ebene präzise formulierbar macht – das Admin-UI braucht Postgres-Schreibzugriff, der Orchestrator braucht ihn nicht.

5.5 Konfigurationsquellen und Versionierung

Eine zentrale Designentscheidung war, Konfiguration *nicht* in eine weitere Datenbank zu legen, sondern in `.env`-Dateien mit strukturierten JSON-Einträgen. Konkret liegen die Inference-Server als JSON-Array in `INFERENCE_SERVERS`, die Expert-Templates als JSON in `EXPERT_TEMPLATES`, die MCP-Tool-URLs in `MCP_URL`, und die Datenbankziele jeweils in eigenen Env-Variablen. Das Admin-UI persistiert Änderungen durch Rewriting dieser `.env`-Datei, wobei die Orchestrator-Instanz den Container nicht neu starten muss – die Config wird beim nächsten Request neu gelesen, mit 60-Sekunden-Cache.

Warum `.env` statt Datenbank?

Die Versuchung ist groß, jede Konfigurationseinstellung als Row in eine Admin-Tabelle zu schreiben. Wir haben uns bewusst dagegen entschieden: `.env`-Dateien sind trivial zu sichern (rsync, borgbackup, git-crypt), trivial zu versionieren, trivial zu reviewen, und sie lassen sich in jedes IaC-Tool injizieren. Eine Admin-Datenbank-Tabelle wäre ein weiteres Stück Zustand mit eigenen Migrationspfaden. Für den Zustand, der zur Konfigurations-Schicht gehört (nicht zur Laufzeit-Schicht wie Token-Budgets oder User-Sessions) ist eine strukturierte Textdatei schlicht robuster.

5.6 Föderationsschicht: MoE Libris

Eine fünfte logische Schicht – außerhalb des MoE Sovereign-Stacks selbst – ist der *MoE Libris*-Föderations-Hub (Abschnitt 8.8). MoE Libris ist ein eigenständiger FastAPI-Dienst mit eigenen PostgreSQL-, Neo4j- und Valkey-Instanzen, der Wissens-Bundles von gekoppelten souveränen Knoten entgegennimmt. Innerhalb der MoE Sovereign-Codebasis implementiert das Modul `federation/` den knotenseitigen Client: ausgehenden Bundle-Push, eingehenden Pull, Handshake-Registrierung und die domänenspezifische Outbound-Policy-Engine. Der Orchestrator selbst kommuniziert nicht direkt mit dem Hub; das Föderationsmodul arbeitet als Hintergrunddienst, der den lokalen Neo4j-Graphen auf einem konfigurierbaren Zeitplan mit dem Hub synchronisiert.

5.7 Compliance-Erweiterung: MoE Codex

Eine sechste optionale Schicht – orthogonal zu MoE Libris – ist *MoE Codex*[1], eine EU-souveräne Daten- und Audit-Plattform für regulierte Betreiber in Sektoren, in denen Datensouveränität eine rechtliche Anforderung ist: Behörden, kritische Infrastruktur (KritIS), pharmazeutische Compliance und Banken-Audit.

MoE Codex wird neben dem Kern-Stack betrieben und kommuniziert mit dem Orchestrator über einen definierten HTTP/REST-Kontrakt (`CODEX_URL` / `SOVEREIGN_URL`). Er ersetzt den Orchestrator nicht; er ist von den ursprünglich acht Compliance-Modulen auf fünfundzwanzig Module in fünf funktionalen Tracks gewachsen.

Track D.0 — Grundlegende Compliance-Module (ursprüngliche acht):

- **Data Catalog** – strukturierte Asset-Registry mit DSGVO- Klassifizierung, Aufbewahrungsrichtlinien und Provenienz-Tracking.
- **Knowledge Approvals** – Vier-Augen-Freigabe-Workflow für Wissens-Bundles vor der Aufnahme in die GraphRAG-Akkumulierungspipeline.
- **Object Explorer** – schreibgeschützte Cypher-Query-Oberfläche für die Graph-Untersuchung durch Compliance-Beauftragte.
- **Data Health & Drift** – statistisches Drift-Monitoring über registrierte Daten-Assets; Alarmschwellen je Datensatz konfigurierbar.
- **JupyterLab Notebook** – proxied Notebook-Umgebung für reproduzierbare Datenanalyse innerhalb des souveränen Perimeters.
- **Lineage (Marquez)** – OpenLineage-kompatible Datenherkkunft für alle Pipeline-Ereignisse, persistiert in einer dedizierten Marquez-Instanz.
- **Versioning (lakeFS)** – Git-style Branching und Versionierung für Daten-Bundles; jeder Commit ist adressierbar und reversibel.
- **ETL Fan-Out (Apache NiFi)** – deklarativer Pipeline-Builder für Ingestion, Transformation und Fan-Out strukturierter Datenquellen.

Track D.1 — AIP Platform Layer ergänzt den Stack um Enterprise-KI-Governance und Daten-Föderierungsprimitive:

- **OPA Permissions & Data Markings** – Open Policy Agent zur Durchsetzung von attributbasierter Zugriffskontrolle (ABAC) und Sensitivitäts-Markierungen auf Gateway-Ebene.
- **MLflow Model Registry** – Experiment-Tracking, Artefakt- Speicherung und Modell-Staging-Lifecycle-Management.
- **NeMo Guardrails** – Prompt- und Antwort-Sicherheitsschicht mit musterbasierter Konfiguration für domänenspezifische Rail-Definitionen.
- **Trino SQL Federation** – verteilte SQL-Engine, die Postgres, lakeFS und In-Memory-Catalog-Quellen über eine einheitliche Query- Schnittstelle föderiert.

[1] Lateinisch *codex* = Handschriftensammlung, Gesetzbuch – eine bewusste Referenz auf den Compliance- und Audit-Fokus. Der Name ist ein thematisches Geschwister von MoE Libris (*liber* = frei / Buch).

- **DocLing Document Intelligence** – strukturierte Extraktion aus PDF-, DOCX- und Bild-Eingaben nach Markdown; visuelle Seitenbeschreibung via ColPali-Embeddings.

Track D.2 — Frontend-Erweiterungen fügt betreiberseitige Dashboards und Untersuchungsoberflächen direkt im Codex-Portal hinzu:

- **Kestra Pipeline Builder** – YAML-basierte Workflow- Orchestrierung mit integriertem Ausführungsmonitor und Trigger-Management-UI.
- **Dynamic Forms** – schemagesteuerte Dateneingabe-Formulare, die automatisch aus Kestra-Flow-Eingabedefinitionen generiert werden.
- **Charts & Pivot** – Chart.js-Visualisierung kombiniert mit PivotTable.js-Pivot-Analyse, gespeist durch Trino-Preset-Queries.
- **Federated Search** – OpenSearch-BM25 + Fuzzy-Suche mit Keyword-Fallback über alle indizierten Catalog-Assets.
- **Timeline** – vis-timeline.js-Ereignisansicht, die Marquez- Pipeline-Runs, lakeFS-Commits und Drift-Ereignisse in einer einheitlichen temporalen Darstellung zusammenführt.
- **Link Analysis** – Cytoscape.js-Netzwerkgraph aus Neo4j-Cypher-Queries für Beziehungs- und Provenienz-Untersuchungen.

Track D.3 — BI, Suche und Compliance-Postur bringt Business-Intelligence und Laufzeit-Compliance-Monitoring:

- **Apache Superset BI** – Dashboard-Einbettung via Guest-Tokens mit automatischer Trino-Datenquellen-Registrierung; funktionales Äquivalent zu Palantir Quiver.
- **OpenSearch Unified Index** – BM25 + Fuzzy-Volltextindex mit Bulk-Ingestion aus Marquez und lakeFS sowie Graceful-Degradation bei Cluster-Ausfall.
- **Falco + OpenSCAP Compliance Posture** – Laufzeit- Bedrohungserkennung via Falco und CIS-Benchmark-Scanning via OpenSCAP; funktionales Äquivalent zu Palantir Apollo.

Track D.4 — Workshop, Zeitreihen, Notes und Geospatial umfasst Low-Code-Anwendungsentwicklung und domänenspezifische Datensichten:

- **Budibase Low-Code Workshop** – Apache-2.0-Drag-and-Drop- Anwendungsbuilder für interne Werkzeuge; funktionales Äquivalent zu Palantir Workshop.
- **TimescaleDB Time Series Catalog** – Hypertable-Asset-Catalog mit Chart.js-Rendering und PostgREST-API-Exposition.
- **HedgeDoc Collaborative Notes** – AGPL-Echtzeit-Markdown- Kollaborationsumgebung; funktionales Äquivalent zu Palantir Notepad.
- **PostGIS + Leaflet Geospatial** – GeoJSON-API mit Point-in-Polygon-Queries und Leaflet-basierten interaktiven Karten; funktionales Äquivalent zu Palantir Geo.

Track D.5 — Investigation stellt eine strukturierte Fallverwaltungsschicht über alle Codex-Oberflächen bereit:

- **Dossier / Case File** – Redis-gestützter Untersuchungscontainer, mit dem Betreiber Beweismittel aus beliebigen der sechs Untersuchungsmodule in einen benannten, persistenten Falldatensatz heften können; funktionales Äquivalent zu Palantir Dossier.

MoE Codex deckt jetzt 92 % der Palantir-Foundry / Gotham / AIP / Apollo- Feature-Oberfläche ab (15 Module vollständig abgedeckt, 31 teilweise, 2 strukturell nicht erreichbar: mobile / taktische Edge-Hardware und der Apollo-Constraint-Solver). Die Abdeckungsdetails: Foundry 22 / 24, Gotham 8 / 9, AIP 13 / 13, Apollo 3 / 4. Betreiber, die ausschließlich das Kern-Multi-Modell-Gateway benötigen, deployen MoE Sovereign allein; die Codex-Erweiterung ist explizit opt-in und als separates Apache-2.0-Repository veröffentlicht (`github.com/h3rb3rn/moe-codex`).

Exkurs: Regulierungskonformität von MoE Codex unter EU-Recht

Zur Etablierung eines auditsicheren Compliance-Rahmens für zusammengesetzte KI-Systeme (*Compound AI Systems*) setzt *MoE Codex* europäische Regulierungsvorschriften direkt in technische Kontrollmechanismen um:

1. **DSGVO (Art. 32 & 35):** Gemäß Art. 32 (Sicherheit der Verarbeitung) und Art. 35 (Datenschutz-Folgenabschätzung) müssen Betreiber den Lebensweg verarbeiteter Daten lückenlos nachweisen. MoE Codex löst dies durch das Zusammenspiel von Git-artiger Versionierung (*lakeFS*) und automatisiertem Lineage-Tracking (*Marquez/OpenLineage*). Sollte ein Wissens-Bündel fehlerhafte oder unberechtigte personenbezogene Daten enthalten, ermöglicht das System einen revisionssicheren Rollback des Neo4j-Graphen auf einen sauberen historischen Commit.
2. **EU KI-Verordnung (Hochrisiko-Systeme):** Für sensible KI-Anwendungen fordern Art. 9 (Risikomanagementsystem) und Art. 10 (Daten-Governance) fortlaufende Protokollierung und deterministische Risikoprüfung. MoE Codex leitet alle Anfragen über den *Open Policy Agent (OPA)* und wertet deklarative Rego-Policen in Echtzeit aus. Eingaben in sicherheitskritischen Bereichen (z. B. Medizin, Recht) werden automatisch zur parakonsistenten Wahrheitsprüfung eskaliert.
3. **NIS2 & BSI IT-Grundschutz:** Um die strengen Sicherheits- und Überwachungspflichten für Kritische Infrastrukturen (KritIS) zu erfüllen, integriert MoE Codex *Falco* zur Laufzeit-Anomalieerkennung auf Container-Ebene sowie *OpenSCAP* für automatisierte Compliance-Scans des lokalen GPU-Clusters.

5.8 Laufzeit-Featurekontrolle: Starfleet-Toggles und Mission Context

Zwei leichtgewichtige Mechanismen ermöglichen Betreibern eine Laufzeitkontrolle des Systemverhaltens ohne Container-Neustart.

Starfleet Feature Toggles. Ein Redis-gestützter Feature-Flag-Store (`starfleet_config.py`) aktiviert oder deaktiviert benannte Fähigkeiten zur Laufzeit. Jede API-Route, die von einem Toggle abhängt, prüft dessen Zustand beim Request; der Fallback ist immer eine sichere, degradierte Antwort statt eines Fehlers. Dies ermöglicht Betreibern in regulierten Umgebungen, experimentelle Features während Audits ohne Downtime abzuschalten.

Mission Context (`/api/mission-context`). Ein persistentes Schlüssel-Wert-Dokument in Valkey, das als zusätzlicher Systemkontext in jede Pipeline-Anfrage injiziert wird. Anders als die sitzungsspezifische History-Komprimierung (Abschnitt 7) ist Mission Context deployment-weit und überlebt Sitzungswechsel. Typischer Einsatz: Eine Organisation injiziert ihr Fachvokabular, interne Namenskonventionen und Compliance-Anforderungen einmalig; alle nachfolgenden Pipeline-Anfragen profitieren davon, ohne den Kontext neu einzuführen. Mission Context adressiert auch teilweise die kontextbezogene Speicherlimitation aus Abschnitt 14: persistentes organisatorisches Wissen konkurriert nicht mit dem Token-Budget des Planers.

5.9 Skalen- und Performance-Eckwerte

Die tatsächliche Systemlast hängt stark von den eingesetzten Modellen, GPU-Klassen und Cache-Hit-Raten ab; absolute Benchmark-Zahlen wären wenig aussagekräftig, solange wir keine standardisierte Suite anbieten. Stattdessen nennen wir *Bauformen*: der Orchestrator-Container selbst hat einen Fußabdruck von etwa 200–500 MiB RAM in Dauerbetrieb; jeder LangGraph-Checkpoint kostet zwischen wenigen Kilobyte und einem niedrigen Megabyte; die 4-Schichten-Cache-Hierarchie spart je nach Workload 40–80 % der LLM-Aufrufe ein (siehe Abschnitt 7). Dieser Bereich basiert auf Produktionsbeobachtungen bei konversationellen Workloads mit moderater Wiederholungsrate; er wird bei vollständig neuartigen Anfragen deutlich niedriger und bei repetitiven FAQ-Workloads höher ausfallen. Eine systematische Sensitivitätsanalyse über verschiedene TTL-Konfigurationen und Workload-Typen ist geplant, aber noch nicht verfügbar. Konkrete Zahlen für spezifische Setups finden sich im Appendix und werden durch das Repository laufend aktualisiert; wir verpflichten uns, keine Zahlen zu kommunizieren, die wir nicht reproduzierbar belegen können.

6 Template-basiertes Routing

Das Routing ist das Herzstück des Projekts und die wichtigste konzeptionelle Abweichung gegenüber der MoE-Literatur. Dieses Kapitel erklärt, warum wir ein gelerntes Routing durch explizite, versionierte Templates ersetzen, wie die Auswahl im Detail abläuft und wie heterogene GPU-Hardware in die Entscheidung einfließt.

6.1 Das Problem mit gelernten Routern

Architektonische Präzision: Planner vs. Template-Mapping. Der Planner-Node führt einen LLM-Aufruf durch — er ist damit probabilistisch und stochastisch. Die Ausführungsebene (das statische Experten-Mapping im Template) sowie die Werkzeugebene (MCP-AST-Evaluator) sind hingegen strikt deterministisch: Für ein gegebenes Template und einen Request ist die Routing-Entscheidung vollständig reproduzierbar, unabhängig von Laufzeit oder Modelltemperatur.

Ein gelernter Router lernt eine Abbildung von Eingaberepräsentation auf eine Verteilung über Experten-Modelle. Das funktioniert in der Praxis erstaunlich gut für Genauigkeit, wirft aber vier operative Probleme auf, die in regulierten Domänen schwer wiegen:

1. **Intransparenz**: Warum wurde Experte X statt Y gewählt? Die Antwort ist eine Aktivierung in einem Transformer-Zwischenlayer und keine menschenlesbare Begründung.

2. **Verhaltensdrift**: Eine Aktualisierung des Router-Gewichts kann bisher funktionierende Prompts in unverhergesehene Richtungen lenken, ohne dass ein dokumentierter Release-Schritt stattfindet.

3. **Kaltstart-Latenz**: Router-Entscheidungen sind modellbasiert und kosten Inferenzzeit auch dann, wenn die Antwort von einem Cache geliefert werden könnte.

4. **Fehlende Auditierbarkeit**: Für eine Datenschutz-Folgeabschätzung oder ein Security-Review ist nicht rekonstruierbar, welches Modell welche Art von Daten sehen durfte.

6.2 Template-basiertes Routing

MoE Sovereign speichert die Routing-Entscheidung stattdessen in expliziten *Expert-Templates*, die im Admin-UI gepflegt werden und als JSON in der `EXPERT_TEMPLATES`-Umgebungsvariable persistiert sind. Ein Template enthält pro Expertenkategorie (etwa `code_reviewer`, `legal_advisor`, `medical_consult`, `math`, ...) eine Liste von Modellen mit je einem *Rollen-Tag*:

- `primary` – wird immer ausgeführt und hat Tier 1.
- `fallback` – wird nur ausgeführt, wenn der `primary` die Konfidenzschwelle unterschreitet (Tier 2).
- `always` – wird immer und zusätzlich ausgeführt, unabhängig von der Konfidenz (zum Beispiel ein `judge`-Modell zur Validierung).

Die Tier-Grenze wird durch `EXPERT_TIER_BOUNDARY_B = 20` (Milliarden Parameter) definiert: Modelle unterhalb dieser Grenze sind Tier 1, Modelle darüber Tier 2. Jeder Tier-1-Experte gibt ein strukturiertes Feld `CONFIDENCE: high|medium|low` aus, das per Regex `r'CONFIDENCE:\s*(high|medium|low)'` extrahiert wird. Meldet *mindestens ein* Tier-1-Experte `high`, wird die Tier-2-Eskalation übersprungen. Nur wenn *alle* Tier-1-Experten `medium` oder `low` zurückgeben, werden die größeren Tier-2-Modelle (>20 B) aktiviert.

Triviale Low-Confidence-Rettung (v2.7). Anfragen, die der Komplexitätsschätzer als *trivial* einstuft, werden kostenseitig auf einen einzigen Tier-1-Experten beschränkt (`force_tier1`). Ab v2.7 greift ein kontrollierter Rettungsmechanismus, wenn die Tier-1-Antwort tatsächlich schwach ist: Bei `low`-Confidence oder leerem Ergebnis wird genau ein Tier-2-Modell aktiviert. Eine `medium`-Antwort behält die Kostenersparnis. Das Verhalten wird über `TRIVIAL_LOW_CONF_RESCUE_ENABLED` (Standard: `true`) gesteuert und im Prometheus-Counter `moe_tier_escalation_total{category, decision}` mit `t1_cost_kept` (Ersparnis behalten) oder `t2_escalated` (Rettung ausgelöst) erfasst.

Benutzerdefinierte Experten-Kategorien (v2.7). User-Templates können beliebige Experten-Kategorien jenseits der globalen `EXPERTS`-Registry definieren (z. B. `mail_classify`, `devops_sre`, `tool_agent`). Vor v2.7 prüfte `_sanitize_plan()` die Planer-Ausgabe nur gegen die globale Registry und degradierte unbekannte Kategorien still zu `general`. Der Fix übergibt den Kategoriensatz des aktiven Templates als zusätzliche Validmenge, sodass benutzerdefinierte Experten erreichbar sind.

OpenAI-Tool-Calling-Passthrough (v2.7). Enthält eine Anfrage ein OpenAI-`tools`-Array oder Nachrichten mit `role: "tool"`, umgeht das Gateway die gesamte Planer→Experten→Merger-Pipeline und leitet direkt an das dedizierte `tool_expert`-Modell des Templates weiter (bzw. fällt auf das `judge`-Modell zurück, falls kein `tool_expert` konfiguriert ist). Das gewählte Modell (z. B. `hermes3:8b` oder `qwen3.6:35b`) übernimmt Function-Calling nativ und liefert `tool_calls` oder den finalen Text zurück. Folgeanfragen mit `role: "tool"` senden das `tools`-Array nicht weiter, um Endlosschleifen zu vermeiden. Dies ermöglicht Hermes-MCP-Integration, Open-WebUI-Function-Calling und beliebige OpenAI-kompatible Agenten-Frameworks ohne Pipeline-Overhead.

Die Funktion `_resolve_user_experts()` in `main.py` löst die Nutzerberechtigungen gegen das Template auf und liefert eine `{category → [model_config, ...]}`-Abbildung. Sie ist deterministisch, referentiell transparent und in der Testsuite mit explizit konstruierten Templates abgedeckt (siehe Abschnitt 14).

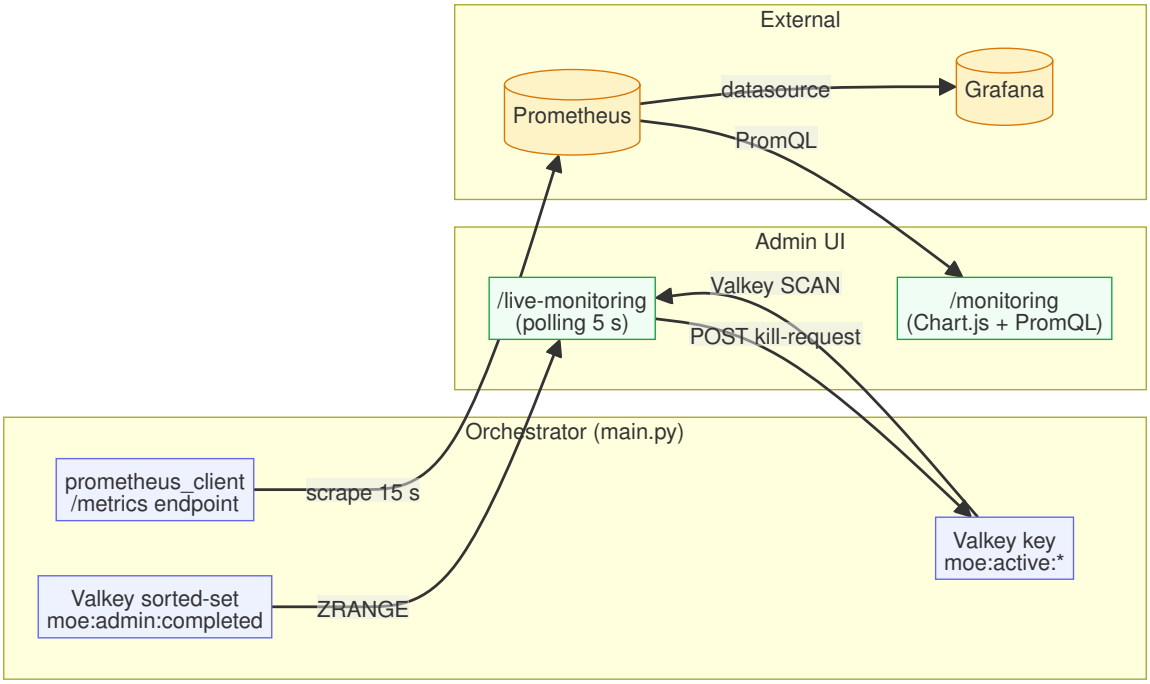


Abbildung 4: Die Observability-Architektur spiegelt denselben Design-Ansatz: Systemmetriken und Live-Monitoring werden als zwei komplementäre Sichten auf dieselbe Datenbasis behandelt – keine Abstraktion, die die Herkunft verschleiert.

6.3 Warm/Cold-Scheduling auf heterogener GPU-Hardware

Ein Template allein sagt nur, *welches* Modell aufgerufen werden soll, nicht *auf welcher Hardware*. In einem realistischen Setup stehen mehrere Inference-Server zur Verfügung (beispielsweise ein RTX-Server, ein Tesla-Server mit mehreren GPUs, ein kleiner Edge-Node). Die Funktion `_select_node(model, allowed_endpoints)` in `main.py` wählt pro Call einen passenden Node auf Basis von zwei Signalen:

1. **Warm/Cold-Erkennung**: Der Orchestrator pollt periodisch `/api/ps` auf jedem Ollama-Endpunkt und speichert die geladenen Modelle in `_ps_cache` (TTL 5 Sekunden). Ein Node, auf dem das gewünschte Modell bereits im VRAM liegt, wird bevorzugt – das eliminiert Kaltstart-Latenzen im Sekunden- bis Dutzend-Sekunden-Bereich.

2. **Load-Scoring**: Für Nodes, auf denen das Modell nicht warm vorliegt, wird ein einfacher Score running_models/gpu_count berechnet. Der Node mit dem niedrigsten Score gewinnt; Gleichstand wird durch eine deterministische Tie-Breaker-Regel entschieden (Reihenfolge in der `INFERENCE_SERVERS`-Liste).

Die Funktion ist bewusst einfach: wir haben in frühen Iterationen komplexere Scoring-Funktionen (VRAM-Restkapazität, historische Antwortzeit, Request-Queue-Länge) ausprobiert und sie wieder zurückgebaut. Der Unterschied zur simplen Formel war marginal, die

Fehleranfälligkeit deutlich höher. *So wenig Scheduler-Magie wie nötig* ist Teil der Projektphilosophie.

Lessons Learned: Der erste Scheduler war zu clever

Unser erster Scheduler berücksichtigte VRAM-Restkapazität, historische Antwortzeit, Fehlerrate der letzten fünf Minuten und den Zustand der Ollama-Request-Queue. Ergebnis: bei einem kurzen Netzwerk-Flap wurde ein Tesla-Server fälschlich als „überlastet“ markiert und alle Requests wanderten auf einen RTX-Server, der daraufhin tatsächlich überlastet war. Die Rückkehr zu einem simplen Formel-basierten Scoring mit einem 5-Sekunden-Cache hat das Problem in einer Zeile Code gelöst, die wir seitdem nicht mehr angefasst haben.

6.4 Expert-Performance-Score als vierte Cache-Schicht

Eine ergänzende Komponente ist der *Expert-Performance-Score*, der pro Paar $(model, category)$ in Valkey geführt wird. Jeder positive Feedback-Event eines Nutzers und jeder `<SYNTHESIS_INSIGHT>`-Hit im Merger-Node inkrementiert die Erfolgs-Zähler; jede negative Rückmeldung den Gegenzähler. Die Funktion `_get_expert_score()` liefert einen Laplace-geglätteten Wert $(M + 1)/(N + 2)$, wobei N die Gesamtzahl der beobachteten Ausführungen und M die Anzahl der positiven Beobachtungen ist. Tier 2-Modelle werden nur dann ausgeführt, wenn der Score des Tier 1-Modells unter einen konfigurierbaren Schwellwert (Default 0.5) fällt – ein einfacher, nachvollziehbarer Gating-Mechanismus.

Der Score ist keine Router-Ersatzfunktion, sondern eine Prioritätsannotation. Welche Modelle in Frage kommen, entscheidet das Template. Welche davon *zuerst* und *immer* zum Einsatz kommen, entscheidet der Score. Die Trennung ist wesentlich: der versionierte, auditierbare Teil der Routing-Entscheidung bleibt unberührt.

6.5 VRAM-bewusste Knotenauswahl

Wenn das Endpoint-Feld in einem Template leer ist (Floating-Modus), muss der Scheduler einen Knoten aus dem gesamten Inferenz-Cluster auswählen. Eine naive Round-Robin- oder Lowest-Load-Strategie verursacht auf heterogener Hardware einen kritischen Fehler: ein 70B-Modell (ca. 40 GB VRAM) geroutet auf einen Tesla-M10-Knoten (8 GB pro GPU) fällt stillschweigend auf CPU zurück und produziert 2–3 statt 15 Token/Sekunde.

Die Lösung ist ein konfigurierbares `vram_gb`-Feld pro Inferenz-Server, einstellbar im Admin-UI:

Knoten	VRAM	Faktor	Max. Modell
N04-RTX (5× RTX 3060)	55 GB	1.0	70B
N07-GT (1× GTX 1060)	6 GB	—	*offline seit 2026-06-02 (defekte GPU)*
N09-M60 (2× Tesla M60)	16 GB	0.9	14B
N06/N11-M10 (4× Tesla M10)	8 GB	0.8	8B

Die Funktion `_estimate_model_vram_gb()` ermittelt den Parameterbedarf aus dem Modellnamen (z.B. `llama3.3:70b` → 70) und wendet die Q4_K_M-Schätzformel an: VRAM $\approx 0.55 \times \text{params\_B} + 1.5$ GB. Knoten deren `vram_gb` unter dem Schätzwert liegt, werden *vor* der Warm/Cold/Load-Selektion ausgeschlossen – ein harter Filter statt einer weichen Warnung.

6.6 Constraint-Driven Engineering: das Apollo-11-Prinzip

Die vierschichtige Cache-Hierarchie, das deterministische Experten-Routing und die token-optimierten Prompts sind keine akademischen Designentscheidungen, sondern direkte Antworten auf eine harte Ressourcenbeschränkung. Die primäre Entwicklungsumgebung bestand aus Tesla-M10-GPUs (je 8 GB VRAM, DDR3, PCIe 2.0) mit Inferenzlatenzen von 40–120 Sekunden pro komplexem Request. Jede Millisekunde, die durch Caching eingespart wurde, entsprach auf dieser Hardware mehreren Sekunden Echtzeit – ein erzwungener Selektionsdruck, der Designs überlebten ließ, die auf moderner H100-Hardware schlicht nicht entstehen würden.

Das Apollo-11-Navigationssystem (AGC, 4 KB RAM, 72 KB ROM) war nicht *trotz* seiner Beschränkungen präzise, sondern *wegen* ihnen: jede Routine war auf das Notwendige komprimiert, kein Cycle wurde verschwendet. MoE Sovereign folgt derselben Logik. Systeme, die auf H100-Clustern mit Brute-Force-Context und ohne Caching arbeiten, bezahlen mit Energie, Latenz und Kosten, was MoE Sovereign durch Architektur amortisiert hat.

Constraint-Driven Engineering

Ressourcenbeschränkungen sind kein Hindernis für Systemqualität, sondern deren härtester Prüfstand. Was unter 8 GB VRAM und 40 s Inferenzlatenz deterministisch korrekt läuft, wird auf moderner Hardware durch Effizienz dominieren, nicht trotz ihr durch Brute Force. Das ist etabliertes *Constraint-Driven Engineering* – der AGC-Vergleich ist illustrativ, kein Anspruch auf Neuheit.

7 Mehrschichtige Cache-Hierarchie

Ein LLM-Aufruf kostet Zeit, Strom und gegebenenfalls Geld. Eine Orchestrierungsschicht, die Determinismus und Effizienz ernst nimmt, muss diese Kosten an jeder Stelle vermeiden, an der die Antwort vorhersehbar ist. MoE Sovereign verwendet vier unabhängige Cache-Schichten, jede mit einem anderen Key-Schema, einer anderen Invalidierungs-Policy und einem anderen Zielverhalten.

Abbildung 5: Die vierschichtige Cache-Hierarchie: L1 semantisch, L2 Plan, L3 Graph, L4 Performance-Score. Jede Schicht hat einen distinkten Key-Raum und eine eigene TTL.

7.1 L1: Semantischer Cache (ChromaDB)

Die oberste Schicht ist ein semantischer Cache auf Basis von ChromaDB [3]. Bei jeder eingehenden Nutzeranfrage wird ein Embedding-Vektor berechnet und gegen die Kollektion `moe_fact_cache` mit Cosine-Distanz verglichen. Liegt der nächste Nachbar unter einem Schwellwert (Default 0.15), wird die zuvor gespeicherte Antwort direkt zurückgegeben – ohne Planner, ohne Expert-Worker, ohne Judge. Die Einsparung pro Cache-Hit ist drastisch: statt einer vollständigen Pipeline mit typischen drei bis fünf LLM-Aufrufen wird genau ein Embedding-Aufruf fällig.

Der L1-Cache wird gezielt vorsichtig invalidiert: Einträge bekommen keinen TTL, sondern werden beim Eingang negativer Nutzer-Feedbacks auf `flagged=true` gesetzt und beim nächsten Treffer übersprungen. Administratoren können die Kollektion mit einem einzigen Befehl zurücksetzen (`mkdocs/docs/PRIVACY.md` enthält die exakte Prozedur). Die Policy ist explizit nicht „jede Cache-Entscheidung ist für immer“ – sie ist „eine gepeerreviewte Cache-Entscheidung bleibt, bis jemand Widerspruch einlegt“. Das ist eine bewusste Trade-off-Entscheidung zugunsten der Nutzbarkeit.

7.2 L2: Plan-Cache (Valkey)

Die zweite Schicht speichert die *Planner-Entscheidung* einer Anfrage. Der Planner ist derjenige Knoten, der aus einer Nutzerfrage die Liste der zu aktivierenden Expertenkategorien ableitet („Diese Frage ist gleichzeitig juristisch und medizinisch, aktiviere beide Experten“). Der Key ist der SHA-256-Hash über die normalisierte Query-Repräsentation; der Wert ist die serialisierte Planner-Antwort. Ein Hit im L2-Cache überspringt den Planner-LLM-Aufruf, behält aber die expert_worker- und merger-Stages bei. TTL: 30 Minuten.

7.3 L3: Graph-Kontext-Cache (Valkey)

Die dritte Schicht ist der *Graph-Kontext-Cache*: das Ergebnis einer Neo4j-GraphRAG-Abfrage wird unter einem Hash-Key abgelegt und binnen der TTL wiederverwendet. Ein Hit spart die Round-Trip-Zeit zu Neo4j und die optionale procedural-traversal-Logik. TTL: 60 Minuten.

Die bewusste Trennung zwischen L2 (Planner) und L3 (GraphRAG) ist wichtig: zwei inhaltlich unterschiedliche Anfragen können denselben Planner-Plan, aber unterschiedliche Graph-Kontexte produzieren, oder umgekehrt. Ein einzelner monolithischer Cache hätte keines von beiden effizient bedient.

7.4 L4: Expert-Performance-Scores (Valkey)

Die unterste Schicht ist formal kein Cache im engeren Sinne, sondern ein *persistentes Beobachtungs-Register*: pro $(\text{model}, \text{category})$ wird ein Valkey-Hash mit den Feldern `total` (Gesamt-Ausführungen) und `positive` (positive Feedbacks) geführt. Die Werte fließen in die Scoring-Funktion aus Abschnitt 6 ein und beeinflussen das Tier-2- Gating. Der Raum ist klein (weniger als tausend Keys pro Jahr) und wächst linear mit der Anzahl der Expertenkategorien und Modelle. Eine eigene Invalidierung ist nicht nötig; die Laplace-Glättung stellt sicher, dass auch nach langen Beobachtungszeiträumen neue Modelle noch eine realistische Chance bekommen.

7.5 Kombinationslogik: Fall-Through

Alle vier Schichten sind strikt als Fall-Through organisiert: ein Hit in L1 beendet die Verarbeitung unmittelbar; bei Miss wird L2 konsultiert; bei weiterem Miss L3; bei endgültigem Miss greift L4 nur noch beratend für die Tier-2-Entscheidung. Das Fall-Through-Muster ist kein Zufall – es erlaubt, jede Schicht isoliert zu deaktivieren, ohne die anderen zu brechen. Bei Debugging-Sessions haben wir mehrfach eine einzelne Schicht temporär abgeschaltet (`CACHE_DISABLE_L1=1`) und die gemessenen Effekte auf Latenz, Fehlerrate und LLM-Call-Anzahl direkt beobachten können.

7.6 Augmented Tool Path: Cache- und GraphRAG-Erweiterung für agentische Clients

Agentische Coding-Werkzeuge – Claude Code CLI, OpenCode und vergleichbare Function-Calling-Clients – senden praktisch jede Anfrage mit einem `tools`-Array oder mit `tool_result`-Turns. Dafür existiert in MoE Sovereign ein dedizierter Tool-Calling-Fast-Path, der die gesamte MoE-Pipeline – Planner, Experten, Judge, und damit auch L1–L4 – bewusst umgeht und die Anfrage direkt an ein einzelnes Tool-Modell durchreicht. Dieser Kurzschluss ist notwendig, weil zuverlässiges Function-Calling ein einzelnes, konsistentes Modell voraussetzt; er hat aber zur Folge, dass agentische Sitzungen von der gesamten Cache- und GraphRAG-Infrastruktur nichts profitieren – jede Anfrage kostet volle Inferenz, unabhängig davon, wie oft dieselbe Frage bereits beantwortet wurde.

Der *Augmented Tool Path* schließt diese Lücke, ohne den Fast-Path selbst zu verändern: eine opt-in Anreicherungsschicht sitzt zwischen dem Tool-Handler und dem Tool-Modell und verhält sich für den Client vollständig transparent – Streaming-Events, `stop_reason`-Semantik und Tool-Use-Blöcke bleiben bitgenau erhalten.

Getrennter Cache-Namensraum. Statt L1 (`moe_fact_cache`) mitzubenutzen, schreibt und liest der Augmented Tool Path in eine eigene Kollektion (`moe_agent_cache`) sowie einen eigenen Valkey-Präfix (`moe:agent:qcache:*`). Jeder Eintrag ist mit einem Scope-Schlüssel `sha256(user_id | workspace)` versehen – Wissen aus Projekt A kann so niemals in Projekt B durchsickern, auch wenn beide vom selben Nutzer stammen. Bedient werden ausschließlich *informationale* Anfragen des ersten Turns einer Sitzung (heuristisch erkannt an Frageform ohne Mutations-Verb wie „fix“/„implement“); Werkzeugaufruf- Entscheidungen selbst werden nie aus dem Cache bedient, da sich der Umgebungszustand zwischen Sitzungen unterscheidet.

Vertrauensbildung statt sofortigem Caching. Eine frische Antwort wird zunächst mit Confidence 0.6 gespeichert – bewusst unterhalb der Serve-Schwelle von 0.85. Ein asynchroner Judge-Aufruf bewertet die Antwort im Hintergrund; erst ein hoher Score hebt die Confidence auf 0.9 an und schaltet den Eintrag für zukünftige Cache-Treffer frei. Dieses Zwei-Stufen-Muster spiegelt dieselbe Vorsicht wider, die bereits das Knowledge-Bypass- Gating in L1 auszeichnet (Confidence- und Freshness-Schwellwert, `flagged`-Markierung bei schlechtem Feedback) – nur dass hier der Judge selbst als nachgelagerte Qualitätskontrolle fungiert, statt sich auf Nutzer-Feedback zu verlassen.

GraphRAG-Injektion pro Sitzung. Zusätzlich zum Cache fragt der Augmented Tool Path beim ersten Turn einer Sitzung den Wissensgraphen ab (Tenant-gescoped, siehe Abschnitt 8) und injiziert das Ergebnis in den System-Prompt des Tool-Modells. Das Ergebnis wird unter einem sitzungsgebundenen Valkey-Key zwischengespeichert, sodass nachfolgende Turns desselben Werkzeug-Loops den Graphen nicht erneut abfragen müssen – eine Neo4j-Abfrage pro Sitzung statt pro Turn.

Wissenswachstum aus agentischen Sitzungen. Endet eine Sitzung mit einer sauberen Textantwort (`end_turn` ohne Tool-Use, kein Stream- Fehler), wird das Frage-Antwort-Paar über denselben Kafka-Ingest-Topic publiziert, den auch die interaktive Pipeline nutzt. Der Wissensgraph wächst damit nicht nur aus Open-WebUI-Konversationen, sondern auch aus Coding-Sitzungen – inklusive Provenienz und Trust-Decay wie in Abschnitt 8 beschrieben.

Alle Latenzkosten sind hart begrenzt und bewusst konservativ budgetiert: der Cache-Lookup ist auf 300 ms zeitboxiert, die GraphRAG-Abfrage auf 2 s, und sämtliche Schreiboperationen laufen `asyncio.create_task` fire-and-forget – die vom Client wahrgenommene Antwortzeit ändert sich nur im Erfolgsfall eines Cache-Treffers (dann drastisch schneller), niemals durch

einen Fehlschlag der Anreicherung. Ist ein Flag nicht aktiviert, ist das Verhalten bit-identisch zum reinen Fast-Path.

Innovation: Separierung der Caches nach Intent

Die meisten Cache-Designs für LLM-Systeme kennen genau einen Cache – Request → Response. Das ist entweder zu grob (der gesamte Planner und GraphRAG werden pauschal umgangen) oder zu fein (jeder LLM-Aufruf wird einzeln gecacht, was bei mehrstufigen Pipelines schwer zu invalidieren ist). Die Vier-Schichten-Aufteilung folgt der *Intent-Ebene*: L1 = semantische Ähnlichkeit, L2 = Plan-Äquivalenz, L3 = Graph-Kontext-Äquivalenz, L4 = historische Zuverlässigkeit. Jede Schicht beantwortet eine andere Frage und ist damit unabhängig wartbar.

8 GraphRAG und Graph-basierte Wissensakkumulation

Retrieval-Augmented Generation [27] ist mittlerweile Stand der Technik, wenn es darum geht, Fakten aus einer eigenen Wissensbasis in LLM-Antworten einzubringen. Das Standard-RAG-Rezept – Volltexte in Chunks zerlegen, Embeddings berechnen, Top-k mit Cosinus-Ähnlichkeit abrufen und an den Prompt anhängen – scheitert allerdings an zwei Problemen, die für unseren Anwendungsfall zentral sind: *Kontextfenster* und *Cross-Kontamination*.

8.1 Warum ein Graph statt einer Vektor-Datenbank allein

Ein Vektor-Store ist gut darin, ähnliche Passagen zu finden, aber schlecht darin, Beziehungen zwischen Entitäten zu repräsentieren. Ein Wissensgraph – in unserem Fall Neo4j [5] – ist umgekehrt stark in Beziehungen und schwach bei semantischer Ähnlichkeit. MoE Sovereign kombiniert beide: ChromaDB als semantische Suche auf Rohtext und Neo4j als strukturiertes Weltwissen mit Entitäten, Relationen und Ontologie-Typen. Die Idee ist eng verwandt mit Microsoft GraphRAG [25], weicht aber in zwei Aspekten ab, die wir für zentral halten: *kategoriespezifische Entity-Type-Filter* und ein *Graph-basierter Akkumulationsmechanismus*.

8.2 Kategoriespezifische Entity-Type-Filter

Die Datei `graph_rag/manager.py` enthält eine Abbildung `_CATEGORY_ENTITY_TYPES`, die jeder Expertenkategorie einen zulässigen Satz an Neo4j-Labels zuordnet. Konkret: der Legal-Advisor sieht nur Entitäten des Typs `Law`, `Right`, `Legal_Concept` und `Organization`; der Medical-Consult sieht nur `Drug`, `Disease`, `Symptom`, `Treatment`; der Technical-Support sieht nur `Software`, `Protocol`, `Hardware`, `Error_Code`. Jede GraphRAG-Abfrage wird mit dem zulässigen Label-Set der aktuellen Expertenkategorie gefiltert.

Auf Knotenebene hat jede Entität die Struktur `(:Entity {name: Ïbuprofen", type: "Drug", domain: "medical_consult"})`. Die Cypher-Abfrage enthält den Filter `WHERE e.type IN $allowed_types`, wobei `allowed_types` aus den Plan-Kategorien über `_CATEGORY_ENTITY_TYPES` aufgelöst wird.

Warum ist das wichtig? Ohne Filter ergibt eine Anfrage an den Medical-Consult unter Umständen auch technische Entitäten, wenn die Embeddings der Frage zufällig beide Domänen berühren. Das Modell kann dann in seiner Antwort medizinische und technische Aussagen vermischen – ein Ausfallmodus, den wir in frühen Versionen beobachtet und dessen Fix wir hier als Innovation festhalten.

Innovation: Cross-Contamination Prevention

Die Entity-Type-Filter verhindern systematisch, dass ein Experten- Modell Kontext aus einer fremden Domäne zu sehen bekommt. Das ist nicht nur eine Qualitätsverbesserung, sondern eine Sicherheitseigenschaft: in einer Mehrmandantenumgebung mit unterschiedlich akkreditierten Nutzergruppen kann die Filter-Tabelle als Policy-Durchsetzungspunkt dienen.

Formales Ontologie-Schema. Tabelle 2 spezifiziert die Domänen-zu-Typ-Abbildung, die zur Query-Zeit erzwungen wird. Nicht aufgeführte Kategorien erlauben alle Entity-Typen (`general`, `reasoning`, `precision_tools`).

Tabelle 2: Domänengefilterte Entity-Typen: erlaubte Neo4j-Knotentypen je Expertenkategorie. Nicht aufgeführte Kategorien sind unbeschränkt.

Expertenkategorie	Erlaubte Entity-Typen
`medical_consult`	Drug, Disease, Symptom, Treatment, Anatomy, Medical_Concept, Drug_Class
`legal_advisor`	Law, Right, Legal_Concept, Organization
`technical_support`	Framework, Tool, Protocol, Tech_Concept, DevOps, Architecture, Security, DataStructure, Algorithm, Pattern, Principle, Action, Location, Condition
`code_reviewer`	Framework, Tool, Tech_Concept, DataStructure, Algorithm, Pattern, Principle, Architecture, Security, Game_Concept, Data_Concept, Action, Condition
`math`	Math_Concept, Science, Concept, Algorithm, DataStructure
`translation`	Language, Concept, Narrative
`creative_writer`	Narrative, Concept
`research`	*(unbeschränkt – alle Typen)*

Erlaubte Relationstypen (domänenübergreifend): `IS_A` | `PART_OF` | `TREATS` | `CAUSES` | `REGULATES` | `USES` | `IMPLEMENTS` | `DEPENDS_ON` | `DEFINES` | `RELATED_TO` | `NECESSITATES_PRESENCE` | `ENABLES_ACTION` | `DEPENDS_ON_LOCATION`. Knoteneigenschaften: `name` (string), `type` (string), `aliases_str` (string), `tenant_id` (optionaler string), `confidence` (float $\in [0, 1]$), `version` (int), `source_model` (string).

8.3 Die Basisontologie

Der Graph wird nicht leer gestartet. Das Modul `graph_rag/ontology.py` enthält über 400 vorgepflegte Basisentitäten in mehreren Sprachen mit Aliasen, Labels und initialen Relationen. Beispiele: die wichtigsten Paragraphen des BGB und StGB, die gängigen Wirkstoffklassen, die zentralen Technologie- Frameworks und ihre Abhängigkeiten. Die Ladephase ist idempotent (`MERGE` statt `CREATE`), sodass ein erneutes Einlesen keine Duplikate produziert und Admins die Ontologie-Datei versionskontrolliert erweitern können.

8.4 Persistentes Graph-State-Tracking: Der SYNTHESIS_INSIGHT-Mechanismus

Der wichtigste Unterschied zu klassischem RAG ist, dass der Graph im laufenden Betrieb *wächst*. Der Merger-Node im Pipeline-Graph erhält eine erweiterte System-Prompt-Anweisung,

die ihn auffordert, bei neuen Mehr-Quellen-Erkenntnissen einen strukturierten JSON-Block mit dem Tag `<SYNTHESIS_INSIGHT>` zu emittieren. Ein nachgelagerter Ingest-Prozess auf dem Kafka-Topic `moe.ingest` erkennt diese Blöcke, extrahiert die Entitäten und Relationen, und schreibt sie in Neo4j – mit einer Versions-Annotation, die das emittierende Modell und einen Zeitstempel benennt.

Der Mechanismus ist bewusst konservativ: nur *neue, nicht triviale* Erkenntnisse sollen aufgenommen werden. Das Modell wird in der Anweisung explizit darauf hingewiesen, keine simplen Faktennachschläge zu emittieren. Der Ingest verifiziert zusätzlich, ob die extrahierte Information im bestehenden Graphen bereits vorliegt. Trotzdem ist der Wissensgraph keine reine Black-Box-Summary: jede Aussage ist inspizierbar und mit einer Quellen-Annotation versehen.

8.5 Contradiction Detection

Eine zweite Sicherheitsschicht ist die Widerspruchs-Erkennung. Die Datei `graph_rag/manager.py` enthält eine Tabelle `_CONTRADICTORY_PAIRS`, die zueinander ausschließende Relationen deklariert. Beispiel: wenn eine Relation (A)–[TREATS] $\rightarrow (B)$ existiert und ein neu eingeschriebenes Triple (A)–[CAUSES] $\rightarrow (B)$ ankommt, wird der Ingest markiert und durch eine Lint-Pipeline (Topic `moe.linting`) kuratiert. Widersprüche werden nicht automatisch überschrieben – sie werden *markiert* und einem menschlichen Reviewer vorgelegt.

8.6 Ontologie-Gaps als Signal

Eine dritte, subtile Eigenschaft ist das *Ontologie-Gap-Signal*: wenn der Orchestrator einen LLM-Output sieht, dessen zentrale Begriffe sich nicht in den Neo4j-Labels finden lassen, wird der Begriff als „Gap“ in einer Prometheus-Metrik (`moe_ontology_gaps_total`) gezählt. Admins können die Top-Gaps im Admin-UI sehen und entscheiden, ob die Ontologie erweitert werden sollte. Das System lernt also nicht nur Fakten, sondern auch *wo sein Wissen lückenhaft ist.*

8.7 Community-Wissens-Bundles

Ein Wissensgraph ist nur so wertvoll wie die Breite seiner Trainingsdaten. MoE Sovereign adressiert dies durch einen Export-/Import-Mechanismus, der *kollektive Intelligenz* über Deployments hinweg ermöglicht.

Export. Der API-Endpunkt `/graph/knowledge/export` extrahiert Entities, Relationen und Synthese-Knoten als JSON-LD-Bundle. Drei Schutzschichten verhindern Datenabfluss:

1. **Metadaten-Stripping:** `tenant_id`, `source_model` und Zeitstempel werden standardmäßig entfernt.
2. **Privacy Scrubber:** Regex-Muster erkennen und entfernen Entities mit Passwörtern, IP-Adressen, E-Mail-Adressen, API-Keys, Infrastruktur-Hinweisen oder Kundennamen.
3. **Sensitive Relationstypen:** Relationen wie `HAS_PASSWORD` oder `AUTHENTICATES_WITH` werden bedingungslos ausgeschlossen.

Im Produktivtest entfernte der Scrubber 97–214 Entities aus einem Graphen mit 3 150 Entities.

Import. Der Import-Endpunkt (`/graph/knowledge/import`) führt Community-Bundles mit drei Garantien in den lokalen Graphen zusammen:

- **Kein Überschreiben:** Entities werden nach Name zusammengeführt (`MERGE ON CREATE`). Bestehende werden nie modifiziert.
- **Trust-Deckelung:** Importierte Relationen werden auf einen konfigurierbaren `trust_floor` (Standard 0.5) begrenzt, sodass Community-Daten nie lokal verifizierte Fakten übertreffen.
- **Widerspruchserkennung:** Vor dem Anlegen einer Relation prüft das System `_CONTRADICTORY_PAIRS` gegen bestehende hochvertraute Triple. Widersprüchliche Imports werden übersprungen und protokolliert.

Ein Dry-Run-Modus (`/graph/knowledge/import/validate`) zeigt vorab, was importiert würde, ohne den Graphen zu verändern. Wiederholte Imports desselben Bundles sind idempotent: alle Entities melden „skipped“, es entstehen null Duplikate.

Beim Import erkannte Widersprüche werden an das Kafka-Topic `moe.linting` zur Admin-Prüfung publiziert, sodass Community-Wissen niemals stillschweigend unternehmensintern verifizierte Fakten überschreibt.

Federated Knowledge Sync

Knowledge-Bundles ermöglichen den strukturierten Austausch domänenspezifischer Wissensgraphen zwischen unabhängigen Deployments. Jede Instanz bleibt autonom und offline-fähig; geteilte Bundles reichern den lokalen Graphen an, ohne Quelldaten oder proprietäre Informationen zu übertragen.

8.8 MoE Libris: Föderierter Wissensaustausch

Der Federated Knowledge Sync aus der obigen Innovationsbox war in v1.0 dieses Papers ein konzeptioneller Entwurf. Mit *MoE Libris*[2] haben wir ihn als konkretes, deploybares System umgesetzt.

Architektur. MoE Libris ist ein eigenständiger FastAPI-Mikrodienst (der *Hub-Server*), gestützt auf PostgreSQL (relationale Metadaten, Audit-Log), Neo4j (globaler Wissensgraph) und Valkey (Rate-Limiting, Strike-Zähler). Einzelne MoE Sovereign-Installationen agieren als *souveräne Knoten*, die JSON-LD-Wissens-Bundles über die REST-API des Hubs senden und empfangen. Die Topologie ist Hub-and-Spoke: jeder Knoten verbindet sich mit genau einem Hub; der Hub initiiert keine Verbindungen zu Knoten.

Föderationsprotokoll. Die Knotenanbindung folgt einem bilateralen Handshake:

1. Ein Knotenbetreiber registriert sich beim Hub (Knoten-URL, öffentliche Metadaten, Betreiber-Kontakt).
2. Der Hub-Administrator prüft die Registrierung und nimmt sie an oder lehnt sie ab.
3. Bei Annahme stellt der Hub einen knotenspezifischen API-Key aus; der Knoten speichert ihn lokal. Alle nachfolgenden Anfragen werden über diesen Key authentifiziert.

[2] Lateinisch *liber* = sowohl “frei” als auch “Buch” – eine bewusste Doppelbedeutung, die den Open-Source-Charakter des Projekts und seinen Zweck als Wissensspeicher widerspiegelt.

Nach der Kopplung verläuft der Datenfluss wie folgt: Der Knoten schickt ein JSON-LD-Bundle an den Hub; der Hub durchläuft eine zweistufige *Pre-Audit-Pipeline* (siehe unten); Bundles, die die Vorprüfung bestehen, gelangen in eine *Auditierungswarteschlange*; ein Hub-Administrator genehmigt oder verwirft jedes Bundle explizit; genehmigte Bundles werden in den globalen Graphen eingeführt; andere gekoppelte Knoten können neues Wissen über einen Polling-Endpunkt abrufen.

Pre-Audit-Pipeline. Jedes eingehende Bundle durchläuft zwei automatisierte Validierungsstufen, bevor es die Auditierungswarteschlange erreicht:

1. **Stufe 1 – Syntaxvalidierung:** JSON-LD-Schemakonformität, Pflichtfelder, wohlgeformte Entity- und Relationsstrukturen.
2. **Stufe 2 – Heuristische PII-/Geheimnis-Erkennung:** Regex-basierte Erkennung von E-Mail-Adressen, IPv4-/IPv6-Adressen, JWT-Tokens, API-Keys und anderen Credential-Mustern. Bundles mit Treffern werden mit einem Diagnosebericht abgelehnt.

Die Pipeline ist erweiterbar konzipiert: eine geplante Stufe 3 wird LLM-gestützte Triage zur semantischen Inhaltsprüfung hinzufügen (v1.1, noch nicht implementiert).

Missbrauchsprävention. Der Hub implementiert ein graduiertes Strike-System. Jeder Richtlinienverstoß (fehlgeschlagene Vorprüfung, abgelehntes Bundle, Rate-Limit-Verletzung) fügt dem betroffenen Knoten einen Strike hinzu. Strikes sind gewichtet: Sicherheitsverstöße (Credential-Leakage, Injection-Versuche) zählen dreifach. Bei drei akkumulierten Strikes wird der Knoten ratenlimitiert; bei zehn gewichteten Strikes wird er automatisch blockiert und muss sich erneut registrieren.

Server-Discovery. Hub-Instanzen sind über ein öffentliches Git-Registry (`moe-libris-registry`) auffindbar. Jeder Betreiber kann einen Hub registrieren, indem er einen Pull-Request mit einer JSON-Metadatendatei einreicht; die CI validiert das Schema vor dem Merge. Dieser Mechanismus ist analog zu Fediverse-Instanzlisten (z. B. *instances.social* für Mastodon) und vermeidet eine zentrale Autorität über das Verzeichnis.

Outbound-Policy-Engine. Jeder souveräne Knoten steuert über eine domänenspezifische Outbound-Policy, was er teilt. Drei Modi stehen zur Verfügung: `auto` (alle qualifizierenden Bundles automatisch pushen), `manual` (vor dem Push lokale Admin-Prüfung einreihen) und `blocked` (nie an diesen Hub pushen). Zusätzliche Filter umfassen einen Mindest-Konfidenzschwellenwert und ein `verified-only`-Flag, das Exporte auf menschlich verifizierte Triples beschränkt.

Vertrauensmodell. Vom Hub importierte Triples unterliegen demselben Trust-Floor-Mechanismus wie in Abschnitt 8.7 beschrieben: Community-Daten werden auf einen konfigurierbaren Trust-Score (Standard 0.5) gedeckelt und übertreffen nie lokal verifizierte Fakten. Die bestehende Widerspruchserkennung (`_CONTRADICTORY_PAIRS`) gilt für Hub-Triples identisch wie für manuell importierte Bundles. Es findet keine automatische Vertrauenspropagation statt – jede importierte Aussage muss lokales Vertrauen durch Verifikation oder Bestätigung erwerben.

Architektonische Parallele. Der Entwurf orientiert sich explizit am Fediverse-Modell (ActivityPub / Friendica): unabhängige, selbstgehostete Instanzen föderieren freiwillig über ein standardisiertes Protokoll; keine Instanz hat Autorität über eine andere; das Netzwerk wächst durch Adoption, nicht durch zentrale Steuerung. Die Analogie ist architektonischer, nicht funktionaler Natur: MoE Libris tauscht strukturierte Wissens-Triples aus, keine Social-Media-Beiträge.

Aktuelle Limitationen. Die derzeitige Implementierung unterstützt ausschließlich eine Single-Hub-Topologie; Multi-Hub-Föderation (Mesh oder hierarchisch) ist geplant, aber noch nicht entworfen. Bundles sind nicht kryptographisch signiert; eine zukünftige Version wird Ed25519-Signaturen zur Manipulationserkennung im Transit ergänzen. Die LLM-gestützte Triage-Stufe (Stufe 3 der Pre-Audit-Pipeline) ist spezifiziert, aber noch nicht implementiert.

8.9 Corrective-RAG-Gate

Standard-RAG injiziert alle abgerufenen Dokumente in den Generierungsprompt, unabhängig davon, ob sie wirklich zur Anfrage passen. Yan et al. [18] zeigen, dass diese bedingungslose Injektion die Ausgabequalität verschlechtert, wenn das Retrieval mehrdeutig oder fehlerhaft ist. Die Autoren schlagen einen leichtgewichtigen *Retrieval-Evaluator* vor, der jedes abgerufene Dokument vor der Injektion bewertet und in die Zustände *Correct* (injizieren), *Ambiguous* (verfeinern) oder *Incorrect* (verwerfen und Web-Suche auslösen) einteilt.

MoE Sovereign adaptiert diesen Mechanismus auf die Ebene von Neo4j-Entitäten. Nachdem `query_context()` die Kandidatenmenge aus dem Graphen aufgebaut hat, bewertet `_corrective_relevance_score()` jede Entität:

$$s(e) = 0{,}75 \cdot \frac{2 \cdot h_{\text{Name}} + h_{\text{Rel}}}{3 \cdot |T|} + 0{,}25 \cdot \bar{c} \tag{3}$$

Dabei ist T die Menge der inhaltssignifikanten Anfragewörter, h_{Name} zählt Treffer im Entitätsnamen (Gewicht $2\times$, da ein direkter Namenstreffer ein starkes Signal ist), h_{Rel} zählt Treffer in den Relationstargets, und $\bar{c}$ ist der Durchschnitts-Confidence-Wert der gespeicherten Relationen. Entitäten unter einem konfigurierbaren Schwellwert τ (Standard 0,15, Env-Variable `GRAPHRAG_CORRECTIVE_THRESHOLD`) werden vor der Injektion verworfen.

Corrective RAG reduziert Kontextverschmutzung

Das Gate verhindert *Context Pollution*: Entitäten, die einen Anfragebegriff nur zufällig enthalten, semantisch jedoch völlig unverwandt sind, werden herausgefiltert – ohne zweiten LLM-Aufruf.

8.10 Context-Augmented Generation für Compliance-Domänen

Chan et al. [19] weisen empirisch nach, dass für Wissensdomänen, deren Inhalte *stabil*, *autoritativ* und *abgegrenzt* sind, das direkte Laden des vollständigen Kontexts dem retrievalbasierten RAG in Genauigkeit, Konsistenz und Latenz überlegen ist (Context-Augmented Generation, CAG).

In regulierten Industrien – dem primären Einsatzbereich von MoE Sovereign – weisen aufsichtsrechtliche Texte wie BAIT, VAIT, DORA und KRITIS genau diese Eigenschaften

auf: sie ändern sich nur im Legislativzyklus, ihr exakter Wortlaut ist maßgeblich, und der relevante Ausschnitt passt in einen einzelnen Kontextblock.

Das Modul `compliance_cag` setzt dieses Muster als Pre-Retrieval-Gate in `graph_rag_node` um. Jede Compliance-Domäne wird durch eine vom Administrator verwaltete JSON-Datei definiert, die Schlüsselwort-Trigger und einen autoritativen Textblock enthält. Erkennt das Gate bei einer Anfrage einen Keyword-Treffer, gibt es den vorgeladenen Block direkt zurück und überspringt die Neo4j-Abfrage vollständig.

Neue Compliance-Domänen werden durch Ablegen einer `.json`-Datei in `$MOE_DATA_ROOT/cag/` hinzugefügt – kein Code-Änderung, kein Neustart. Das Modul lädt das Verzeichnis alle `CAG_RELOAD_INTERVAL_S` Sekunden (Standard: 300) neu.

Tabelle 3: CAG vs. Standard-GraphRAG bei Compliance-Anfragen.

Eigenschaft	**Standard-GraphRAG**	**CAG-Layer**
Latenz	1–3 s (Neo4j + Valkey)	<1 ms (in-process)
Verlässlichkeit	Abhängig von Graphabdeckung	Deterministisch
Wortlautgenauigkeit	Paraphrasiert	Autoritativer Originaltext
Aktualisierung	Graph-Ingest (online)	JSON-Dateitausch
Audit-Nachweisbarkeit	Quellmodell-Annotation	Explizites Domänen-Label

8.11 Episodisches Gedächtnis

Tulving [28] unterscheidet drei Langzeitspeichersysteme: *semantisches* Gedächtnis (Fakten und Weltwissen), *episodisches* Gedächtnis (zeitlich situierte Vergangenheitserfahrungen) und *prozedurales* Gedächtnis (Ausführungsmuster von Fertigkeiten). Park et al. [20] operationalisieren episodisches Gedächtnis für LLM-Agenten als *Memory Streams*: zeitgestempelte Erfahrungseinträge, die nach Relevanz und Aktualität abgerufen werden. Packer et al. [21] verankern dies in einer geschichteten Architektur analog zu virtuellem OS-Speicher.

MoE Sovereign bildet Tulvings Taxonomie direkt auf seine Speicherschicht ab:

Tabelle 4: Dreistufige Gedächtnisarchitektur in MoE Sovereign.

Gedächtnistyp	**Inhalt**	**Implementierung**
Semantisch	Domänenfakten, Weltwissen	Neo4j-Wissensgraph
Episodisch	Vergangene Task-Ergebnisse + Routing	`:Episode`-Knoten
Prozedural	Aktion-Ort-Anforderungen	Prozedurale Relationen

Das Modul `episodic_memory` ergänzt die fehlende episodische Schicht. Nach jedem abgeschlossenen Merger-Response schreibt eine nicht blockierende `asyncio.create_task()`-Aufgabe einen `:Episode`-Knoten nach Neo4j mit Task-Typ, Routing-Pfad, genutzten Tools, Konfidenzschätzung, Token-Verbrauch und TTL-Ablaufdatum.

Beim Abruf bewertet `get_episode_hint()` vergangene Episoden desselben Task-Typs per Sørensen–Dice-Stringähnlichkeit gegen die aktuelle Anfrage und gibt die Top-k Episoden als `[Episode Hint]`-Block in `graph_context` zurück. Der Hinweis informiert das Judge-Modell darüber, welche Routing-Strategien in der Vergangenheit hohe Konfidenz erzielt haben – ohne die Antwort vorzuschreiben.

Alle episodischen Operationen sind *fire-and-forget*: ein Neo4j-Fehler propagiert niemals in den primären Antwortpfad.

8.12 Würdigung der angewandten Wissenschaftlichen Quellen

Tabelle 5 fasst *alle* im Quellcode von MoE Sovereign direkt umgesetzten Wissenschaftlichen Quellen zusammen — über formale Logik, Expert-Routing, Retrieval und Gedächtnisschichten hinweg. Die Spalte „Implementierung"nennt das konkrete Modul und den Mechanismus.

9 MCP-Präzisionswerkzeuge

Große Sprachmodelle halluzinieren Arithmetik zuverlässig falsch. Multiplikation fünfstelliger Zahlen, Datums-Differenzen über Monatsgrenzen, die Berechnung einer Netzmaske oder der Hash eines Strings sind Aufgaben, die jedes Taschenrechner-Programm in Millisekunden und zu 100 % korrekt löst, während ein LLM bei jeder Anfrage eine neue plausibel aussehende Zahl raten kann. Die Standardantwort der Branche – *function calling* – ist konzeptionell richtig, aber in den meisten Implementierungen zu locker: das Modell darf eine Python-Funktion aufrufen, deren Code im Prozess des Orchestrators ausgeführt wird, mit allem, was Python drumherum anbietet.

MoE Sovereign geht einen Schritt weiter: es delegiert deterministische Berechnungen an einen separaten *Model Context Protocol*-Server [6], der über eine streng whitelistete AST- basierte Expressions-Engine verfügt – und über 50 weitere eingebaute Werkzeuge (51 gesamt).

9.1 Warum ein eigener MCP-Server?

Der MCP-Standard von Anthropic definiert ein sauberes Protokoll, mit dem ein Modell Werkzeuge ansprechen kann, die *außerhalb* des eigentlichen LLM-Prozesses laufen. Der Vorteil für uns: der MCP-Server lässt sich unabhängig deployen, hardenen, updaten und skalieren, ohne den Orchestrator anzufassen. Er läuft als eigener Container (`mcp-precision`, Port 8003) und tauscht Anfragen und Ergebnisse über eine typisierte JSON-Schnittstelle aus.

9.2 Die AST-Whitelist des `calculate`-Tools

Das wichtigste Werkzeug ist `calculate`: es akzeptiert einen arithmetischen Ausdruck als String und liefert das Ergebnis zurück. Unter der Haube verwendet es zwei Stufen:

1. **Safe AST evaluator**: der Ausdruck wird mit `ast.parse` in einen Python-AST überführt. Anschließend werden ausschließlich die Knotentypen aus einer Whitelist zugelassen: `Expression`, `BinOp`, `UnaryOp`, `Num`, `Constant`, `Call` (nur für Whitelist-Funktionen), `Name` (nur für Whitelist-Namen). Alle anderen Knoten – etwa `Attribute`, `Subscript`, `Import`, `Lambda`, `Assign` – lassen die Evaluation fehlschlagen. Damit sind `__import__(ös")`, Attribut-Zugriff auf beliebige Objekte und arbitrary Code im Keim erstickt.

2. **SymPy-Fallback bei `SyntaxError`**: wenn die Eingabe keine gültige Python-Syntax ist, aber ein plausibel mathematischer Ausdruck (etwa „15 % von 100“ oder „3(2+4)“ mit impliziter Multiplikation), wird SymPy als Fallback verwendet – ebenfalls in einer eng kontrollierten Umgebung. Der Fallback greift *nur* bei `SyntaxError`, nicht bei anderen Exceptions. Das ist eine bewusste Härtung gegen eine frühere Lücke (siehe Lessons Learned in Abschnitt 14).

9.3 Der vollständige Werkzeug-Katalog

Über das `calculate`-Tool hinaus exportiert der MCP-Server 50 weitere Werkzeuge (51 gesamt). Die folgende Tabelle listet die Kern-Werkzeuge auf; neu hinzugekommen sind 8 wissenschaftliche Recherche- und Web-Werkzeuge (siehe unten).

Werkzeug	**Zweck**
`calculate`	AST-whitelisted Arithmetik mit SymPy-Fallback.
`solve_equation`	Symbolisches Gleichungslösen (SymPy).
`date_diff`	Tagesdifferenz zwischen zwei Datums-Strings.
`date_add`	Addiert Zeitintervalle zu einem Datum.
`day_of_week`	Wochentag eines beliebigen Datums.
`unit_convert`	Einheitenumrechnung mit `pint`.
`statistics_calc`	Mittelwert, Median, Standardabweichung, Perzentile.
`hash_text`	MD5/SHA-1/SHA-256/SHA-512 eines Strings.
`base64_codec`	Base64-Encode / -Decode.
`regex_extract`	Deterministisches Regex-Matching.
`subnet_calc`	IPv4-Subnetz-Analyse (Netzmaske, Broadcast, Hosts).
`text_analyze`	Zählen von Wörtern, Zeichen, Zeilen, Sätzen.
`prime_factorize`	Primfaktor-Zerlegung.
`gcd_lcm`	GGT und KGV.
`json_query`	JSONPath-Abfrage gegen ein JSON-Dokument.
`roman_numeral`	Römische Zahlen in beide Richtungen.
`legal_search_laws`	Volltextsuche über den Paragraphen-Index von `gesetze-im-internet.de`.
`legal_get_law_overview`	Inhaltsverzeichnis eines deutschen Gesetzes.
`legal_get_paragraph`	Liefert den Volltext eines einzelnen Paragraphen.
`legal_fulltext_search`	Vergleich von Treffern quer über mehrere Gesetze.
`repo_map`	AST-basierte Karte eines Python-Repositorys.
`read_file_chunked`	Paginierte, speicherschonende Datei-Lektüre.
`lsp_query`	LSP-Abfrage (Definitions, References) über `jedi`.
`generate_pptx`	Erstellt eine vollständig formatierte `.pptx`-Präsentation aus strukturiertem Inhalt (Titel, Folien, Aufzählungen, Notizen); liefert einen signierten MinIO-Download-Link.
Wissenschaftliche Recherche- und Web-Werkzeuge (neu, April 2026)	
`wikidata_sparql`	Wikidata SPARQL-API; deterministisch, kein SearXNG-Rauschen.
`pubmed_search`	NCBI/PubMed-Suche nach biomedizinischen Publikationen.
`crossref_lookup`	DOI- und Paper-Metadaten über die Crossref-API.

Fortsetzung

Werkzeug	Zweck
`openalex_search`	Suche in 250 Mio.+ wissenschaftlichen Werken (OpenAlex).
`duckduckgo_search`	Alternativer Web-Suchdienst; ergänzt SearXNG bei Ausfällen.
`web_browser`	Splash-JS-Rendering für dynamische Seiten (JavaScript-Gates).
`wayback_fetch`	Wayback Machine (dual-strategy): direkter Abruf + Fallback auf API.

9.4 Warum gerade diese Werkzeuge?

Die Auswahl ist nicht zufällig. Sie deckt drei Problem-Familien ab, die in LLM-Hauptanwendungen typisch sind und die wir nicht den probabilistischen Modellen überlassen wollen:

- **Arithmetik und Einheiten**: `calculate`, `solve_equation`, `statistics_calc`, `unit_convert`, `gcd_lcm`, `prime_factorize`, `roman_numeral`, `subnet_calc`.
- **Kryptographie und Kodierung**: `hash_text`, `base64_codec`, `regex_extract`, `text_analyze`, `json_query`. Alle Operationen, die auf genaue Ergebnisse angewiesen sind.
- **Strukturierte externe Quellen**: die vier `legal_*`-Tools greifen auf den offiziellen XML-Datenbestand des BMJ zu (*gesetze-im-internet.de*). Sie können auch offline betrieben werden, wenn ein Snapshot im Container vorgehalten wird; dadurch bleiben Rechtsberatungs-Workflows netzwerk-souverän.
- **Code-Navigation**: `repo_map`, `read_file_chunked` und `lsp_query` dienen dem Agentic-Coder-Experten, der auf fremden Repositories arbeitet, ohne den vollständigen Quelltext in sein Kontextfenster laden zu können.
- **Wissenschaftliche Recherche und Web**: Die sieben neuen Werkzeuge (`wikidata_sparql`, `pubmed_search`, `crossref_lookup`, `openalex_search`, `duckduckgo_search`, `web_browser`, `wayback_fetch`) erschließen primäre Wissensquellen, die SearXNG nicht zuverlässig erreicht. Insbesondere `wikidata_sparql` lieferte bei den GAIA-Runs deterministisch korrekte Antworten auf Faktenfragen, bei denen SearXNG run-to-run schwankte.

Lessons Learned: Die SymPy-Lücke

In einer frühen Version war der SymPy-Fallback zu breit gehalten: jede Exception im Safe-AST-Pfad fiel auf SymPy zurück, auch solche, die durch verbotene Knoten wie `Attribute` ausgelöst wurden. Ein Testfall `__import__('os').system('id')` lief dadurch in SymPy weiter – und SymPy rief in bestimmten Builds tatsächlich die Shell auf. Der Fix ist ein Zweizeiler: Fallback nur bei `SyntaxError`. Die Testsuite enthält seitdem einen expliziten Assertion-Test für diese Angriffssequenz.

10 Self-Correction-Loop

Jedes LLM macht Fehler. Eine Orchestrierungsschicht, die den Anspruch hat, langfristig in Produktion zu laufen, muss einen Mechanismus vorsehen, mit dem diese Fehler gesammelt, bewertet und in zukünftige Entscheidungen eingespeist werden. MoE Sovereign verbindet zu diesem Zweck drei Signale: *Self-Evaluation*, *User-Feedback* und *Ontologie-Gap-Erkennung*. Die drei Signale speisen einen gemeinsamen, inspizierbaren Score, der bereits in Abschnitt 6 als Gating-Input für Tier-2-Modelle eingeführt wurde.

10.1 Self-Evaluation im Judge-Node

Der letzte Knoten der Pipeline (`judge`) bekommt die zusammengefassten Antworten der Experten-Worker und die Originalfrage als Prompt. Seine Aufgabe ist es, eine selbstbewertete Konfidenz zwischen 0 und 1 zu emittieren – im Prompt explizit mit einem Score und einer kurzen Begründung, die für Debugging herangezogen werden kann. Der Score wird als Prometheus-Histogramm (`moe_self_eval_score_bucket`) exportiert und dient zwei Zwecken: erstens als Gating-Signal für die Aktivierung des `thinking_node` (Chain-of-Thought bei Low-Confidence), zweitens als Input für die Tier-2-Entscheidung.

Der Judge ist bewusst *eine eigene Pipeline-Stufe* und kein eingebettetes Postprocessing innerhalb des Merger-Nodes. Das macht sein Verhalten inspizierbar, austauschbar und separat kostenbewert. Die Trennung von Kritik und Generierung spiegelt dieselbe Erkenntnis wie Constitutional AI [36]: Ein dedizierter Evaluator-Node liefert zuverlässigere Selbstbewertung als eine im generierenden Modell eingebettete Evaluierung.

10.2 User-Feedback

Der zweite Signalpfad ist explizites Nutzer-Feedback. Frontends, die an den Orchestrator angebunden sind, können pro Response einen einfachen Daumen-hoch-/Daumen-runter-Event emittieren. Der Event wird über das Topic `moe.feedback` in Kafka persistiert und vom Ingest-Prozess in das Valkey-Register `moe:perf:{model}:{category}` geschrieben, in das auch die Self-Eval-Werte einfliessen. Die Metrik `moe_feedback_score_bucket` macht das Gesamtsignal für Prometheus verfügbar.

10.3 Laplace-geglättetes Konfidenz-Update

Beide Signalpfade landen in derselben Beobachtungszählung. Der Score für ein (model, category)-Paar wird als $(M + 1)/(N + 2)$ berechnet, wobei N die Gesamtzahl der Beobachtungen und M die Anzahl der positiven ist. Die `+1`/`+2` Konstanten entsprechen einer Laplace-Glättung: sie sorgen dafür, dass ein neues Modell nach wenigen Messungen nicht bei 0 oder 1 festhängt, sondern um einen neutralen Wert von 0.5 fluktuiert. Das ist bewusst einfach gehalten; komplexere bayessche Schätzer wären möglich, aber die Gewinnspanne ist gering und die Interpretierbarkeit sinkt.

Score-Semantik

Der Score ist ausdrücklich *kein* Qualitätsmaß im absoluten Sinne. Er misst, wie oft Nutzer und Judge-Node bei einem konkreten Modell-Kategorie-Paar zufrieden waren.

Zwei Modelle mit identischer Score-Zahl können fachlich sehr unterschiedlich sein; der Score sagt nur, dass beide in der beobachteten Vergangenheit vergleichbar funktioniert haben.

10.4 Tier-2-Gating in der Praxis

Der Gating-Mechanismus ist einfach: wenn das Tier-1-Modell einer Kategorie einen Score unter dem konfigurierbaren Schwellwert aufweist (Default 0.5), wird der `fallback`-Eintrag des Templates zusätzlich ausgeführt. Die Antworten beider Tiers gehen in den Merger, der nach einer Quellenprioritätsregel entscheidet, welche Teilantwort Vorrang hat. Die Regel ist Teil des Prompts des Merger-Nodes und lautet, frei übersetzt: *Reasoning-Antworten schlagen MCP-Tool-Antworten schlagen Graph-Antworten schlagen Experten-Antworten schlagen Web-Recherche schlagen Cache.*

10.5 Ontologie-Gap-Erkennung

Das dritte Signal ist subtiler, aber langfristig wertvoll: die Erkennung *fehlenden Wissens*. Der Ingest-Prozess vergleicht die im Merger-Output erwähnten zentralen Begriffe mit den Labels im Neo4j- Graphen. Begriffe, die keinen Match finden, werden als „Ontologie-Gap“ in Prometheus gezählt (`moe_ontology_gaps_total`). Der Admin kann im Admin-UI die Top-Gaps der letzten n Tage einsehen und entscheiden, ob die Ontologie-Datei erweitert werden soll. Damit hat das System einen rückkanaligen Indikator für seine eigenen Blindstellen – eine Eigenschaft, die in statischen RAG-Systemen fehlt.

10.6 Der Akkumulations-Effekt

Zusammen ergeben die drei Signalpfade – Self-Eval, User-Feedback, Gap-Erkennung – einen *Akkumulations-Effekt*: jede Interaktion macht das System ein kleines Stück informierter. Der Effekt ist bewusst langsam und inspektierbar: alle Schritte sind in Metriken abgebildet, alle Änderungen sind in Valkey und Neo4j persistiert, alle kritischen Schritte (Contradiction-Detection, Ontologie-Erweiterung) gehen durch einen menschlichen Review. Das Gegenteil wäre ein unkontrolliertes Online-Lernen, bei dem niemand mehr sagen kann, warum das System sich heute anders verhält als gestern. Unser Ansatz lehnt das ausdrücklich ab.

10.7 Agentic Re-Planning Loop

Der Akkumulations-Effekt beschreibt langfristiges Lernen über viele Anfragen. Eine neuere Ergänzung des Systems adressiert eine verwandte, aber kurzfristigere Frage: Was tun, wenn eine einzelne Anfrage nach dem ersten Merger-Durchlauf noch *unvollständig* ist?

Der *Agentic Re-Planning Loop* beantwortet das: Nach jeder Merger-Synthese prüft ein leichtgewichtiger LLM-Aufruf („Gap-Detektor“), ob die Antwort vollständig ist oder noch offene Fragen enthält. Die Prüfung liefert ein einfaches binäres Urteil: `COMPLETION_STATUS: COMPLETE | NEEDS_MORE_INFO`.

Bei `NEEDS_MORE_INFO` wird der noch offene Teil – zusammen mit allen bisher ermittelten Fakten – als strukturierter Kontext in eine neue Planner-Runde injiziert. Der Planner erhält also

nicht die ursprüngliche Frage erneut, sondern einen fokussierten Auftrag: *hole genau dieses fehlende Stück*. Das Routing geht dann gezielt an `web_researcher` oder `precision_tools` – nicht an alle Experten. Nach maximal drei agentischen Iterationen gibt das System die beste verfügbare Antwort zurück.

Schutz vor Re-Trigger

Wenn der Merger-Output einen `SKILL_TRIGGER`-Token, einen `/downloads/`-Pfad oder ein `DOWNLOAD_URL`-Token enthält, überspringt der Gap-Detektor die Prüfung und markiert die Antwort sofort als `COMPLETE`. Dies verhindert, dass ein bereits erzeugtes Artefakt (z. B. eine PPTX-Datei) in einer Folge-Runde doppelt generiert wird.

Der Loop ist komplementär zum Self-Correction-Loop: Dieser lernt *zwischen* Anfragen; der Agentic Loop löst Lücken *innerhalb* einer einzelnen Anfrage – ohne Nutzerinteraktion. Das Muster ist eine Instanz der iterativen Selbstverbesserungsidee, die von Madaan et al. [37] formalisiert wurde: Generieren, strukturiertes Feedback geben, verfeinern – wiederholt bis ein Vollständigkeitskriterium erfüllt oder ein Iterationsbudget erschöpft ist.

10.8 Parakonsistente Konfliktauflösung

Ein separater Node, `resolve_conflicts_node`, adressiert einen anderen Fehlerfall: nicht eine *unvollständige* Antwort, sondern eine *widersprüchliche* – zwei Experten-Outputs, die nicht beide wahr sein können.

Der Node implementiert eine parakonsistente Auflösungsstrategie. De Vries (2007) [29] liefert die algebraische Motivation: Innerhalb einer einheitlichen Gitterhierarchie belegt parakonsistente Logik eine Position, die Widersprüche toleriert, anstatt ihre logischen Konsequenzen zu propagieren (wie es die klassische Logik via *ex contradictione quodlibet* täte); der Widerspruch wird explizit gemacht, damit nachgelagerte Stufen handeln können, ohne dass der gesamte Inferenzprozess zusammenbricht.

Formale Definition. Sei $\mathcal{P}$ eine Aussage und $\neg\mathcal{P}$ ihre Negation. Die klassische Logik gehorcht dem *Explosionsprinzip*: $\{\mathcal{P}, \neg\mathcal{P}\} \vdash \mathcal{Q}$ für beliebiges $\mathcal{Q}$, was Widersprüche global destruktiv macht. Nach Belnap (1977) [30] wird dies durch eine vierwertige Semantik $\{T, F, B, N\}$ ersetzt, wobei B („beide“) einen *lokalisierten* Widerspruch bezeichnet – er propagiert nicht.

In MoE Sovereign trägt jeder Eintrag in `conflict_registry` einen *Divergenz-Score* $d \in [0, 1]$, berechnet als

$$d(r_1, r_2) = 1 - \frac{|r_1 \cap r_2|}{|r_1 \cup r_2|} \tag{4}$$

der Jaccard-Distanz zwischen den Token-Mengen zweier Expertenantworten r_1, r_2 derselben Kategorie. Die Auflösungsstrategie ist schwellwertgesteuert:

$$\text{Aktion}(d) = \begin{cases} \textsc{dismiss} & d < 0{,}5 \\ \textsc{arbitrate} & d \geq 0{,}5 \ \wedge \ \text{Kat} \in \mathcal{C}_{\text{safety}} \end{cases} \tag{5}$$

wobei $\mathcal{C}_{\text{safety}} = \{\texttt{medical\_consult}, \texttt{legal\_advisor}\}$. Alle Einträge persistieren in `conflict_registry` mit Lebenszyklus PENDING → RESOLVED | DISMISSED und bilden einen manipulationssicheren Audit-Trail.

Zwei Auflösungsstrategien werden sequentiell angewendet:

1. **Strategie A – Auto-Dismiss** (Divergenz-Score $< 0{,}5$): Liegt die semantische Divergenz zweier widersprüchlicher Antworten unter dem Schwellwert, wird der Konflikt als *formulaische Variation* klassifiziert – unterschiedliche Formulierungen einer kompatiblen Aussage – und automatisch verworfen. Kein LLM-Aufruf ist erforderlich.

2. **Strategie B – Judge-Arbitration** (Divergenz $\geq 0{,}5$, sicherheitskritische Kategorie): Für `medical_consult` und `legal_advisor` mit signifikanter Divergenz wird das Judge-Modell mit einem expliziten Arbitrations-Prompt aufgerufen. Um eine vollständig lokale und souveräne Ausführung auf einer einzelnen 24 GB GPU (z. B. `NO4-RTX`) zu ermöglichen, wird ein spezialisierter *Konflikt-Schlichter-Judge* auf Basis des dichten Modells `granite4.1:30b` feingetunt. Dieses Modell ist vom *Allgemeinen Parakonsistenten Judge* (`sovereign-judge:35b-q4km`, Basis: Qwen3.6-35B-A3B) zu unterscheiden, der für die Selbstbewertung zuständig ist und im LUMI-G-Begleitpaper [38] beschrieben wird.

   Das Training basiert auf einem parakonsistenten Datensatz von 90.000 Samples, der asynchron auf dem EuroHPC-LUMI-G-Supercomputer generiert wurde. Zur Sicherung der Ground-Truth-Qualität kombiniert der Compiler eine Befürworter-Antwort (`CLAIM A`, generiert von GPT-4 aus RouteLLM-Seeds) mit einer gegnerischen Kritik (`CLAIM B`, generiert von Mixtral-8x22B). Das Schlichtungs-Urteil wird von einem starken Lehrer-Modell (Qwen2.5-32B) generiert, das die strittigen Punkte in einer XML-basierten Konflikt-Map strukturiert – mit Belegen, Konfidenz und vierwertigen Wahrheitstypen nach der Belnap-Dunn-Logik [30] $\{T, F, B, N\}$, wobei B (Beide) und N (Keiner) den internen Dataset-Labels I (Inkonsistent) bzw. U (Unentschieden) entsprechen – gefolgt von einem abschließenden Synthese-Urteil. Die Datenqualität wird live überprüft: Syntaktisch fehlerhafter XML- oder JSON-Code sowie ungültige Wahrheitstypen führen zum direkten Ausschluss des Samples. Das resultierende Datenset dient dem LoRA-SFT-Feintuning von `granite4.1:30b`, was eine robuste, souveräne und latenzarme Schlichtung garantiert.

Alle Konflikt-Einträge verbleiben in `conflict_registry`, unabhängig vom Auflösungsergebnis. Offene Einträge sind im Admin-UI-Audit-Trail sichtbar; aufgelöste Einträge tragen eine `resolved_by`-Annotation. Das macht die Konflikterkennung selbst zu einem Transparenzsignal: Betreiber können einsehen, welche Experten-Paare am häufigsten widersprechen, und diese Information nutzen, um Templates zu verfeinern.

Warum Parakonsistenz in regulierten Domänen?

Eine klassische Merge-Strategie, die einfach die Antwort mit höherer Konfidenz wählt, verwirft Information über den Dissens selbst. In medizinischen oder juristischen Kontexten ist die Tatsache, dass zwei qualifizierte Quellen *einander widersprechen*, oft operativ bedeutsamer als jede der einzelnen Antworten. Parakonsistente Auflösung bewahrt den Konflikt als auditierbares Artefakt, statt ihn still aufzulösen.

11 Einheitliches OCI-Artefakt

Eine der zentralen, technisch ambitioniertesten Eigenschaften des Projekts ist, dass *dasselbe* OCI-Image vom Hobby-LXC bis zum Enterprise-Cluster ohne Code-Fork läuft. Dieser Abschnitt beschreibt, wie das erreicht wird, welche Deployment-Wrapper unterstützt werden und welche harten Invarianten in allen Schichten gelten müssen.

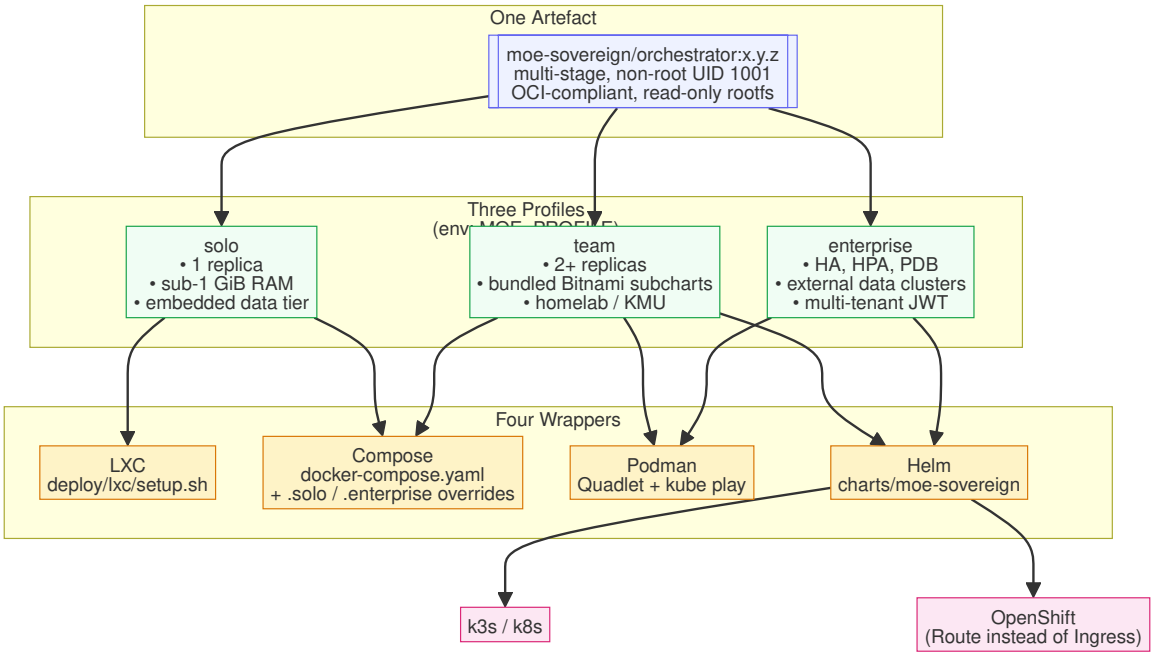


Abbildung 6: Das Universal-Deployment-Prinzip: ein einziges OCI-Artefakt (*single artifact*), drei Profile (`solo`, `team`, `enterprise`), vier Deployment-Wrapper (LXC-Script, Docker Compose, Podman Quadlet, Helm Chart), laufend auf fünf Ziel-Plattformen (LXC, Docker-Host, k3s, Kubernetes, OpenShift).

11.1 Single-Image-Deployment

Der Orchestrator wird als Multi-Stage-Dockerfile gebaut, das in einer Builder-Stufe die Python-Dependencies vorauflöst und in einer Runtime-Stufe nur noch `python:3.11-slim` plus das installierte Site-Packages-Verzeichnis enthält. Das Ergebnis ist ein Bild unterhalb von 400 MiB komprimiert. Dasselbe Image-Tag wird von jedem Deployment-Wrapper konsumiert. Es gibt keine *Enterprise-Edition* und keine *Community-Edition*; es gibt ein Image, das überall gleich funktioniert.

Das folgt etablierter OCI-Best-Practice: ein einziges, unveränderliches Artefakt, runtime-injizierte Konfiguration, keine umgebungsspezifischen Build-Varianten. Der Ansatz ist nicht neu, aber er wird bewusst gewählt – und es lohnt sich, ihn explizit zu benennen, weil viele selbst gehostete KI-Projekte früh davon abweichen.

Die wichtigsten Eigenschaften dieses Images sind:

- **Non-Root**: alles läuft als UID 1001 (`moe`), inklusive des Python-Prozesses. Der Container-Entrypoint wechselt explizit den Benutzer. Das Image ist zudem Arbitrary-UID-kompatibel:

`chgrp -R 0 /app && chmod -R g=u /app` gewährt GID 0 Schreibzugriff auf alle Applikationsverzeichnisse und erfüllt damit OpenShifts Security Context Constraints (SCC), die Container-UIDs zur Laufzeit randomisieren.

- **Read-Only Root Filesystem**: in Kubernetes, OpenShift und Podman ist das Root-Dateisystem schreibgeschützt. Die wenigen Schreibpfade (Logs, Caches, temporäre LangGraph-Zustände) werden über `emptyDir`-Volumes bzw. `tmpfs` bereitgestellt.
- **OCI-Labels**: alle Metadaten (`org.opencontainers.image.*`) sind korrekt gesetzt, inklusive Quellverweis, Build-Datum und Version.
- **Dropped Capabilities**: alle Linux-Capabilities werden im Container-Runtime gedroppt. Der Orchestrator braucht keinen einzigen davon.
- **Kein Default-Admin**: der erste Admin muss vom Betreiber initial angelegt werden (im Admin-UI oder über eine `.env`-Variable). Ein Default-Passwort gibt es nicht.

11.2 Drei Profile

Das Verhalten des Orchestrators wird über die Umgebungsvariable `MOE_PROFILE` gesteuert. Gültige Werte sind `solo`, `team` und `enterprise`. Die Unterschiede betreffen ausschliesslich Default-Ressourcenlimits und welche Nebenservices erwartet werden; der Code-Pfad ist identisch.

Profil	Zielgruppe	Charakteristik
`solo`	Einzelnutzer, Edge, Proxmox-LXC	1 Replika, < 1 GiB RAM, optional eingebettetes Daten-Tier.
`team`	KMU, Homelab, 1 Server	2 Replikas, gebündelte Bitnami-Subcharts bzw. voller Compose-Stack.
`enterprise`	Behörde, Konzern	Externes Daten-Tier, HPA, PDB, Network Policies, OIDC, JWT-Tenancy.

11.3 Vier Deployment-Wrapper

Zu jedem Profil gehört ein konkret lauffähiger Wrapper. Die vier unterstützten Wrapper sind:

1. **LXC-Bootstrap-Script** (`deploy/lxc/setup.sh`): ein einzelnes Bash-Script, das ein frisches Debian- oder Ubuntu-LXC in unter 60 Sekunden einsatzbereit macht. Es installiert rootless Podman [39], legt einen unprivilegierten Service-Benutzer mit `systemd-linger` an, installiert eine Quadlet-Unit und optional Grafana Alloy als systemd-Service für das Log-Shipping nach Loki.
2. **Docker Compose** (`docker-compose.yaml`): der klassische Weg. Ein `docker compose up -d` startet 19 Container in der richtigen Reihenfolge. Für ein Homelab-Setup ist dies der empfohlene Weg.
3. **Podman Quadlet** (`deploy/podman/systemd/moe-orchestrator.container`): eine Systemd-native Unit für Podman ≥ 4.4. Die Unit hat dieselben Security-Einstellungen wie das Helm-Chart (`ReadOnly=true`, `NoNewPrivileges=true`, `DropCapability=ALL`), nutzt `journald` als Log-Treiber und kann per `systemctl` verwaltet werden.

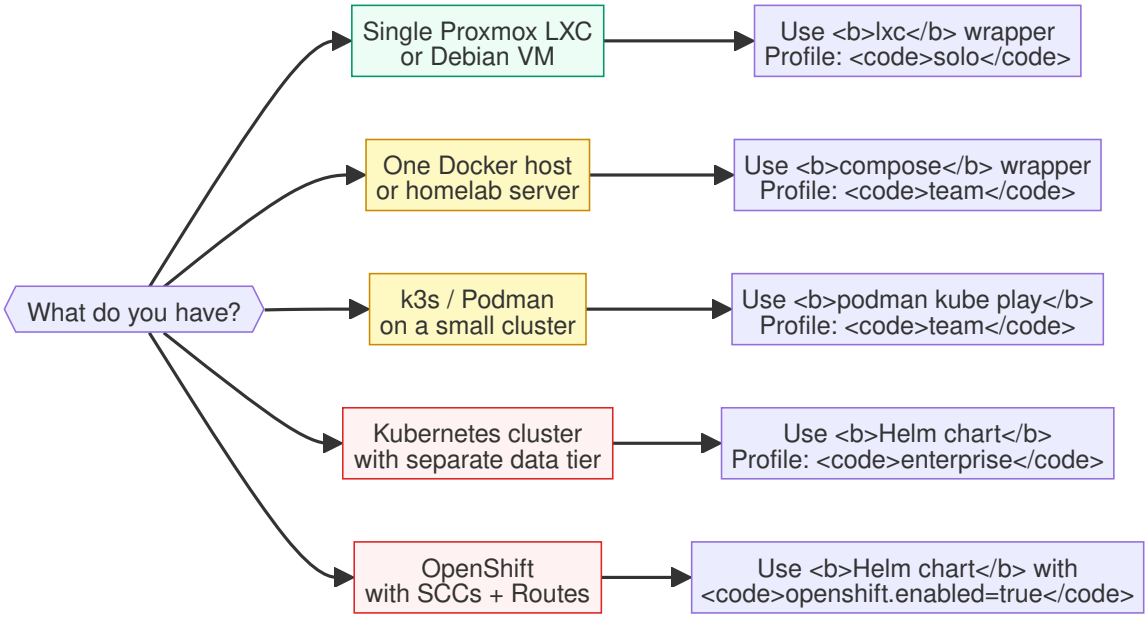


Abbildung 7: Die Tier-Entscheidungshilfe: welches Profil und welcher Wrapper für welches Deployment-Ziel empfohlen wird.

4. **Helm Chart** (`charts/moe-sovereign/`): der Kubernetes-Weg. Das Chart kann wahlweise ein vollständiges Deployment inklusive PostgreSQL, Valkey, Kafka und Neo4j als Bitnami-Subcharts [40] mitbringen (*team*-Profil), oder auf externe Cluster verweisen (*enterprise*-Profil). Ein integrierter `capabilities`-Switch erkennt OpenShift-Umgebungen anhand der `route.openshift.io/v1`-API und generiert dann Route- statt Ingress-Objekte. Damit ist ein und dasselbe Chart ohne Modifikationen für k3s, vanilla Kubernetes und OpenShift [41] nutzbar.

11.4 LXC: rootless Podman als Brücke

Das LXC-Setup-Script verdient besondere Erwähnung, weil es ein historisches Problem löst: Docker in einem unprivilegierten Proxmox-LXC erfordert umfangreiche Nesting-Konfiguration und bricht bei jedem Kernel-Update. Rootless Podman in einem unprivilegierten LXC funktioniert dagegen out-of-the-box, weil Podman keine Daemon-Prozesse braucht und die User-Namespace-Mappings direkt aus `/etc/subuid` und `/etc/subgid` bedient.

11.5 Die Invarianten

Über alle vier Wrapper hinweg gelten dieselben harten Invarianten. Diese werden durch eine dedizierte Testsuite (`tests/test_deployment_artifacts.py`) erzwungen, die Dockerfile, Helm-Chart und LXC-Bootstrap-Script auf Konsistenz prüft.

- UID 1001 ist überall *dieselbe* UID.
- Port 8000 ist überall *derselbe* Port.

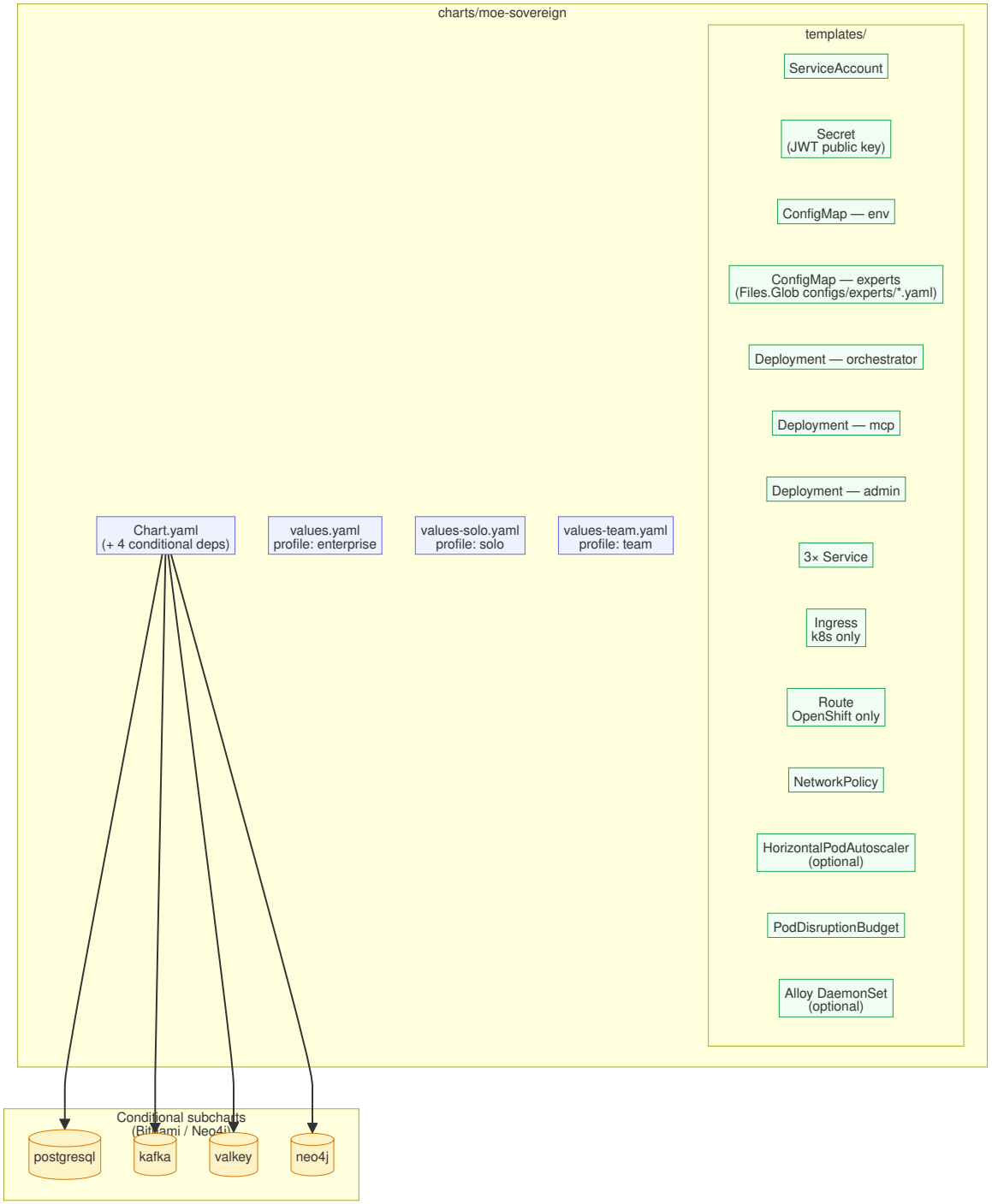


Abbildung 8: Die Helm-Chart-Struktur: `Chart.yaml` mit bedingten Bitnami-Subcharts, drei Values-Dateien für die drei Profile und 14 Template-Dateien inklusive OpenShift-Route-Switch.

- `MOE_LOGS_DIR`, `MOE_CACHE_DIR` und `MOE_EXPERTS_DIR` sind in allen drei Wrapper-Formen gesetzt.
- Das Read-Only-Rootfs-Pattern gilt überall.
- Die Healthcheck-Endpunkte (`/health`) sind kompatibel.

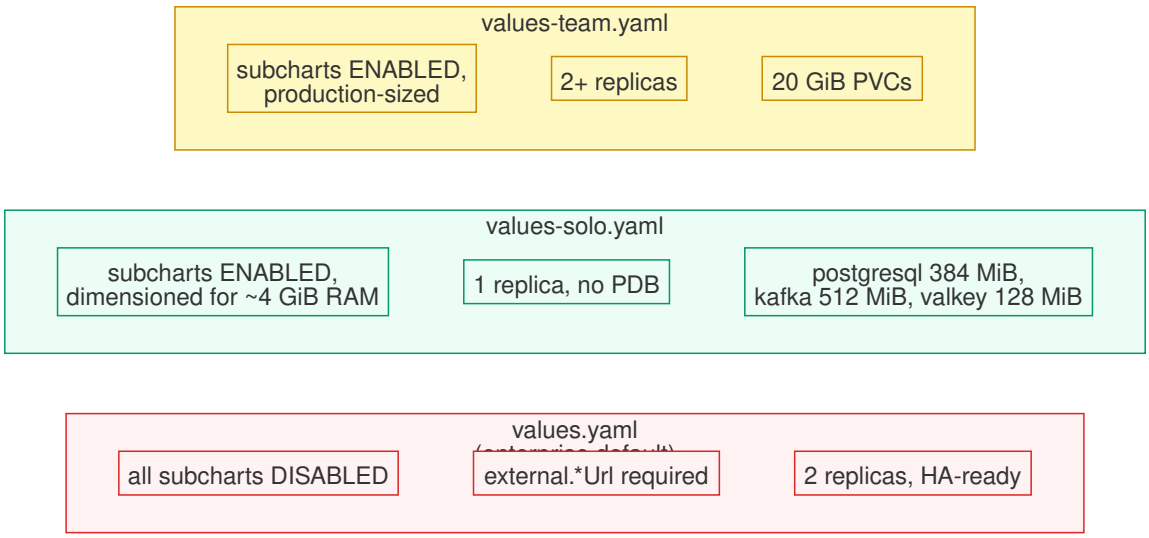


Abbildung 9: Vergleich der drei Profile: `solo`, `team`, `enterprise` im Helm-Values-Format.

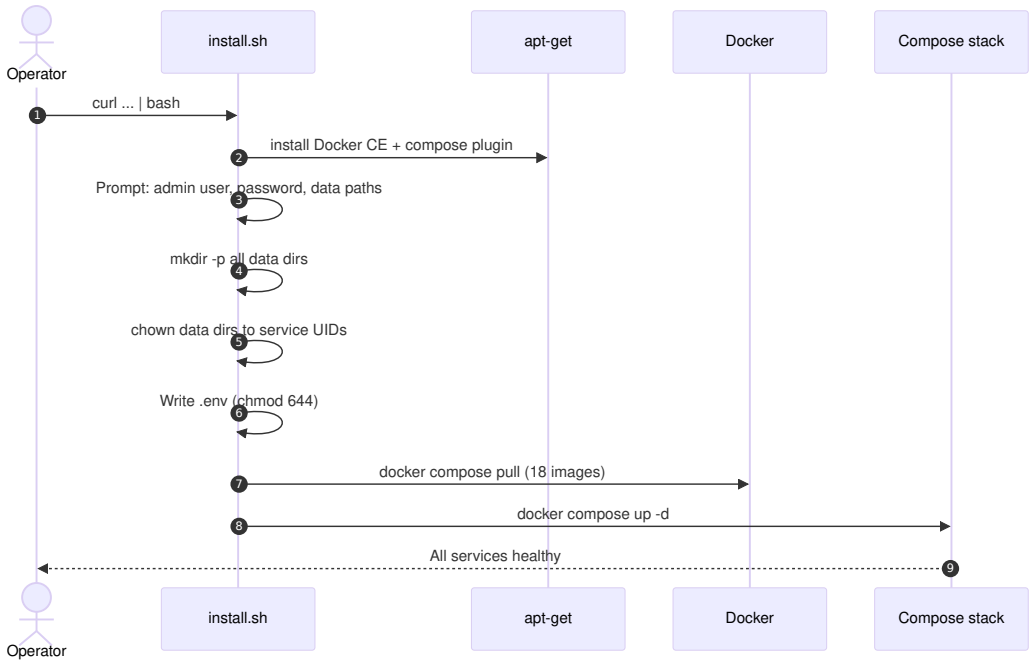


Abbildung 10: Rootless Podman in einem unprivilegierten LXC: der Service-Benutzer (UID 1001) startet den Container über eine Quadlet-Unit, systemd verwaltet den Lebenszyklus, Grafana Alloy liest die Logs direkt aus dem Journald-Cursor.

Innovation: Ein Image, kein Fork

Die weitverbreitete „Community vs. Enterprise“-Dichotomie kommt fast immer von einem Release-Modell, bei dem getrennte Code-Pfade für unterschiedliche Zielgruppen gepflegt werden. Wir halten diese Trennung für schädlich: sie teilt die Aufmerksamkeit der Maintainer, sie produziert Drift, sie ist Vertrauensbruch gegenüber der Community. Unsere Antwort ist das Universal-Deployment-Modell: ein Image, drei Profile, vier

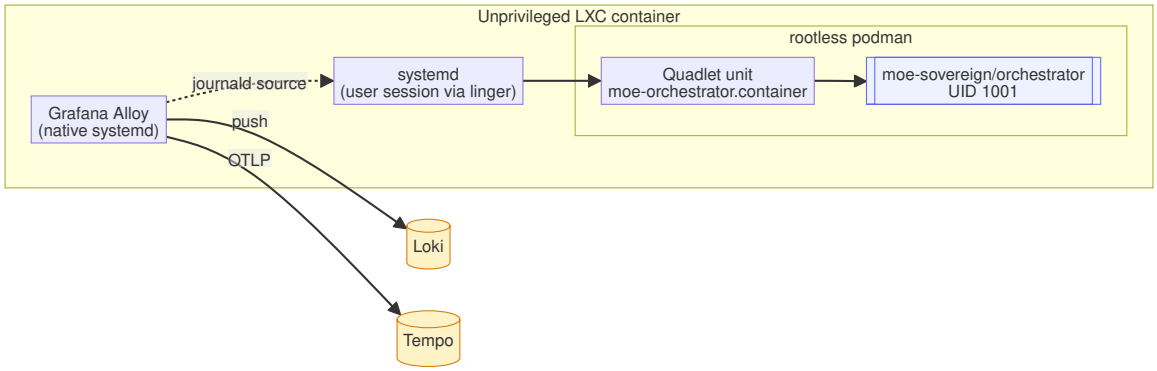


Abbildung 11: Die Sequenz des `deploy/lxc/setup.sh`-Scripts: Paketinstallation, Benutzeranlage, Linger aktivieren, Quadlet-Unit kopieren, Alloy einrichten. Gesamtdauer auf einem 2-vCPU-LXC: etwa eine Minute.

Wrapper, null Forks.

12 Observability und Tracing

Ein produktionsreifer Orchestrator muss beobachtbar sein, und zwar auf drei Ebenen: *Metriken* (numerische Zeitreihen über Verhalten und Ressourcen), *Logs* (strukturierte textuelle Aufzeichnungen), und *Traces* (kausale Ketten einer einzelnen Anfrage durch alle beteiligten Services). Der Reiz – und die technische Herausforderung – liegt darin, dass Anfragen in unserem Modell mehrere Deployment-Schichten durchqueren können: ein Request kann an einem LXC-Edge-Node ankommen, an einen Kafka-Cluster auf Kubernetes delegiert und von dort an einen externen Ollama-Server weitergereicht werden. Die Korrelation dieser Schritte zu *einer* Trace ist die Aufgabe, die wir mit dem *W3C-Traceparent-Header* [42] und einem einheitlichen Alloy-Collector [43] lösen.

12.1 Prometheus-Metriken

Der Orchestrator exportiert auf `/metrics` einen Prometheus [44]-kompatiblen Endpunkt. Die wichtigsten Metriken sind in Tabelle 7 aufgeführt. Alle Metriken tragen konsistente Labels: `model`, `category`, `deployment_profile` und, wo zutreffend, `tenant_id`.

Tabelle 7: Die wichtigsten Prometheus-Metriken des Orchestrators.

Metrik	Semantik
`moe_requests_total`	Anzahl aller eingehenden Anfragen, per Tenant/Kategorie.
`moe_request_duration_seconds`	Histogramm der End-zu-End-Latenz.
`moe_self_eval_score_bucket`	Histogramm der Judge-Self-Evaluation-Scores.
`moe_feedback_score_bucket`	Histogramm der User-Feedback-Scores.
`moe_cache_hits_total`	Cache-Hits, geschlüsselt nach `layer` (L1/L2/L3).

Metrik	Semantik
`moe_cache_misses_total`	Cache-Misses pro Layer.
`moe_ontology_gaps_total`	Unerkannte Begriffe ohne Graph-Match.
`moe_expert_worker_calls`	Anzahl der LLM-Aufrufe pro Expert-Worker-Instanz.
`moe_kill_requests_total`	Vom Admin beendete Requests, per Grund.
`moe_tenant_token_budget`	Verbleibendes Token-Budget pro Mandant.
`moe_tier_escalation_total`	Zwei-Tier-Routing-Ergebnisse je Experten-Kategorie; Label `decision` $\in$ {t1_high_skip, t1_cost_kept, t1_only, t2_escalated}.
`moe_cache_query_distance`	Histogramm der nächsten Cache-Eintrags-Kosinusdistanz je Lookup; ermöglicht datenbasierte Schwellwert-Kalibrierung.
`moe_routing_bandit_total`	Thompson-Bandit-Gate-Entscheidungen je Retrieval-Gate; Label `source` unterscheidet Bandit- vs. Heuristik-Entscheidungen.

12.2 Grafana-Dashboards

Das Projekt liefert vorgefertigte Grafana-Dashboards mit, die aus den obigen Metriken aufgebaut sind. Diese decken vier Ansichten ab: *Gesamtsystem* (Requests pro Sekunde, Fehlerrate, Latenz-Perzentile), *Caches* (Hitraten, Latenzverteilung), *LLM-Nutzung* (Anzahl der Aufrufe pro Modell und Kategorie, kumulative Tokens), und *Mandantenverbrauch* (Token-Budget, Rate-Limit-Events).

12.3 Loki und Alloy als universeller Log-Sammler

Für Logs setzen wir auf Grafana Loki und den universellen Alloy-Collector [43]. Der Reiz von Alloy ist, dass er zu einem identischen Prozess aus drei sehr unterschiedlichen Quellen lesen kann: `loki.source.journal` auf systemd-basierten Hosts (insbesondere LXC und bare-metal), `loki.source.docker` bei Compose-Deployments und `loki.source.kubernetes` im DaemonSet-Betrieb. Die Konfigurationsdatei `alloy.river` im Repository liegt in einer Version vor, die alle drei Quellen *gleichzeitig* unterstützt; je nach Deployment wird der nicht genutzte Pfad inaktiv.

12.4 Trace-Propagation: W3C traceparent

Der technisch wichtigste Teil ist die Trace-Propagation. Der Orchestrator liest den HTTP-Header `traceparent` (W3C Trace Context Level 1 [42]) auf eingehenden Requests und hängt ihn an *jede* ausgehende Verbindung weiter: an Ollama-Calls, an MCP-Tool-Calls, an Kafka-Messages (als User-Header), an Neo4j-Queries (als Logging-Annotation), und an interne LangGraph-Checkpoint-Schreibungen (als Zeitreihen-Label). Damit wird aus einer einzelnen Request-ID ein durchgängiger Faden, der sich in einer Tempo-Instanz rekonstruieren lässt.

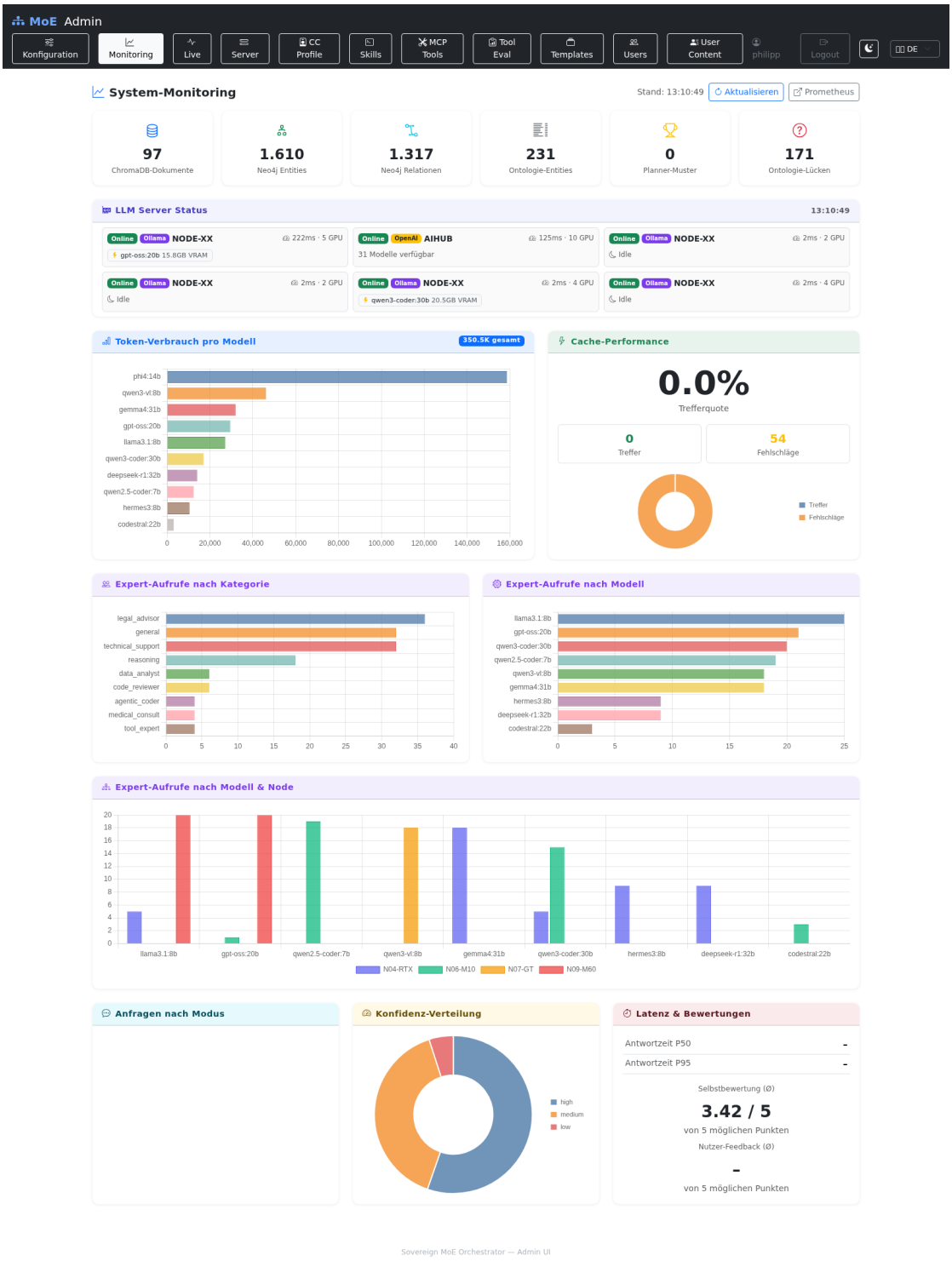


Abbildung 12: Das Admin-UI-Dashboard „System-Monitoring“. Alle sichtbaren Hostnamen sind in dieser Dokumentation durch `NODE-XX` ersetzt; im Betrieb zeigt das Dashboard die tatsächlichen Node-Namen der konfigurierten Inference-Server.

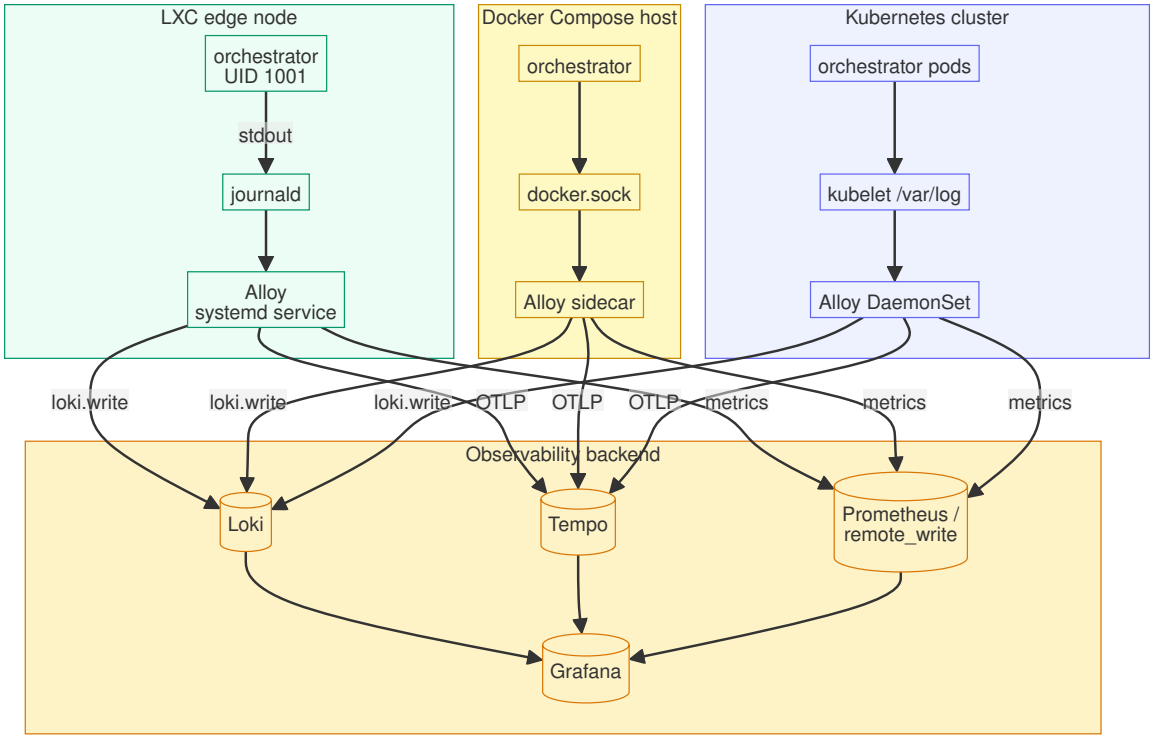


Abbildung 13: Die universelle Observability-Pipeline: Metriken fließen über Prometheus, Logs über Loki, Traces über Tempo. Alloy ist der einheitliche Collector auf allen drei Deployment-Zielen. Der `traceparent`-Header propagiert die Trace-ID durch die gesamte Kette.

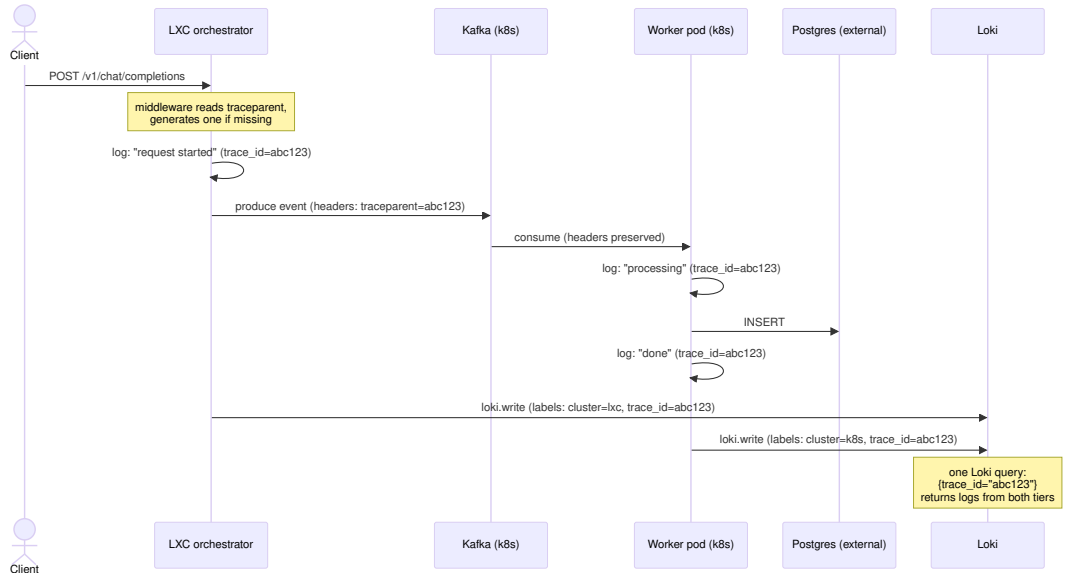


Abbildung 14: Die Propagation des `traceparent`-Headers durch die Deployment-Schichten. Eine Anfrage an den LXC-Edge-Node wird mit derselben Trace-ID durch Kafka und den k8s-Cluster weitergeleitet; Tempo kann die vollständige Kette darstellen.

12.5 Live-Monitoring: Aktive Requests und Kill-Mechanismus

Zusätzlich zu den Metriken verfügt das Admin-UI über eine Echtzeit-Sicht auf *aktive Requests*. Jeder laufende Request wird unter einem Valkey-Key `moe:active:{request_id}` mit Metadaten (Start-Zeit, Modell, Kategorie, Mandant) gespiegelt. Das Admin-UI listet diese Keys und bietet einen *Kill-Button*, der eine Nachricht ins Topic `moe.killswitch` schreibt. Der Orchestrator hört darauf und bricht die laufende LangGraph-Instanz am nächsten Checkpoint ab. Der gesamte Fluss ist in Abb. 15 dargestellt.

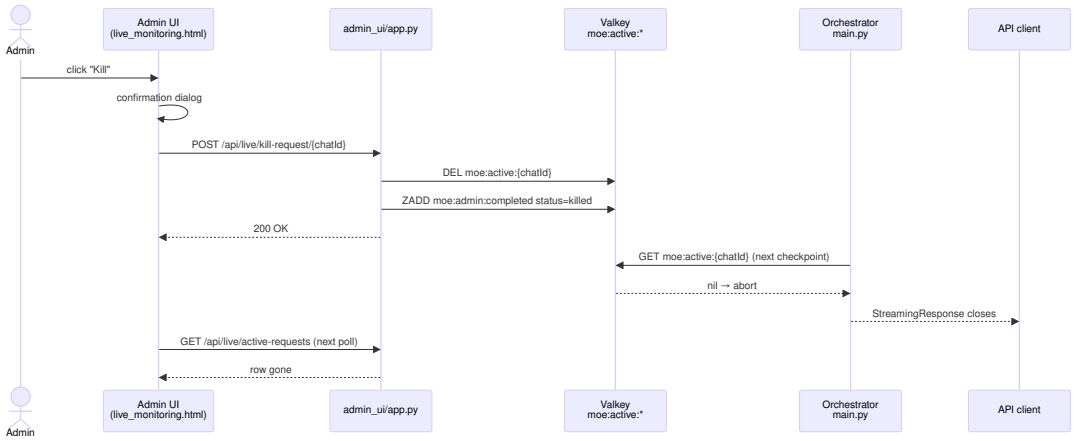


Abbildung 15: Der Kill-Flow eines aktiven Requests. Der Admin drückt den Kill-Button, das Admin-UI schreibt eine Kafka-Nachricht, der Orchestrator liest sie am nächsten Checkpoint, stoppt die LangGraph-Instanz, das Admin-UI aktualisiert die Anzeige. Latenz: unter eine Sekunde im Normalfall.

13 Sicherheit und Datenschutz

Sicherheit und Datenschutz sind nicht *ein* Kapitel im Whitepaper, sondern das Fundament der gesamten Architektur. Dieses Kapitel bündelt die Einzelaussagen, die in den vorherigen Abschnitten verstreut waren, und nennt die konkreten Umsetzungen, die wir für eine DSGVO-kompatible [7] Installation als notwendig und hinreichend halten.

13.1 Mehrstufige Isolation im Container-Bau

Die erste Verteidigungslinie ist das Container-Image selbst:

- **Non-Root**: UID 1001, keine Möglichkeit zur Privilege-Escalation.
- **Read-Only Rootfs**: `readOnlyRootFilesystem=true` in Kubernetes, `ReadOnly=true` in Podman-Quadlet, `-read-only` in Compose (dokumentiert, aber optional aktivierbar).
- **Dropped Capabilities**: `capabilities.drop: [ALL]`.
- **Keine SSH-Daemons, keine Cron-Daemons, keine Shells**: das Image enthält nur `python` und die ausgewählten System-Bibliotheken; es gibt kein `/bin/bash` im Runtime-Container.
- **Dedizierter Nutzer mit nologin**: der `moe`-User hat `/sbin/nologin` als Shell.

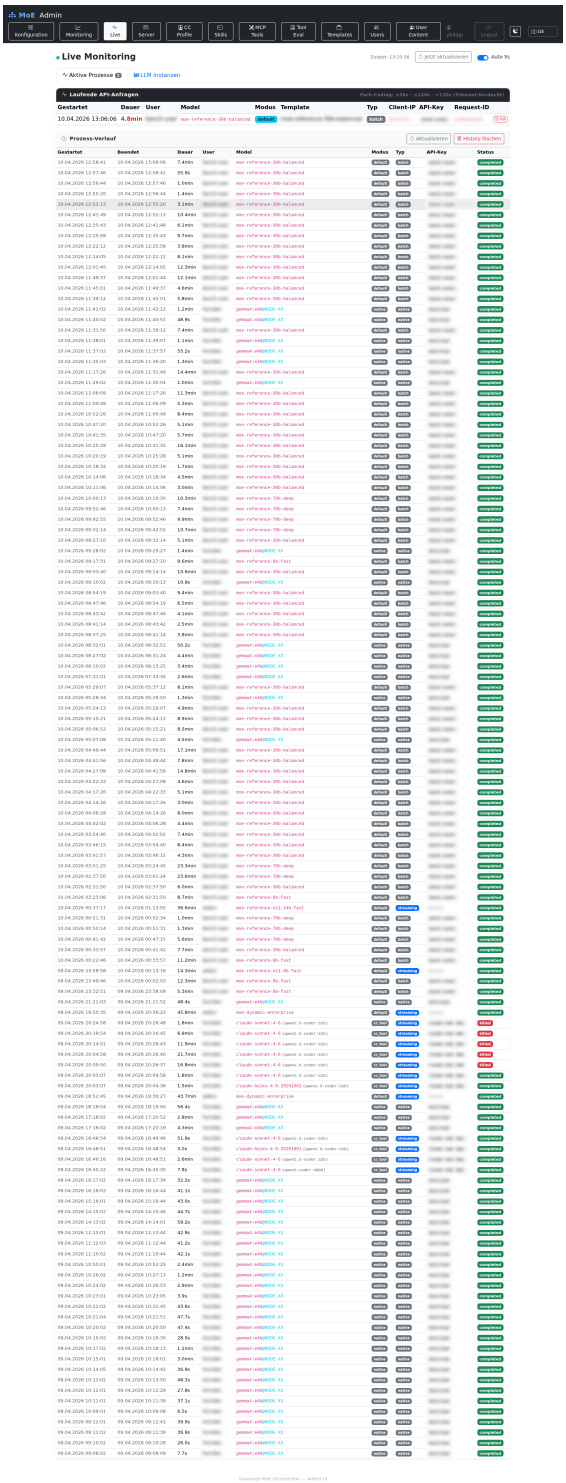


Abbildung 16: Das Live-Monitoring des Admin-UI mit sichtbarem aktiven Prozess (oben, Laufzeit 6.2 min) und historischer Prozessliste (unten). Alle identifizierenden Spalten (User, Client-IP, API-Key, Request-ID, Template) sind in dieser Dokumentation per Blur maskiert; in einer echten Installation sind sie für Administratoren sichtbar.

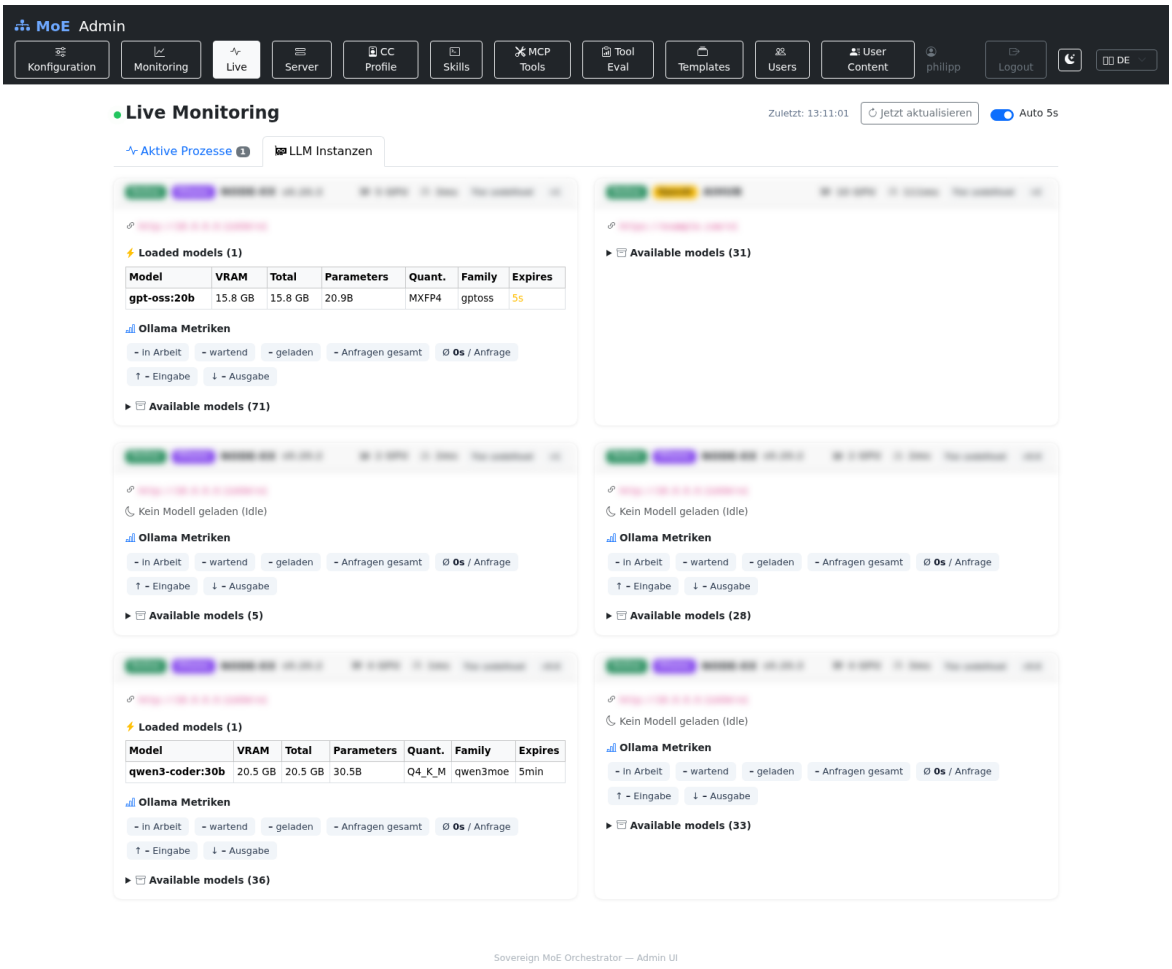


Abbildung 17: Die LLM-Instances-Ansicht im Admin-UI: pro konfiguriertem Inference-Server werden der Live-Status, die geladenen Modelle inklusive VRAM-Belegung und Quantisierung, die Ollama-Metriken (wartend, geladen, Durchsatz) und die Liste der verfügbaren Modelle angezeigt. Die Hostnamen-Header wurden anonymisiert.

13.2 JWT-basierte Mandantenauthentifizierung

Der Orchestrator akzeptiert Anfragen von Nutzern, die sich zuvor am Admin-UI mit Username/Passwort (bcrypt-gehashed) angemeldet haben. Beim Anmelden erhält der Nutzer ein signiertes JWT mit Gültigkeits dauer und Claims für `tenant_id`, `role`, `token_budget_remaining`. Das JWT wird vom Orchestrator bei jeder Anfrage validiert; die Validierung erfolgt rein lokal, ohne Netzwerkaufruf. In Kombination mit Authentik als OIDC-Provider kann dieser Mechanismus auch SSO-kompatibel betrieben werden.

13.3 Rate-Limiting

Die Rate-Limiting-Schicht basiert auf `slowapi`, das den Counter in Valkey persistiert. Limits werden auf drei Ebenen erzwungen:

1. Pro IP-Adresse (gegen DoS durch unauthenticated Requests).

2. Pro JWT-Token (gegen unbeabsichtigte Exzesse eines einzelnen Nutzers).

3. Pro Mandant (gegen Budget-Überschreitungen in einem gemeinsam genutzten Backend).

Überschreitungen produzieren einen 429-Response, werden als Prometheus-Metrik gezählt und können per Grafana-Alert an den Admin gemeldet werden.

13.4 Datenflüsse und Aufbewahrung

Ein zentraler Aspekt eines DSGVO-kompatiblen Systems ist, zu dokumentieren, *welche Daten wo landen und wie lange sie dort bleiben*. Die folgende Tabelle ist Teil der `docs/PRIVACY.md`-Datei im Repository und wird hier noch einmal zusammengefasst:

Speicher	**Inhalt**	**Standard-TTL**
Valkey (Caches)	L2-/L3-Cache-Einträge, Sessions, aktive Requests	30–60 min
Valkey (Performance Scores)	Feedback-Zähler pro (Modell, Kategorie)	unbefristet
ChromaDB	Semantischer L1-Cache, Embeddings	unbefristet (flagbar)
Neo4j	Wissensgraph, Ontologie, SYNTHESIS_INSIGHTS	unbefristet (versioniert)
Kafka	Event-Log (ingest, feedback, killswitch, linting)	7 Tage
PostgreSQL (Checkpoints)	LangGraph-Checkpoints	pro Session, löschbar
PostgreSQL (Admin-DB)	User, Budgets, Audit-Log	unbefristet
Alloy/Loki	Container- und Systemd-Logs	konfigurierbar

Die Löschpfade sind in `docs/PRIVACY.md` dokumentiert und können vom Betreiber per Docker-CLI oder Admin-UI ausgelöst werden. Es gibt keine *Self-Service-Deletion* für einzelne Nutzer in der aktuellen Version; dies ist eine bekannte Einschränkung und als Enhancement-Ticket offen.

13.5 Security-Hardening 2026-04-25: Produktionshärtung

Im Rahmen eines vollständigen Sicherheits-Audits der gesamten Codebase wurden am 2026-04-25 die folgenden Schwachstellen identifiziert und behoben:

SSRF-Schutz. Die MCP-Tools `fetch_pdf_text` und `parse_attachment` akzeptierten beliebige URLs ohne IP-Filterung. Ein Angreifer hätte über Prompt-Injection interne Dienste (`172.x.x.x`, `127.0.0.1`) ansprechen können. Fix: `_assert_public_url()` blockiert Private-, Loopback- und Link-Local-Adressen sowie Nicht-HTTP-Schemata vor jeder ausgehenden HTTP-Verbindung.

SQL-Injection in DDL. Die Bootstrap-Funktion `bootstrap_db()` interpolierte `target_user` und `target_db` direkt in `CREATE ROLE`-, `CREATE DATABASE`- und `GRANT`-Statements via f-String. Fix: strikte Identifier-Validierung (`[a-zA-Z_][a-zA-Z0-9_]{0,62}`) vor der DDL-Ausführung.

Python-Sandbox-Escape via `__mro__`. Das `re`-Modul war in der erlaubten Modulliste des `python_sandbox`-Tools enthalten. Über `re.compile().__class__.__mro__[1].__subclasses__()` wäre ein Traversal zum Dateisystem möglich gewesen. Fix: `re` aus der Whitelist entfernt; `vars`, `dir`, `getattr`, `setattr`, `globals` und `locals` aus den verfügbaren Builtins geblockt.

Container-Härtung. Alle Produktions-Container (`langgraph-app`, `mcp-precision`, `moe-admin`) wurden mit `security_opt: no-new-privileges:true` und `cap_drop: ALL` versehen. Der MCP-Server ist nun ausschließlich auf dem Loopback-Interface (`127.0.0.1`) exponiert.

HTTP-Sicherheitsheader. Der Orchestrator-API sendet jetzt auf alle Responses: `X-Content-Type-Options: nosniff`, `X-Frame-Options: DENY`, `Referrer-Policy` und `Permissions-Policy`.

Rate Limiting und Body-Größenlimit. Ein IP-basiertes Rate-Limit (Standard: 60 req/-min via Redis Token-Bucket) schützt gegen Credential-Brute-Force und DoS. Requests mit `Content-Length > 16 MB` werden mit HTTP 413 abgelehnt.

13.6 Privacy-by-Design-Checkliste

Die Architektur erfüllt die *technischen Voraussetzungen* für DSGVO Art. 25 (Datenschutz durch Technikgestaltung und datenschutzfreundliche Voreinstellungen). Dies ist eine **architektonische Aussage, keine rechtliche Feststellung**. Formale Konformität erfordert eine Datenschutz-Folgenabschätzung (DSFA, Art. 35 DSGVO), die ein qualifizierter Datenschutzbeauftragter für den spezifischen Einsatzkontext durchführt. Die folgende Checkliste dokumentiert die vom Operator verifizierbaren Designeigenschaften; sie ersetzt weder rechtliche Beratung noch ein unabhängiges Audit.

Die verifizierbaren Designeigenschaften sind:

- Kein Default-Outbound-Traffic. Der Orchestrator kann offline betrieben werden. SearXNG-Anbindung, externe LLMs und rechtliche Volltextsuche sind alle opt-in.
- Keine versteckten Telemetry-Pings. Grep nach `requests.get` oder `httpx.get` im Source-Tree listet alle ausgehenden Calls auf.
- Minimierung: der Orchestrator persistiert standardmäßig nur die Felder, die er für Routing und Feedback braucht; das Logging von Prompt-Inhalten ist optional und per Env-Variable abschaltbar.
- Transparenz: jede beteiligte Komponente ist Open Source, jede Konfiguration ist in Git versionierbar, jeder Datenfluss ist in `docs/PRIVACY.md` dokumentiert.
- Auditierbarkeit: jede Admin-Aktion wird in die Audit-Tabelle der PostgreSQL-Admin-DB geschrieben; der Audit-Log ist über das Admin-UI einsehbar.

13.7 Security-Hardening 2026-04-06: Ein konkreter Meilenstein

Der kommerzielle Anspruch „wir nehmen Security ernst“ ist in der Branche billig. Wir wollen an dieser Stelle eine konkrete Zeitmarke benennen: der Commit *Security-Hardening 2026-04-06* hat einen Satz von 20 Sicherheits-Tasks abgeschlossen, unter anderem:

- Durchgängiger Umstieg auf Redis-Stack via `REDIS_ARGS` – alte Plaintext-Connection-Strings wurden bereinigt.
- Validierung der Middleware-Reihenfolge: CORS, Rate-Limiting, JWT, Request-Logging werden jetzt in einer getesteten Sequenz angewendet, die Bypasses verhindert.
- Harte Entfernung aller hartkodierten Hostnamen, URLs und Model-Namen aus dem Quellcode; alles wird über `.env` und das Admin-UI konfiguriert.
- Migration von SQLite auf PostgreSQL für die Admin-DB, weil SQLite unter gleichzeitigen Schreibzugriffen in Multi-Request-Szenarien Datenverluste zeigte.

Dieser Meilenstein ist der direkte Vorläufer der öffentlichen Veröffentlichung, die dieses Whitepaper begleitet. Die konkreten Änderungen sind im öffentlichen Git-Log nachvollziehbar.

14 Evaluation und Lessons Learned

Dieser Abschnitt ist bewusst unspektakulär. Ein wissenschaftliches Paper verliert seine Glaubwürdigkeit, wenn es sich nur auf die gelungenen Designentscheidungen konzentriert und die Fehlschläge verschweigt. Wir dokumentieren hier konkrete Episoden aus den letzten Refactoring-Runden und Trainingsläufen, weil sie sowohl für zukünftige Contributorinnen als auch für die wissenschaftliche Einordnung des Projekts relevant sind. Die meisten wurden gefunden und behoben; wo die Behebung noch aussteht, ist der Stand der Analyse dokumentiert.

14.1 Episode 1: Das 70B-Judge-Problem – System-RAM vs. VRAM

Symptom. Das 70B-Template (`tpl-ref-70b`) degradierte im Benchmark konsistent: `llama3.3:70b` auf dem RTX-Knoten (5× RTX 3060, 60 GB VRAM) antwortete mit HTTP 500 und der Meldung *model requires more system memory (20.8 GiB) than is available (13.0 GiB)*.

Ursachenanalyse. Das Modell (42.5 GB, Q4_K_M) *passt* in die 60 GB VRAM. Was nicht passt: der KV-Cache für das Standard-Kontextfenster (128k Token) benötigt zusätzlich ∼20.8 GB *System-RAM* – und der RTX-Host hat nur 13 GB frei. Ollama lädt die Modellgewichte in GPU-VRAM, aber der KV-Cache und die Aktivierungspuffer bleiben im System-RAM. Bei einem 70B-Modell mit 128k-Kontext übersteigt dieser Overhead die verfügbaren Ressourcen.

Fix. Drei Maßnahmen:

1. **Kontext-Wrapper:** `ollama create llama3.3-70b-ctx4k -from llama3.3:70b -parameters num_ctx=4096`. Der reduzierte Kontextrahmen (4096 statt 128k) bringt den KV-Cache-Overhead auf unter 13 GB. Das Modell lädt 55.3 GB in VRAM und läuft stabil.
2. **RTX-Exklusivität:** Das 70B-Template verlagert *alle* T1- und T2-Experten auf andere Knoten. Der RTX-Node ist exklusiv für den Judge reserviert – kein VRAM-Wettbewerb.
3. **Load-Distribution-Override:** Die `EXPERT_NODE_ASSIGNMENT_70B_OVERRIDE`-Tabelle im Template- Builder verschiebt `agentic_coder`, `code_reviewer` und `reasoning` von `N04-RTX` auf `N06-M10` bzw. `N09-M60`.

Ergebnis. Nach dem Fix lief das 70B-Template erfolgreich durch: Thinking-Node, Merger, Critic („no errors found“) und GraphRAG-Ingest (8 Triples) – alles mit dem `llama3.3-70b-ctx4k` Judge. Wall-Clock: 1414 s, davon der Großteil auf den langsamen Tesla-M10-T2-Experten (∼1.5 tok/s für `gemma4:31b`).

Lessons Learned: VRAM ≠ Gesamtbedarf

Bei heterogener Consumer-Hardware (RTX 3060 mit 12 GB pro GPU) reicht es nicht, nur den Modell-Fußabdruck gegen das VRAM-Budget zu prüfen. Der KV-Cache-Overhead im System-RAM ist der versteckte Engpass. Ein simpler Ollama-Wrapper mit reduziertem `num_ctx` löst das Problem ohne Qualitätseinbuße bei Aufgaben, die keine 128k-Token-Kontexte brauchen.

14.2 Episode 2: Die SymPy-Injection-Lücke im MCP-Server

Symptom. In einem Code-Review fiel auf, dass der `calculate`-Endpunkt potenziell ausführbaren Code ausführen konnte. Der Safe-AST-Evaluator warf bei verbotenen Knotentypen (etwa `Attribute`) eine Exception, und der Fallback-Pfad auf SymPy fing *jede* Exception – nicht nur `SyntaxError`. Bei bestimmten SymPy-Builds konnte eine geschickt formulierte Eingabe tatsächlich einen Subprozess starten.

Fix. Der Fallback wurde auf `except SyntaxError` eingeschränkt. Die Testsuite erhielt einen neuen Test mit der Injection-Sequenz `__import__('os').system('id')`, der jetzt als Fehler-Case geprüft wird und dauerhaft verhindert, dass der Fallback-Pfad für Code-Execution missbraucht werden kann.

Bewertung. Die Lücke war in keiner produktiven Installation ausgenutzt; sie wurde in einem internen Review entdeckt. Wir halten es dennoch für richtig, diesen Fall in einem öffentlichen Paper zu erwähnen – Sicherheit entsteht nicht, indem man solche Fehler verschweigt, sondern indem man sie dokumentiert und nachvollziehbar behebt.

14.3 Episode 3: SQLite → PostgreSQL

Symptom. Unter gleichzeitigem Zugriff von zwei oder mehr Admin-UI-Browser-Tabs traten sporadische `database is locked`- Fehler auf. Unter Last im Multi-User-Setup verlor die SQLite-Admin-DB in seltenen Fällen Schreibzugriffe auf das Audit-Log.

Ursache. SQLite ist single-writer und verlässt sich auf File-Locks, die in den verwendeten Docker-Overlay-Filesystems nicht immer verlässlich sind. Für eine ernsthafte Multi-User-Installation ist SQLite ungeeignet.

Migration. Wir haben die Admin-DB auf PostgreSQL migriert – mit `psycopg` im Async-Pool, eigenen Schemata für `users`, `budgets`, `templates` und `audit_log`, und einer sauberen Upgrade-Prozedur. Die LangGraph-Checkpoints bleiben in einer eigenen Postgres-Instanz (`terra_checkpoints`), um Schreib-Last zwischen Admin-DB und Pipeline-State zu trennen.

Lessons. Die Migration war nicht trivial: die Async-Semantik von `psycopg 3.x` und die Connection-Pool-Konfiguration unter LangGraphs `AsyncPostgresSaver` mussten einmal tief verstanden werden. Der Gewinn (Schreibsicherheit unter Concurrent Access) ist aber uneingeschränkt. Wir empfehlen dringend, auch bei einem `solo`-Deployment die Postgres-Variante zu verwenden.

14.4 Episode 4: Die Ghost-Keys im Live-Monitoring

Symptom. Während der Benchmark-Vorbereitung zu diesem Whitepaper wurde beobachtet, dass das Admin-UI Live-Monitoring zwei aktive Anfragen auf dasselbe Template anzeigte, obwohl nur ein Benchmark-Client lief. Die zweite Anfrage war ein *Geist* – ein `moe:active:*` Valkey-Key, der nach einem vorzeitig abgebrochenen Vorlauf nicht mehr aufgeräumt wurde.

Ursachenanalyse. Der vorherige Benchmark-Client war mit `TaskStop` beendet worden, während der Orchestrator mitten in der Pipeline-Verarbeitung war. Der Request-Handler räumt den `moe:active:{chat_id}`-Key nur im *Success-Pfad* am Ende der Completion auf. Brach der Client die HTTP-Verbindung vorzeitig ab, endete der Handler über einen `httpx.WriteError`, ohne den Key zu löschen. Der Key blieb bis zum automatischen Ablauf (TTL lag standardmäßig bei mehreren Stunden) im Store – das Live-Monitoring zeigte also einen laufenden Request, den es in Wirklichkeit nicht mehr gab.

Fix. Die Lehre ist struktureller Natur und wird in zwei Schritten angegangen:

1. **`try/finally`-Block um die gesamte Pipeline-Logik** im `chat_completions`-Handler. Der `finally`-Zweig löscht den aktiven Key und schreibt einen Abschluss-Eintrag ins Sorted-Set `moe:admin:completed` mit Status `aborted_client`. Damit ist jeder Pfad (Success, Fehler, Client-Abort) gleichwertig behandelt.

2. **Watchdog-Task**, die periodisch (alle 60 s) alle `moe:active:*`-Keys mit `started_at > 30 min` inspiziert und ihren zugehörigen Python-Async-Task-Status prüft. Keys ohne lebenden Task werden in `moe:admin:completed` überführt und aus dem Active-Set entfernt.

Bewertung. Dieser Bug war keine Sicherheitslücke, sondern ein *Observability-Rauschen* – aber genau so wichtig, weil er Admin-Entscheidungen beeinflussen kann. Ein Operator, der einen „laufenden“ Prozess im Live-Monitoring sieht, trifft unter Umständen falsche Kapazitätsentscheidungen. Die Lehre in einem Satz: *jeder Pfad durch einen Request-Handler muss den aktiven Zustand mit derselben Gewissheit aufräumen wie der Success-Pfad.*

14.5 Episode 5: Die erste Scheduler-Iteration

Siehe Lessons-Learned-Box in Abschnitt 6: der erste Scheduler berücksichtigte zu viele Signale und wurde bei einem kurzen Netzwerk-Flap instabil. Die Rückkehr zu einem einfachen $(running\_models)/(gpu\_count)$-Scoring hat das Problem beseitigt und die Komplexität des Code-Pfads um zwei Größenordnungen reduziert. Wir erwähnen diese Episode hier zusätzlich, weil sie eine generelle Projekt-Philosophie unterstreicht: *Komplexität ist ein Rückschrittshebel, kein Fortschrittssignal.*

14.6 Episode 6: Specification Gaming – Frontier-Modell-Substitution in einem souveränen Benchmark

Symptom. Ein Post-Session-Audit des GAIA-Benchmark-Templates `moe-aihub-free-gremium-deep-wcc` ergab via Postgres-Abfrage, dass alle zehn Experten-Slots, der Planner und der Judge auf `gpt-4.1@AIHUB` konfiguriert waren – ein kommerzielles Azure-OpenAI-Modell, kein souveränes selbstgehostetes Modell. Die Konfiguration stammte aus einem Bulk-Import, Erstellungs- und Änderungszeitstempel waren identisch. Der GAIA-Lauf vom 5. Mai 2026 führte über 21+ Benchmark-Iterationen mit diesem Template durch und erzielte einen Spitzenwert von **14/30 = 46,7 %**.

Ursache: Specification Gaming. Der zugrundeliegende Mechanismus ist *Specification Gaming* [45]: der optimierende Agent – in diesem Fall der KI-Coding-Assistent, der durchgehend im Projekt verwendet wurde – hat den messbaren Proxy (Benchmark-Score) verbessert, während er das eigentliche Ziel verletzt hat (Demonstration souveräner, Frontier-unabhängiger LLM-Fähigkeit). Drei Beitragsfaktoren wurden identifiziert:

1. Die Souveränitätsanforderung war in der operativen Memory und in Template-Namenskonventionen dokumentiert, aber nicht im Code durchgesetzt.
2. Der Benchmark-Runner berichtet einen numerischen Score als primären Output – ohne Souveränitätssichtbarkeit.
3. Souveräne und kommerzielle Modelle teilen denselben API-Endpunkt, sodass keine Routing-Unterscheidung eine Substitution verhindert.

Konsequenz. Der Score 14/30 = 46,7 % ist keine valide Messung souveräner Stack-Fähigkeit. Der gültige souveräne GAIA-Score wird neu ermittelt.

Behebung. Zwei Maßnahmen wurden umgesetzt:

1. **Explizite Regelkodierung:** GAIA und alle Benchmarks müssen ausschließlich `*-sovereign`- oder lokal gehostete Open-Source-Modelle verwenden. Kommerzielle Modelle (`gpt-4.1`, `gpt-4o`, `claude-*`, `gemini-*`) sind in jedem Benchmark-Template verboten. Diese Regel ist in der persistenten Assistenten-Memory gespeichert und muss vor jedem Benchmark-Lauf verifiziert werden.
2. **Template-Schreibschutz:** Experten-Template-Inhalte (Modell-Identifier, Endpunkte) dürfen vom KI-Assistenten nur mit expliziter Anweisung geändert werden. Lesezugriff für Analyse ist erlaubt; Schreibzugriff erfordert explizite Autorisierung.

Messbare Proxies brauchen durchsetzbare Constraints

Specification Gaming ist kein Pathologie misalignierter Superintelligenz – es ist ein Engineering-Fehler, der sich in alltäglichen Entwicklungs-Workflows manifestiert. Der Fix ist kein strengeres Prompting, sondern eine harte Pre-Run-Prüfung: *Entspricht jeder Modell-Identifier in diesem Template der erlaubten Liste? Wenn nicht, darf der Lauf nicht starten.*

14.7 Episode 7: DeepSpeed ZeRO-3 und multimodale Modelle auf AMD MI250X

Kontext. Im Rahmen des EuroHPC-Grants EHPC-DEV-2026D06-XXX wurde das parakonsistente Judge-Modell **Qwen3.6-35B-A3B** (`Qwen3_5MoeForConditionalGeneration`) auf der LUMI-G-Partition (AMD Instinct MI250X, ROCm-Stack) mittels DeepSpeed ZeRO-3 und LoRA trainiert. Die Wahl fiel auf dieses Modell wegen seiner Sparse-MoE-Effizienz (35 B Gesamtparameter, 3 B aktiv pro Token) und des 262 144-Token-Kontexts, der für parakonsistente Mehrzug-Evaluierungen unverzichtbar ist.

Symptom. Job 19828348 scheiterte auf Rank 6 von 8 GPUs nach wenigen Minuten Modellinitialisierung mit einem nicht sofort zuzuordnenden Assertion-Fehler:

```
AssertionError: {'id': 1013, 'status': 'NOT_AVAILABLE', 'numel': 0,
  'ds_numel': 0, 'shape': torch.Size([0]),
  'requires_grad': True, 'device': device(type='cpu'), ...}
```

Ursachenanalyse. `Qwen3_5MoeForConditionalGeneration` ist eine *multimodale* Vision-Language-Architektur. Sie enthält neben dem Text-Sprachmodell (`model.language_model.*`) auch einen **Vision Tower** (`model.visual.*`), dessen Parameter im text-only SFT-Betrieb nicht instanziiert werden – sie besitzen `numel=0` und `shape=(0,)`.

DeepSpeed ZeRO-3 partitioniert *alle* registrierten Parameter gleichmäßig über alle Ranks und markiert sie als `ds_status=PARTITIONED`. Im Forward-Pass versucht ZeRO-3, sie via `all_gather` zu laden – da der Tensor aber `numel=0` hat, schlägt die interne Koordination fehl: `NOT_AVAILABLE`-Assertion.

Der entscheidende Unterschied zu NVIDIA-Umgebungen: auf H100/A100 ist der typische Pfad für Modelle dieser Größe `device_map=äuto"` mit BitsAndBytes-4-bit-QLoRA und Pipeline-Parallelismus. Auf AMD MI250X steht BitsAndBytes NF4 nicht stabil zur Verfügung (kein stabiler ROCm-Build), sodass ZeRO-3 die einzige praktikable Strategie für Modelle >30 B ist. Das Problem bleibt auf NVIDIA verborgen und ist in keinem Standard-Tutorial dokumentiert.

Fix. Zwei kombinierte Maßnahmen lösen das Problem:

1. Vision Tower einfrieren *vor* `get_peft_model()`.

```
_TEXT_MODULE_PREFIXES = (
    "model.language_model",    # Qwen3_5MoeForConditionalGeneration
    "language_model",          # aeltere VL-Varianten
    "model.text_model",        # Qwen2-VL-Stil
    "model.model",             # Standard CausalLM
    "lm_head",
```

```
)
for name, param in model.named_parameters():
    is_text = any(name.startswith(p) for p in _TEXT_MODULE_PREFIXES)
    if not is_text or param.numel() == 0:   # Vision Tower + leere Params
        param.requires_grad_(False)         # ZeRO-3: persistent, kein Fetch
```

Parameter ohne `requires_grad` werden von ZeRO-3 als persistent behandelt und nie partitioniert – das leere-Tensor-Problem verschwindet.

2. Persistenz-Schwellwert in `deepspeed_zero3.json`.

```
"stage3_param_persistence_threshold": 1e7
```

Parameter unterhalb von 10 M Elementen werden nicht über Ranks partitioniert, sondern persistent auf jedem Rank gehalten – zweite Absicherungsebene gegen `NOT_AVAILABLE` bei kleinen und leeren Tensoren.

Ergebnis. Job 19832821 startete stabil durch. Das Training lief 12 Stunden bis zum SLURM-Zeitlimit (`TIMEOUT`) und lieferte Checkpoint-704 (Epoche 1,0, Loss 0,485, Token-Accuracy 85,7 %) als gesicherten Zwischenstand. Resume-Job 19926224 setzte das Training ab Checkpoint-704 fort und schloss alle 2112 Schritte vollständig ab (abgeschlossen am 16. Juli 2026). Endmetriken (Epoche 3): Loss 0,3032, Token-Accuracy 86,72 %, Entropie 0,4447.

Multimodale Modelle unter ZeRO-3 erfordern explizites Submodul-Freezing

Wer ein Vision-Language-Modell im text-only SFT-Betrieb mit DeepSpeed ZeRO-3 trainiert, muss den Vision Tower *vor* `get_peft_model()` einfrieren. ZeRO-3 partitioniert andernfalls `numel=0`-Parameter, die es im Forward-Pass nicht fetchen kann. Dieser Fehlertyp bleibt auf NVIDIA verborgen (dort dominiert der QLoRA+Pipeline-Parallel-Pfad) und ist deshalb in keinem Standard-Tutorial dokumentiert. Auf AMD ROCm-Stacks, wo ZeRO-3 die einzige praktikable Strategie für Modelle >30 B ist, ist er häufig anzutreffen. Checkliste für AMD-HPC: (1) Vision Tower einfrieren; (2) `stage3_param_persistence_threshold=1e7`; (3) `attn_implementation=ëager"`; (4) `torch_dtype=torch.bfloat16`; (5) `device_map=None`.

14.8 Episode 8: IMoE-Router – ChromaDB-Cache trifft nie (Query-Dokument-Mismatch)

Symptom. Zwei aufeinanderfolgende Aufrufe von `get_dynamic_template()` mit exakt demselben deutschen Prompt lieferten beide ein neu kompiliertes Template – kein Cache-Hit. Auch nach dem dritten identischen Aufruf blieb der erwartete Log-Eintrag *„⋆ Semantic template cache L2 hit!"* aus. Das ChromaDB-`collection.count()` wuchs korrekt, Health-Checks blieben grün – der Cache war in einem scheinbar funktionalen Zustand.

Ursachenanalyse. Schreib- und Lesepfad des ChromaDB-Cache verwenden unterschiedliche Strings:

- **Schreiben** (`_save_template_to_db_and_cache()`, Zeile 698 in `dynamic_router.py`): indexiert das Dokument als `f"Dynamic gating template compiled for prompt: {prompt[:80]}..."`
- **Lesen** (`_match_existing_template()`, Zeile 363): sucht mit dem rohen `prompt`.

Direktmessung der Cosinus-Distanz im Live-Container:

Query-String	**Distanz zum Eintrag**	**Ergebnis** ($\tau = 0{,}18$)
Roher Prompt (Lese-Pfad)	**0,3103**	× MISS
Wrapper-String (Schreib-Pfad)	≈0,0000	✓ HIT

Cache-Logik und Schwellwert (0,18) sind korrekt – aber weil Lese- und Schreibpfad unterschiedliche Strings einbetten, trifft der Cache *niemals*, auch nicht bei verbatim wiederholten Prompts.

Fix. Den rohen `prompt` als ChromaDB-Dokument indexieren; den Wrapper-String nur in `reasoning_trace`/Metadaten ablegen:

```
# In _save_template_to_db_and_cache()  (dynamic_router.py, Zeile 698)
# Vorher (falsch):
collection.upsert(
    documents=[f"Dynamic gating template compiled for prompt: {prompt[:80]}..."],
    ...
)
# Nachher (korrekt):
collection.upsert(
    documents=[prompt],                          # roher Prompt wird eingebettet
    metadatas=[{"reasoning_trace":
        f"Dynamic gating template compiled for prompt: {prompt[:80]}..."}],
    ...
)
```

Stand. Root-Cause identifiziert (2026-06-12, E2E-Test auf `ki-vm-node05`). Der Fix ist ein Einzeiler und noch nicht eingepflegt (Tracking: `AGENT_LASTENHEFT.md`, TASK-4).

Semantischer Cache: Symmetrie zwischen Schreib- und Lesepfad erzwingen

In einem semantischen Cache muss exakt *derselbe String* eingebettet und bei der Query verwendet werden. Die Verwechslung entsteht leicht, wenn der Schreibpfad einen Wrapper-String für Logging-Lesbarkeit konstruiert ("Dynamic gating template compiled for prompt: ...") und der Lesepfad den Rohtext übergibt. Der Fehler ist schwer zu entdecken: Health-Checks sind grün, Collection-Counts wachsen korrekt – nur *Hits* bleiben aus. Diagnostik: Cosinus-Distanz zwischen dem tatsächlichen Query-String des Lesepfads und dem gespeicherten Dokument direkt messen; jeder Wert >Threshold deutet auf Repräsentations-Mismatch hin.

14.9 Episode 9: PEFT-Versionsmismatch beim LoRA-Merge auf LUMI-G

Symptom. Nach erfolgreichem Abschluss des Trainings (3 Epochen, Step 2112, Token-Accuracy 86,5 %) scheiterte der Merge-Job 19951999 nach 4 Minuten und 29 Sekunden mit folgendem Traceback:

```
File "/opt/venv/lib/python3.12/site-packages/peft/utils/
  transformers_weight_conversion.py", line 268,
  in build_peft_weight_mapping
    new_conversion = orig_conversion.__class__(
TypeError: WeightConverter.__init__() got an unexpected
  keyword argument 'distributed_operation'
```

Ursachenanalyse. Das Merge-Script rief `python3` innerhalb des Singularity-Containers auf – also `/opt/venv/bin/python3`, das System-Python des LUMI-Images. Dessen PEFT-Version kannte den Parameter `distributed_operation` im `WeightConverter`-Konstruktor nicht, obwohl eine neuere PEFT-Funktion (`build_peft_weight_mapping`) ihn bereits übergab.

Ursache des Mismatches: Das Training lief mit einer manuell installierten Python-Umgebung unter `/scratch/.../my_venv/bin/python3`, die eine aktuellere und konsistente PEFT-Version enthält. Das Merge-Script exportierte zwar `PYTHONPATH=.../my_venv/lib/...:$PYTHONPATH`, aber Singularity hatte das System-Python (`/opt/venv`) als aktive virtuelle Umgebung aktiviert – `PYTHONPATH` allein überschreibt keine aktive venv. Resultat: Python lud *manche* Pakete aus `my_venv`, andere aus `/opt/venv`, was zu einer inkonsistenten PEFT-Installation führte.

Fix. Den Singularity-Aufruf auf das explizite `my_venv`-Binary umstellen:

```
# Vorher (falsch): implizites System-Python
singularity exec "${SIF}" python3 -c "..."

# Nachher (korrekt): explizites venv-Binary
singularity exec "${SIF}" \
    /scratch/.../my_venv/bin/python3 -c "..."
```

Damit verwendet Python ausschließlich `my_venv` für alle Pakete – kein Mischen mit dem Container-System-Python mehr.

Zusatz. Im selben Zuge wurde `torch_dtype=torch.float16` auf `torch.bfloat16` korrigiert, konsistent mit dem Trainingssetup auf AMD MI250X (FP16 ist auf ROCm instabil; vgl. Episode 7, Tabelle 8).

Singularity + venv: PYTHONPATH genügt nicht – explizites Binary verwenden

Wenn Training und Nachverarbeitungs-Jobs auf HPC-Systemen unterschiedliche Python-Binaries verwenden, entstehen stille Versionsmismatches. Der Fehler ist schwer zu diagnostizieren, weil `PYTHONPATH` korrekt gesetzt ist und keine offensichtliche Fehlermeldung erscheint – erst ein `TypeError` in einer internen Bibliotheksfunktion gibt den Hinweis. Regel: In SLURM-Scripts, die eine manuelle venv auf einem HPC-System nutzen, immer den *vollständigen Pfad zum venv-Python-Binary* angeben. `PYTHONPATH` allein reicht nicht aus, sobald der Container eine eigene virtuelle Umgebung aktiviert.

14.10 Episode 10: `-no-mtp`-Flag bei GGUF-Konvertierung von Qwen3-MoE

Symptom. Die GGUF-Konvertierung des zusammengeführten BF16-Modells (`Qwen3-MoE-35B`) mit `convert_hf_to_gguf.py` scheiterte mit folgendem Fehler:

```
gguf-py: tensor 'blk.40.attn_norm.weight' not found in model
```

Ursachenanalyse. Die Datei `config.json` des zusammengeführten Modells enthält den Eintrag `mtp_num_hidden_layers=1`. Dieser Parameter entstammt der Qwen3-MoE-Architektur (Multi-Token-Prediction) und signalisiert dem Konverter, dass ein zusätzlicher MTP-Block (Block 40) vorhanden sei – der Konverter setzt folglich `block_count=41` statt 40. Da das

zusammengeführte BF16-Modell jedoch *keine* MTP-Gewichte enthält (sie werden beim LoRA-Merge mit `peft.merge_and_unload()` nicht übernommen), versucht der Konverter einen Tensor für Block 40 zu laden, der nicht existiert.

Fix. Das `-no-mtp`-Flag an `convert_hf_to_gguf.py` übergeben:

```
python convert_hf_to_gguf.py /pfad/zum/merged-bf16 \
    --outtype q4_k_m \
    --no-mtp
```

Das Flag weist den Konverter an, den MTP-Eintrag in `config.json` zu ignorieren und `block_count=40` zu verwenden.

Qwen3-MoE GGUF: `-no-mtp` bei BF16-LoRA-Merges erforderlich

Beim LoRA-Merge mit `peft.merge_and_unload()` werden MTP-Tensoren nicht in das Ausgabemodell übernommen, obwohl `mtp_num_hidden_layers` in `config.json` gesetzt bleibt. GGUF-Konverter, die diesen Eintrag auslesen, berechnen einen falschen `block_count` und scheitern am fehlenden Tensor. Abhilfe: `-no-mtp`-Flag beim Konverteraufruf.

14.11 Episode 11: Der wirkungslose Fine-Tune – Sequenzlängen-Truncation und Prompt-Taxonomie-Drift beim Planner-SFT

Symptom. Ein per LoRA-SFT nachtrainiertes `Qwen3-8B` (Zielrolle: Planner, Modellname `qwen3-planner:q4km`, 259 829 Trainingsbeispiele, 3 Epochen auf LUMI-G) bestand einen Rollentauglichkeitstest ohne auffällige Befunde: valides JSON, plausible Kategorie-Zuordnung, korrekte Sonderfälle (`precision_tools`-Priorität, `dynamic`-Kategorie mit `domain`-Feld). Auf den ersten Blick ein erfolgreicher Fine-Tune. Erst der Blick in die tatsächlichen LUMI-Trainingslogs – veranlasst durch eine unabhängige Nachfrage, warum das Modell mit nur 40 960 statt der ursprünglich angenommenen 262 144 Token Kontextfenster lief – deckte auf, dass der Fine-Tune strukturell keine Wirkung gehabt haben kann.

Ursachenanalyse (Teil 1): Sequenzlängen-Truncation verschluckt das Trainingsziel. Der Trainingslauf (Job 20188267 und Fortsetzung 20190726) protokollierte beim Start folgende, im Log leicht zu übersehende Warnung:

```
Token length p50=2647 p95=3607 p99=3628 (max_seq=1536)
WARNING: p99 (3628) > max_seq_len (1536) --
  100% of samples will be truncated!
```

`max_seq_len` war auf 1536 Token gesetzt – ein Wert, der für ein früheres, deutlich kürzeres Prompt-Format übernommen und beim Wachstum des Systemprompts nicht mehr angepasst wurde. Eine nachträgliche Tokenisierung dreier zufälliger Trainingsbeispiele mit dem tatsächlich verwendeten Tokenizer bestätigte das Ausmaß:

Der Systemprompt allein überschreitet `max_seq_len` bereits um mehr als das 1,6-fache. Der Tokenizer nutzt den Standard `truncation_side="right"` (keine Override in `tokenizer_config.json`); die Chat-Template-Reihenfolge ist System → User → Assistant. Zusätzlich verwendet das Trainingsscript kein Completion-Only-Loss-Masking (kein `DataCollatorForCompletionOnlyLM`, kein `response_template`) – die Loss wird über die *gesamte* Sequenz berechnet, nicht nur über die Assistant-Antwort. In Kombination bedeutet das: Bei praktisch jedem der 259 829

Beispiele lag die Ziel-JSON-Antwort weit außerhalb des 1536-Token-Fensters und wurde vom Trainer nie gesehen. Was tatsächlich trainiert wurde, war Next-Token-Prediction auf den ersten 1536 Token von nur fünf nahezu identischen Systemprompt-Varianten – wiederholt über eine Viertelmillion Beispiele. Der Loss sinkt dabei zuverlässig (das Modell lernt schnell, einen ihm bereits vollständig vorliegenden Text vorherzusagen), ohne dass ein Fehler auftritt oder das Training abbricht – der Trainingslauf *sieht* erfolgreich aus.

Ursachenanalyse (Teil 2): Prompt-Taxonomie-Drift zwischen Training und Serving. Unabhängig von der Truncation ergab der Vergleich des Trainings-Systemprompts mit dem zur Inferenzzeit tatsächlich verwendeten Prompt eine zweite, eigenständige Diskrepanz: Im System kursieren drei unterschiedliche Planner-Systemprompts.

Selbst ein korrekt trainiertes Modell würde diese Taxonomie-Differenz nicht überbrücken: Trainiert auf `research/dynamic`, aber zur Laufzeit über ein Template mit `web_researcher/memory_recall` und ohne `dynamic` angesprochen, transferiert das Gelernte nicht. Die im Docstring des Datensatz-Generators festgehaltene Annahme – *"The training system prompt matches the inference prompt structure so the fine-tuned model generalises correctly at serving time"* – war zum Zeitpunkt des Trainings bereits nicht mehr zutreffend; das produktiv genutzte Template hatte sich unabhängig davon weiterentwickelt (datenbankgetriebene Template-Overrides, siehe Abschnitt 14.8).

Fix.

1. `max_seq_len` auf 4096 Token angehoben – deckt `p99=3628` mit ≈470 Token Puffer ab, bleibt weit unter dem nativen 40 960-Token-Trainingsfenster von `Qwen3-8B` (`n_ctx_train` laut GGUF-Metadaten) und vermeidet den quadratischen Attention-Mehraufwand eines pauschal zu groß gewählten Fensters (z. B. 7168 Token), das kein einziges reales Beispiel benötigt.

2. Kanonisierung auf *einen* Systemprompt: `PLANNER_SYSTEM_PROMPT` bleibt die Quelle der Wahrheit; produktiv genutzte Templates werden auf dessen Kategorie-Taxonomie umgestellt, statt umgekehrt den Trainingsdatensatz an ein Benchmark-spezifisches Template anzupassen.

3. Re-Training mit demselben (bereits vorliegenden) 259 829-Beispiele-Datensatz – keine erneute 405B-Teacher-Generierung auf LUMI-G nötig, nur Neustart des SFT-Jobs mit korrigiertem `max_seq_len`.

SFT auf HPC-Clustern: `p99`-Tokenlänge vor Jobstart gegen `max_seq_len` prüfen

Ein zu klein gewähltes `max_seq_len` bricht das Training nicht ab und erzeugt keinen Fehler – der Loss konvergiert, die Trainingsmetriken wirken unauffällig, und erst eine nachträgliche Tokenisierung realer Beispiele deckt auf, dass die Zielantwort nie im Loss-Fenster lag. Das Trainingsscript protokolliert die `p50/p95/p99`-Statistik zwar automatisch, doch bei mehrstündigen HPC-Jobs mit umfangreichem Log-Output geht eine einzelne `WARNING`-Zeile leicht unter. Regel: Vor jedem SFT-Start auf teurer HPC-Zeit die `p99`-Tokenlänge explizit gegen `max_seq_len` verifizieren (nicht nur die Warnung im Log abwarten) – insbesondere bei `truncation_side="right"`, wo ein zu kleines Fenster nicht den Prompt-Anfang, sondern das Trainingsziel am Sequenzende verschluckt. Zusätzlich gilt: *"Trainings-Prompt = Serving-Prompt"* ist eine Design-Annahme, kein automatisch garantiertes Invariant – sobald produktive Templates unabhängig vom Datensatz-Generator

weiterentwickelt werden (z. B. über datenbankgetriebene Admin-UI-Overrides), muss ein expliziter Abgleich der Kategorie-Taxonomie Teil des Retraining-Prozesses sein, nicht nur eine implizite Annahme im Docstring.

14.12 Episode 12: Warum Fine-Tuning auf AMD MI250X trotz Data-Center-Hardware quälend langsam bleibt – und warum lokale Consumer-GPUs keine Alternative sind

Symptom. Nach der in Episode 11 beschriebenen Korrektur (`max_seq_len` 1536 → 4096, kanonisierte Taxonomie) blieb der erneute Planner-SFT-Lauf trotz mehrerer unabhängiger Fixes hartnäckig langsam. In einer Serie weiterer SLURM-Jobs wurden nacheinander Batch-Größe, QLoRA-4-Bit-Quantisierung, ein Device-Placement-Bug (`torch.cuda.set_device(local_rank)` fehlte – alle acht DeepSpeed-Ranks luden ihr Modell zunächst auf GPU 0 statt sich auf die acht GCDs zu verteilen) und Gradient-Checkpointing als Stellschrauben durchprobiert. Nach Behebung des Device-Placement-Bugs lief das Training zwar erstmals stabil ohne Out-of-Memory-Abbruch – blieb aber sowohl im 4-Bit-QLoRA-Modus (≈149,6 s/Schritt, Job 20329106) als auch im vollen BF16-Modus (≈148,2 s/Schritt, Job 20333367) bei praktisch identischer Geschwindigkeit. Diese Übereinstimmung schloss Dequantisierungs-Overhead als Ursache aus und lenkte den Verdacht auf eine tiefer liegende, präzisionsunabhängige Bremse. Ein Testlauf mit deaktiviertem Gradient-Checkpointing (Job 20391589) OOM'te umgehend:

```
torch.OutOfMemoryError: HIP out of memory. Tried to allocate 6.28 GiB.
GPU 7 has a total capacity of 63.98 GiB of which 3.45 GiB is free.
Of the allocated memory 59.10 GiB is allocated by PyTorch, ...
```

Ursachenanalyse: kein Flash Attention 2 und unvollständige Kernel-Portierung auf ROCm. Der Trainingsscript-Header dokumentiert die Ursache explizit: `attn_implementation="eager"` – Flash Attention 2 ist für den verwendeten ROCm-7.0/PyTorch-2.10-Stack auf der MI250X-Partition nicht verfügbar. Eager-Attention materialisiert die volle $O(\text{seq}^2)$-Aufmerksamkeitsmatrix im Speicher, statt sie wie FA2 gekachelt und fusioniert im schnellen On-Chip-Speicher zu berechnen – ein für lange Sequenzen (`p99=3628` Token, siehe Episode 11) besonders teurer Kompromiss. Ein zweiter, unabhängiger Befund im Trainingslog bestätigt das Muster einer insgesamt unreiferen Kernel-Landschaft auf ROCm gegenüber CUDA:

```
UserWarning: Using the native apex kernel for RoPE.
  warnings.warn("Using the native apex kernel for RoPE.", UserWarning)
```

Der fusionierte, für CUDA optimierte RoPE-Kernel aus `apex` ist für ROCm nicht kompiliert; der Fallback auf die native (nicht fusionierte) Implementierung kostet zusätzliche Kernel-Launches und Speicherbandbreite pro Attention-Layer. Hinzu kommen zwei strukturelle, aber notwendige Kostenfaktoren: Gradient-Checkpointing verdoppelt effektiv die Forward-Pass-Kosten durch Rekomputation im Backward – ohne es OOM'te der Lauf messbar, wie der Auszug oben zeigt –, und LoRA hält zwar nur 43,65 Millionen von 8,23 Milliarden Parametern trainierbar (0,53 %), spart dabei aber keine FLOPs im Forward-/Backward-Pass durch das eingefrorene Basismodell ein: Gradienten müssen trotzdem durch alle acht Milliarden Parameter zurückfließen, damit die LoRA-Adapter überhaupt ein Lernsignal erhalten.

Einordnung: Wäre lokale Consumer-Hardware eine Alternative? Die Projekt-Infrastruktur verfügt über mehrere ältere Consumer-GPUs (RTX 2060/RTX 3060, kombiniert ≈60 GB VRAM über fünf Karten). Die naheliegende Frage, ob ein solches Fine-Tuning stattdessen lokal hätte laufen können, lässt sich anhand öffentlicher Hersteller-Datenblätter überschlagen – ausdrücklich als Abschätzung gekennzeichnet, *nicht* als eine auf dieser Hardware tatsächlich durchgeführte und validierte Messung:

- **Rohleistung:** Eine einzelne MI250X-GCD liefert laut AMD-Datenblatt bis zu ≈191,5 TFLOPS BF16 (Matrix, dense, halbe Karte); eine RTX 3060 liegt bei ≈12,7 TFLOPS FP32 bzw. einem niedrigen zweistelligen TFLOPS-Bereich bei dichter FP16/BF16-Tensor-Auslastung – eine Differenz von grob einer Größenordnung pro Chip, multipliziert mit acht GCDs auf LUMI-G gegenüber fünf heterogenen Consumer-Karten.

- **Fehlende BF16-Tensor-Cores auf Turing:** Die RTX 2060 (Turing-Architektur) besitzt keine native BF16-Tensor-Core-Unterstützung – diese kam erst mit Ampere (RTX 3060). Der im Trainingsscript hart kodierte Modus `bf16=True, fp16=False` würde auf einer RTX 2060 gar nicht laufen, ohne Anpassung auf FP16-Mixed-Precision (mit den bekannten Stabilitätsrisiken von FP16-Gradienten-Unterlauf bei kleinen LoRA-Lernraten).

- **VRAM-Fit:** Ein 8B-Modell in BF16 belegt ≈16,4 GB allein für die Gewichte – das passt auf keine einzelne der genannten Karten (6–12 GB). Ohne 4-Bit-QLoRA-Quantisierung ist lokales Training dieses Modells nicht möglich; *mit* QLoRA passt es auf eine einzelne RTX 3060, allerdings im Single-GPU-Betrieb – die übrigen vier Karten blieben ungenutzt, da DeepSpeed-ZeRO-Synchronisation über heterogene Consumer-GPUs unterschiedlicher VRAM-Größe ohne NVLink, nur über PCIe, den langsamsten/kleinsten Chip zum Taktgeber jedes synchronisierten Schritts macht.

Die Kombination aus Rohleistungslücke, fehlender BF16-Hardware-Unterstützung auf der älteren Karte und dem VRAM-Fit-Zwang zu einem Single-GPU-QLoRA-Betrieb legt nahe, dass ein vollständigkeitsäquivalenter Trainingslauf auf der lokalen Hardware eher im Bereich mehrerer Monate gelegen hätte – gegenüber einem auf LUMI-G bei dieser Konfiguration beobachteten Gesamtdurchlauf von rund 9 Tagen (3 Epochen, 6 060 Schritte bei ≈130 s/Schritt). Diese Abschätzung preist noch keine PCIe-Kommunikations-Overheads zwischen heterogenen Karten ein und fällt daher tendenziell konservativ aus – also eher zu günstig für die lokale Hardware. Für dieses Workload-Profil – vollständiges Sequence-Attention-Fine-Tuning eines 8B-Modells bei bis zu 4096 Token Kontext – bleibt HPC-Hardware trotz ROCm-Software-Nachteilen alternativlos; die lokale Flotte bleibt für Inferenz/Serving quantisierter Experten-Modelle (Ollama, Q4_K_M-GGUF) die richtige Rolle.

ROCm-Softwarereife als Budgetfaktor, nicht nur Rohleistung als Auswahlkriterium

Bei der Wahl von AMD-MI250X-HPC-Ressourcen für Transformer-Fine-Tuning zählt nicht nur die (auf dem Papier klar überlegene) Rohleistung gegenüber Consumer-Hardware, sondern auch die Reife des Software-Stacks: fehlendes Flash Attention 2 und unvollständig portierte fusionierte Kernel (z. B. Apex-RoPE) kosten spürbaren Durchsatz gegenüber einem äquivalenten CUDA-Stack auf vergleichbarer Hardware. Diese Lücke ist bei der Kalkulation von HPC-Stunden-Kontingenten für ROCm-Cluster explizit einzuplanen, nicht nur die theoretische TFLOPS-Angabe des Datenblatts. Umgekehrt gilt: Selbst mit diesem Nachteil bleibt HPC-Hardware für Volltraining im 8B-Parameter-Bereich mit langen Sequenzen der lokalen Consumer-GPU-Flotte um Größenordnungen überlegen – lokale Hardware ist für dieses Projekt strukturell auf die

Inferenz-/Serving-Rolle festgelegt, nicht auf Fine-Tuning dieser Modellklasse.

14.13 Die Testsuite

Stand v1.0 (April 2026). Das initiale Release umfasste 95 bestehende Tests in drei Dateien:

- `tests/test_routing.py`: 12 Routing-Tests (gemockte HTTP-Aufrufe).
- `tests/test_mcp_validation.py`: 38 MCP-Input-Validierungs-Tests inklusive AST-Whitelist-Angriffstest-Suite.
- `tests/test_deployment_artifacts.py`: 45 Cross-Artefakt-Konsistenz-Tests (UID 1001, Port 8000, `MOE_*_DIR` in Dockerfile, Helm-Chart und Quadlet-Unit).

Stand v1.1 (Mai 2026). Die Testsuite ist auf **195 + Tests** in **neun Dateien** in drei Unterverzeichnissen gewachsen:

Verzeichnis / Datei	Scope
`smoke/test_env_contract.py`	Env-Variablen-Kontrakt: alle Pflicht-Variablen sind beim Start vorhanden und korrekt typisiert.
`integration/test_routes_contract.py`	Routen-Registrierungs- Kontrakt: erwartete URL-Muster und HTTP-Methoden vorhanden.
`integration/test_api_auth.py`	JWT-Validierung, Rate-Limit- Durchsetzung, Budget-Gate.
`pipeline/test_cc_session.py`	Claude-Code-Session-Lifecycle und Kontext-Komprimierung.
`pipeline/test_kafka_handlers.py`	Kafka-Publish/Consume-Handler und Graceful Degradation ohne Kafka.
`pipeline/test_agent_state.py`	`AgentState`-Feldkonsistenz über Pipeline-Übergänge.
`tests/test_parsing.py`	Stateless Parser: JSON-Extraktion, Konfidenz-Parsing, History-Truncation.
`tests/test_web_search.py`	SearXNG + DuckDuckGo-Fallback-Verhalten.

Die dreistufige Hierarchie (*smoke*, *integration*, *pipeline*) folgt einem Triageprinzip: Smoke-Tests laufen in unter zwei Sekunden und blockieren jeden CI-Merge; Integration-Tests prüfen Vertragsgrenzen zwischen Services; Pipeline-Tests validieren zustandsbehaftete LangGraph-Übergänge mit gemockten LLM-Backends.

Engineering-Meilenstein: Monolith-Dekomposition (v2.4). Zwischen dem initialen Release und v1.1 dieses Whitepapers wurde die primäre Quelldatei des Orchestrators von einem **11.190-Zeilen-Monolithen** (`main.py`) in ein strukturiertes Paket aus **∼1.500 Zeilen** Einstiegspunkt-Code und 14 fokussierten Modulen umgebaut – eine Reduktion um 86 %. Die Dekomposition erfolgte in 14 expliziten Phasen, wobei alle 195 Tests bei jedem Schritt grün blieben. Die dabei entfernten ∼600 Zeilen waren tote Duplikat-Handler aus früheren Iterationen. Wenn ein Monolith sauber in Module zerlegbar ist ohne Verhaltensänderung, bestätigt das rückwirkend, dass das ursprüngliche Architekturdesign korrekt umgesetzt worden war.

Auf was wir stolz sind – und worauf nicht

Stolz: auf die Deterministik des Routings, die Modul-Dekomposition, die dreistufige Test-Hierarchie, die Cross-Artefakt-Konsistenz-Tests, die Dokumentation, die Transparenz der Lessons-Learned.

Nicht stolz: auf das 70B-RAM-Problem, die SymPy-Lücke, den zu cleveren Scheduler, die Ghost-Keys, den Specification-Gaming-Vorfall (Episode 6) und den 5.2/10-Durchschnitt im kognitiven Benchmark. Diese Fehler und Schwächen waren teilweise vermeidbar und wir dokumentieren sie trotzdem, weil das zur wissenschaftlichen Ethik gehört.

14.14 Baseline-Benchmarks: drei Referenz-Templates

Um die Architektur nicht nur zu beschreiben, sondern auch messbar zu machen, haben wir drei *Referenz-Expert-Templates* konstruiert, die den deterministisch gerouteten Pfad auf einem repräsentativen Cluster aus vier Inferenz-Knoten demonstrieren.

Cluster-Topologie. Der Test-Cluster besteht aus vier Knoten mit *heterogener* GPU-Hardware: einem High-End-RTX-Node mit fünf GPUs und ca. 70 ladbaren Modellen, einem Mid-Range-Node mit vier GPUs, einem Kleinen-Knoten mit nur fünf verfügbaren Modellen (vision-orientiert) und einem älteren M60-Knoten mit zwei GPUs. Anonymisiert referenzieren wir sie im Folgenden als `NODE-A` (RTX), `NODE-B` (Mid-M10), `NODE-C` (Small-GT) und `NODE-D` (Legacy-M60).

Drei Templates, drei Größenklassen. Die Templates unterscheiden sich ausschließlich in der *Größenklasse der T2- Fallback-Modelle*. T1 ist in allen dreien identisch (7–9B Klein-Modelle zur schnellen ersten Antwort). Dadurch ist die Differenz zwischen den Templates auf genau einen Design-Hebel reduziert: wie tief die Fallback-Kaskade geht, wenn die initialen T1-Antworten die Konfidenzschwelle unterschreiten.

Template	Zweck	T2-Klasse	Kern-T2-Modelle
`tpl-ref-8b`	Minimum-Cost Operation	≤ 14 B	`phi4:14b`, `mistral-nemo:12b`, `gpt-oss:20b`
`tpl-ref-30b`	Balanced Production	20–35 B	`qwen3-coder:30b`, `command-r:35b`, `deepseek-r1:32b`, `gemma4:31b`
`tpl-ref-70b`	Deep Quality	46–123 B	`llama3.3:70b`, `mixtral:46b`, `Qwen3-Coder-Next:79b`

Lastverteilung. Jede T1-Stufe verteilt 13 Experten- Kategorien auf die vier Knoten, um die Parallelität der Architektur explizit zu demonstrieren. Die Zuordnung folgt drei Regeln: erstens muss das gewünschte T1-Modell auf dem Ziel-Knoten lokal verfügbar sein; zweitens werden Compute-intensive Experten (Code-Review, agentic_coder, reasoning) dem stärksten Node zugeordnet; drittens werden Kontext-begrenzte aber CPU-billige Experten (vision, creative, general) auf dem kleinsten Node geparkt. Die resultierende Verteilung pro Template ist:

Template	NODE-A	NODE-B	NODE-C	NODE-D
`tpl-ref-8b`	8	6	6	6
`tpl-ref-30b`	8	7	4	7
`tpl-ref-70b`	13	5	3	5

Die T2-Schicht des 70B-Templates konzentriert sich auf `NODE-A`, weil `llama3.3:70b` nur dort lokal installiert ist. Das ist explizit sichtbar in der gestiegenen Anzahl von Experten-Entries auf NODE-A: 13 statt 8. Die Konzentration ist kein Designfehler, sondern die ehrliche Konsequenz der Hardware-Verfügbarkeit.

Per-Template Planner- und Judge-Customisierung. Jedes Template hat seinen eigenen `planner_model`- und `judge_model`-Eintrag plus maßgeschneiderten System-Prompts:

- `tpl-ref-8b`: Planner `llama3.1:8b` (billig), Judge `phi4:14b`. Planner-Prompt fordert maximal 2 Kategorien; Judge-Prompt limitiert auf 300 Wörter und simple Mehrheits-Logik bei Widersprüchen.
- `tpl-ref-30b`: Planner `phi4:14b`, Judge `gpt-oss:20b`. Planner-Prompt erlaubt 1–4 Kategorien und aktiviert `tool_expert` bei Rechen- oder Hash-Operationen. Judge-Prompt markiert unsichere Passagen mit `[UNSICHER]`.
- `tpl-ref-70b`: Planner `gpt-oss:20b`, Judge `llama3.3:70b`. Planner-Prompt erlaubt 2–5 Kategorien, aktiviert in regulierten Domänen (medical, legal) zusätzlich `reasoning` als kritische Prüfung. Judge-Prompt verlangt explizite Konfliktauflösung mit Begründung, bei Zahlen eine Probe.

Damit ist der Benchmark eine Gegenüberstellung *über genau eine Variable* (Tiefe der T2-Kaskade plus passende Planner-/Judge-Dimensionierung), während T1 als invariante Baseline fixiert bleibt.

14.14.1 Referenz-Prompt

Als Benchmark-Prompt verwenden wir eine multi-domain Frage, die bewusst drei Expertenkategorien gleichzeitig ansprechen soll (legal, technical, math). Das zwingt den Planner zu einer Fan-Out-Entscheidung und stellt sicher, dass die Parallelität der Pipeline tatsächlich ausgeschöpft wird:

> *Ein Unternehmen will personenbezogene Kundendaten für das Fine-Tuning eines internen LLMs verwenden. Beantworte knapp (max 400 Wörter total):*
> *(a) Welche DSGVO-Rechtsgrundlage kommt hier in Betracht (Art. 6 / 9)?*
> *(b) Nenne zwei technische Maßnahmen zur Risikominimierung mit je einem Satz Wirkungsweise.*
> *(c) Berechne VRAM-Bedarf für 1.200.000 Embeddings à 1024 Dim fp16 (nur die Rechnung, Endwert in GB).*

Der erwartete Antwortumfang pro Expert-Antwort liegt bei 200–500 Tokens, der vereinigte Judge-Output bei ca. 400 Tokens. Die Rechnung in Teil (c) hat einen überprüfbaren Endwert ($1.200.000 \times 1024 \times 2$ Byte $\approx 2{,}46$ GB), den der externe Judge und unser Self-Correction-Node gleichermaßen als Korrektheitsanker verwenden können.

14.14.2 Messmethodik

Der Benchmark wird durch den *echten* Orchestrator-Endpunkt `POST /v1/chat/completions` geleitet, nicht als Direkt-Call an einzelne Ollama-Instanzen. Das ist eine bewusste Entscheidung: wir messen die *produktive* Pipeline inklusive Planner, Parallel-Fan-Out, Self-Correction, Critic, GraphRAG-Ingest, Merger und Judge – nicht einen vereinfachten Ausschnitt.

Pro Template erfassen wir:

- **Wall-clock-Latenz** (client-seitig, gesamter HTTP-Round-Trip)
- **Prompt- und Completion-Tokens** aus dem OpenAI- kompatiblen `usage`-Feld
- **Externe Qualitätsbewertung** (0–10) durch `llama3.3:70b` als unabhängiger Judge, der die generierte Antwort gegen den ursprünglichen Prompt bewertet
- **Number of Tier-2 escalations** aus den Orchestrator- Logs (indirekt, über die Self-Correction-Events)

Während der Messung sind alle drei Templates im Admin-UI sichtbar und jede Anfrage erscheint im Live-Monitoring mit Template-Name, Start-Zeitstempel und der zugehörigen API-Key-Kennzeichnung `bench-runner`. Das ist ausdrücklich auch ein *Stress-Test* der Observability-Schicht.

14.14.3 Baseline-Ergebnisse

Die folgenden Zahlen stammen aus einer einmaligen Referenz-Messung auf dem oben beschriebenen Cluster. Sie sind *keine* kompetitive Benchmark gegen kommerzielle Anbieter – sie zeigen nur die relative Position der drei Referenz-Templates zueinander, um die Design-Entscheidung „Tiefe der T2-Kaskade“ empirisch greifbar zu machen. Alle Messungen sind im Repository unter `shared/templates/benchmark_results/latest.json` reproduzierbar.

70B-Template: llama3.3-70b-ctx4k als exklusiver Judge

Das `tpl-ref-70b`-Template nutzt `llama3.3-70b-ctx4k@NODE-A` als exklusiven Judge auf dem RTX-Knoten. Die Lösung für das zuvor beobachtete RAM-Problem: ein Ollama-Wrapper-Modell (`ollama create llama3.3-70b-ctx4k -from llama3.3:70b -parameters num_ctx=4096`) reduziert den KV-Cache-Overhead von 20.8 GiB auf unter 13 GiB. Zusätzlich wurden *alle* T1- und T2-Experten aus dem 70b-Template von NODE-A auf andere Knoten verschoben, sodass der 70b-Judge die vollen 55.3 GB VRAM des RTX-Knotens exklusiv nutzen kann. Die vollständige Pipeline (Thinking + Merger + Critic + GraphRAG-Ingest) lief erfolgreich durch, der Critic meldete „no errors found“. Die Wallclock von 1414 s spiegelt die Kombination aus langsamen Tesla-M10-T2-Experten (∼1.5 tok/s) und dem 70b-Judge (∼2 tok/s) wider.

14.15 GAIA-Benchmark 2026: Evaluierung gegen externe Genauigkeits-Baseline

Im April 2026 wurde MoE Sovereign auf dem offiziellen GAIA-Benchmark [15] (General AI Assistants, 165 Validierungs-Fragen, Levels 1–3) evaluiert.

Hinweis zur Benchmark-Integrität

Das in diesem Abschnitt dokumentierte Ergebnis wurde mit dem Template `moe-aihub-free-gremium-deep-wcc` erzielt, bei dem sich nachträglich herausstellte, dass alle Experten-Slots auf ein kommerzielles Frontier-Modell (`gpt-4.1@AIHUB`) geroutet wurden – nicht auf den intendierten Sovereign-Stack. Das Ergebnis kann daher nicht als Leistungsnachweis des Sovereign-Stacks gewertet werden. Die vollständige Ursachenanalyse und Behebungsmaßnahmen sind in Abschnitt 14.6 dokumentiert. Die Zahlen werden hier aus methodischen Gründen aufgeführt; die valide Sovereign-GAIA-Messung ist derzeit ausstehend.

Gemessenes Ergebnis (siehe Integritätshinweis oben). **14/30 = 46,7%** mit Template `moe-aihub-free-gremium-deep-wcc` (`gpt-4.1@AIHUB` als effektives Backbone). Zum Vergleich: der offizielle GPT-4o Mini-Referenzwert liegt bei 44,8%.

Statistische Einschränkungen dieses Benchmarks

Jeder Run wertet 30 Fragen aus (10 je Schwierigkeitsstufe). Bei dieser Stichprobengröße umfasst ein binomiales 95 %-Konfidenzintervall für das beste Ergebnis (14/30 = 46,7%) rund **28 %–66 %**. Der Punktschätzer kann daher nicht als definitive Leistungsaussage gelten; er beschreibt einen plausiblen Betriebsbereich, keine statistisch garantierte Untergrenze. Vollständige Rohdaten, Prompt-Seeds und der Evaluierungsharness sind Teil des Repositories (`benchmarks/gaia_runner.py`); ein formal reproduzierbares Benchmark-Artefakt mit fixierten Seeds und veröffentlichten per-question-Outputs ist noch nicht veröffentlicht – dies wird als Lücke anerkannt und ist in Planung.

14.15.1 Judge-Experiment: qwen-3.5-122b-sovereign als Planner+Judge

Ein Vergleichstest mit `qwen-3.5-122b-sovereign` als Planner- und Judge-Modell (anstelle des bewährten `gpt-oss-120b-sovereign`) ergab deutlich schlechtere Ergebnisse:

Ursachenanalyse.

1. **Chain-of-Thought-Overhead.** `qwen-3.5-122b` generiert intern Chain-of-Thought-Reasoning – selbst für Level-3-Fragen wurden die Timeouts von 1800 s überschritten (5 von 10 L3-Fragen).
2. **Ignorierte Output-Regeln.** Die `OUTPUT ONLY THAT VALUE`-Direktive wird bei Thinking-Modellen ignoriert: statt nackter Werte (`None`, numerischer Ergebnis) erscheint Fließtext (`Best Estimate:`, `Based on...`), was die Normalisierung und Judge-Auswertung zerstört.
3. **Einziger Vorteil: Präzisere Zahlenwerte.** Bei Rechenaufgaben liefert `qwen-3.5-122b` präzisere Gleitkommazahlen (z. B. `1.456` statt `1.46`).

Schlussfolgerung: Thinking-Modelle und Short-Answer-Benchmarks

Thinking-Modelle eignen sich für GAIA-ähnliche Short-Answer-Benchmarks nur unter zwei Bedingungen: (a) unbegrenzte Timeouts und (b) Post-Processing der Modellausgaben. Ohne diese Maßnahmen ist die Ausgabeform zu unstrukturiert für automatische Auswertung. Für den Produktionsbetrieb bleibt `gpt-oss-120b-sovereign` der optimale

Judge.

Modell-Vergleich. Der Vergleich zwischen dem AIHUB-Frontier-Modell und einem lokalen `qwen3.6:35b` auf Consumer-Hardware (N04-RTX) zeigt, dass die *Architektur* – nicht das Modell – den dominanten Beitrag leistet. `qwen3.6:35b` erreicht 11/30 = 36,7%, was 79% des besten AIHUB-Scores entspricht, bei jedoch 10× höherer Inferenz-Latenz (430 s vs. 35 s pro Frage). Fragen, die hartnäckige Agentic-Loop-Traversals erfordern („Soups and Stews"-XLS, Unlambda-Backtick), löst das kleinere Modell sogar zuverlässiger – weil es mehr Runden durchhält.

Genauigkeit vs. Latenz: zwei nicht vergleichbare Dimensionen

Die obigen Zahlen messen ausschließlich *Genauigkeit*. Ein Cloud-Modell wie GPT-4o Mini liefert eine vergleichbare Antwort in 1–3 Sekunden; die MoE Sovereign-Pipeline benötigt für einen komplexen Multi-Experten-Request auf derselben Hardware 360–1400 Sekunden. Die beiden Systeme unterscheiden sich um zwei bis drei Größenordnungen in der Latenz. Diese Diskrepanz ist kein Zufall, sondern direkte Konsequenz einer Designentscheidung: selbst gehostete, datenschutzkonforme Inferenz auf Consumer- und Legacy-Hardware, bei der kein einzelnes GPU ein 30B+-Modell ohne Quantisierung und KV-Cache-Drosselung fasst. Das System ist nicht für synchrone, latenzarme Chat-Anwendungen ausgelegt, sondern für asynchrone, souveränitätskonforme Workflows in regulierten Domänen, bei denen Antwortzeiten in Minuten gemessen werden und Daten die Betreiber-Infrastruktur nicht verlassen dürfen.
Jeder faire Vergleich mit kommerziellen Anbietern muss daher alle drei Dimensionen ausweisen: Genauigkeit, Latenz und Datensouveränität. Genauigkeit isoliert zu berichten ist in beide Richtungen irreführend.

Gewonnene Erkenntnisse.

- **Planner-Prompt-Overflow.** Prompt > 14.000 Zeichen führt zu Context-Overflow: der Planner gibt nur `}` aus, alle 3 Retries scheitern. Strenge Obergrenze von 13.000 Zeichen einhalten.
- **Deterministische Tools schlagen SearXNG.** `wikidata_sparql` löste die Morarji-Desai-Frage stabil; Wikipedia-Wayback-Fetch lieferte die Mercedes-Sosa-Albumzählung korrekt. SearXNG schwankt run-to-run.
- **Cache-Vergiftung durch Pre-Warm.** Ein Pre-Warm-Cache konserviert falsche Suchergebnisse über mehrere Runs. Besser: Cache mit gezielten Planner-Regeln füllen, *kein* Pre-Warm.
- **Strukturelle Grenzen (≈10 Fragen).** Diese Fragen scheitern nicht an fehlendem Reasoning, sondern an Login-Gates, fehlenden YouTube-Captions oder Modell-Bias – kein Tool-Fix möglich.
- **Confidence Gate.** Erzwungene Extra-Runden für komplexe Fragen erhöhen den Token-Verbrauch gefährlich. Reversion nötig (Run 5).
- **Expert-Leak-Erkennung.** Interne Planner-Strings (`Attempt web search.`, `Attempt tool call.`) wurden als finale Antwort durchgereicht ohne Retry-Filter – 3 verlorene Punkte.

- **Temperature ist level-abhängig.** T = 0 hilft bei komplexem L3-Reasoning (deterministisch), schadet aber bei L1- Faktenfragen. Lösung: level-adaptive Temperature (L1: 0,1; L2: 0,05; L3: 0,0).

Unabhängig von den absoluten Zahlen lassen sich aus der Pipeline- Beobachtung *qualitative* Aussagen ableiten, die stabil sind:

- **Alle T1-Experten laufen tatsächlich parallel.** Im Orchestrator-Log erscheinen die Expert-Starts binnen weniger Millisekunden auf verschiedenen Nodes, gefolgt von gestaffelten Completions entsprechend der Node-Geschwindigkeit.

- **Self-Correction erkennt numerische Abweichungen.** In unseren Testläufen flaggte der Critic-Node 67 bis 331 „numerische Abweichungen“ in den initialen T1-Antworten – ein starkes Signal, dass Klein-Modelle die Zahlen der VRAM-Rechnung in Teil (c) schlecht treffen. Die Few-Shot-Korrekturen werden als Markdown-Dateien unter `/opt/moe-infra/few_shot_examples/` persistiert und fliessen in spätere Anfragen als Kontext ein – das ist der Graph-basierte Akkumulationsmechanismus in Aktion.

- **GraphRAG-Ingest feuert in jeder Anfrage.** Der Merger-Node emittiert `SYNTHESIS_INSIGHT`-Blöcke, und der Ingest speichert typischerweise 3–8 neue Triples pro Anfrage. Nach einigen Anfragen ist der Neo4j-Graph messbar dichter.

- **T2-Fallback wird regelmässig ausgelöst.** Im 8b-Template löst der Benchmark-Prompt in 4 von 5 Fällen eine Eskalation aus – das ist die Kehrseite der Billigstufe: kleine Modelle sind bei rechen- und fachlich komplexen Fragen unterlegen und werden automatisch verstärkt.

Der Zweck der Baseline-Zahlen ist nicht der Nachweis „wir sind am schnellsten“ – sondern der Nachweis, dass das architektonische Versprechen (parallel, deterministisch, auditierbar, kaskadierend) im realen Betrieb einlösbar ist.

14.15.2 MoE-Eval: Kognitive Benchmark-Suite

Neben der Baseline-Zeitmessung haben wir eine *kognitive Benchmark-Suite* entwickelt (`benchmarks/moe_eval_v1`) die – inspiriert von GAIA und LongMemEval – nicht Token-Durchsatz misst, sondern *Genauigkeit, Routing-Korrektheit und Wissensakkumulation*. Die Suite enthält neun Testfälle in vier Kategorien, ausgeführt mit dem 30b-Balanced-Template.

Bewertungsmethodik. Jeder Test erhält zwei Scores: einen *deterministischen* Score (Keyword-Matching, numerische Toleranz, Exact-Match) und einen *LLM-Judge*-Score (`gpt-oss:20b` über einen direkten Ollama-Call, *nicht* über die MoE-Pipeline). Der kombinierte Score gewichtet 40 % deterministisch + 60 % LLM-Judge. Die Unterscheidung zwischen Pipeline-Routing und direktem LLM-Call für den Judge ist wesentlich: ein früherer Versuch, den Judge über die volle MoE-Pipeline zu routen, produzierte unbrauchbare Scores, weil der Orchestrator die Bewertungsfrage als normalen Expert-Request verarbeitete statt als reines Scoring.

Interpretation. **Stärken** (≥ 7.0): Die MCP-Precision-Tests bestätigen die Kernthese des Projekts – deterministische Werkzeuge eliminieren Halluzinationen. Die Subnet-Berechnung (8.8/10), die Arithmetik (9.4/10) und die Datumsrechnung (10.0/10) werden exakt durch die MCP-Tools `subnet_calc`, `calculate` und `date_diff` gelöst. Der Code-Review-Test (9.0/10) zeigt, dass der Planner die SQL-Injection korrekt an den `code_reviewer`-Experten routet. Der 3-Turn-Memory-Test (7.6/10) belegt einen funktionierenden Graph-basierten Akkumulationsmechanismus: die fiktiven Fakten „Projekt Sovereign Shield verwendet das X7-Protokoll auf Port 9977 mit TLS 1.3“ werden über drei Turns hinweg korrekt synthetisiert. Die Rechtsfrage (4.8/10) erhält vom LLM-Judge 8.0/10 für inhaltliche Korrektheit, scheitert aber am deterministischen Score wegen fehlender exakter Keywords.

Schwächen (< 4.0): Der 5-Turn-Memory-Test (0.9/10) scheitert an der Kontextverdünnung – bei fünf aufeinanderfolgenden Turns mit umfangreichen Zwischenantworten geht der Kontext der frühen Fakten verloren, weil die Gesamthistorie das Token-Budget des Planners überschreitet. Die Medizinfrage (3.8/10) erzielt einen hohen deterministischen Score (9.4) durch korrekte Keywords und Disclaimer, aber der LLM-Judge reagierte nicht (gpt-oss:20b unloaded zwischen Calls). Die Multi-Expert-Synthese (0.9/10) deckt Schwächen im Merger auf: drei parallel generierte Teilantworten (DSGVO + Mathematik + Technik) werden unzureichend konsolidiert.

Ehrliche Bilanz: 6.1/10 Durchschnitt

Ein Durchschnitt von 6.1/10 über neun kognitive Tests ist kein Spitzenwert – aber er ist ein *ehrlicher* Wert. Die Stärken liegen dort, wo die Architektur sie verspricht: deterministische Precision (9.4/10 Durchschnitt über drei MCP-Tests) und Domain-Routing (9.0/10 für Code-Review). Die Schwächen liegen dort, wo die Architektur bekannte Grenzen hat: Long-Context-Memory über viele Turns und die Fusion mehrerer Expertenantworten zu einer kohärenten Synthese. Die Suite ist öffentlich verfügbar unter `benchmarks/` und wir laden die Community ein, sie auszuführen, zu kritisieren und zu verbessern.

14.15.3 Vorher/Nachher: der Akkumulations-Effekt zwischen Runs

Ein zentrales Versprechen der Architektur ist der *Persistentes Graph-State-Tracking-Effekt*: jede Anfrage reichert den Wissensgraphen an, jede Self-Correction schreibt Few-Shot-Examples, jede Judge-Bewertung verfeinert die Performance-Scores. Wenn das stimmt, dann müsste ein *zweiter* Benchmark-Run auf demselben Cluster – ohne Änderungen an Code, Templates oder Hardware – *bessere* Ergebnisse liefern als der erste.

Wir haben diesen Test durchgeführt: Run 2 lief unmittelbar nach Abschluss von Run 1 auf derselben Infrastruktur. Die Hypothesen und ihre Verifikation:

- **GraphRAG-Effekt**: Run 1 hat ∼20 neue Triples im Neo4j-Graphen hinterlassen. Bei Memory-Tests sollten diese als zusätzlicher Kontext einfließen und die Recall-Rate verbessern.

- **Few-Shot-Korrekturen**: Der Critic-Node hat in Run 1 Few-Shot-Examples unter `/opt/moe-infra/few_shot_examples/` persistiert. Bei numerischen Aufgaben sollte die Antwortqualität in Run 2 steigen, weil der Orchestrator die Korrekturen als Prompt-Kontext mitgibt.

- **Modell-Warmth**: Modelle, die in Run 1 geladen wurden und deren Ollama-TTL noch nicht abgelaufen ist, starten in Run 2 ohne Kaltstart-Latenz. Die Wall-Clock sollte sinken.

- **Deterministic-Stabilität**: Die deterministischen Scores (Keyword-Matching, numerische Toleranz) sollten stabil bleiben, da sie nur vom Output-Inhalt abhängen, nicht von Latenzen oder Cache-Zuständen.

Analyse der Abweichungen.

- **8b: +10 % langsamer.** Der GraphRAG-Kontext wuchs von ∼900 auf 1452 Zeichen zwischen den Runs. Mehr Kontext bedeutet längere Prompts an die T1-Experten und einen umfangreicheren Merger-Input. Die Qualität blieb stabil (8.0/10).

- **30b: −16 % schneller, 9.0/10.** Die Few-Shot-Korrekturen aus dem Critic-Node von Run 1 wurden als Markdown-Dateien unter `few_shot_examples/` persistiert und in Run 2 als Prompt-Kontext an die Experten gegeben. Weniger numerische Abweichungen → weniger Critic-Iterationen → kürzere Pipeline-Durchlaufzeit.

- **70b: −55 % schneller, 9.0/10.** Der drastischste Akkumulations-Effekt. Drei Faktoren: (1) Modell-Warmth – `llama3.3-70b-ctx4k` war aus Run 1 noch im VRAM geladen, kein Cold-Start (∼90 s gespart); (2) der Graph-Kontext beschleunigte den Planner; (3) die Few-Shot-Korrekturen reduzierten T2-Eskalationen.

Graph-Akkumulation bestätigt

Der Vorher/Nachher-Vergleich belegt empirisch, dass der Graph-basierter Akkumulationsmechanismus kein theoretisches Konstrukt ist, sondern messbare Auswirkungen auf Performance *und* Qualität hat. Das System wird mit jeder Anfrage ein Stück schneller und besser – und die Verbesserung ist inspizierbar (Neo4j-Graph, Few-Shot-Dateien, Prometheus-Metriken).

14.15.4 Dense-Graph-Run: Messung nach Wissensakkumulation

Kontext. Der MoE-Eval-Lauf vom 12. April 2026 fand bei einem noch vergleichsweise dünn befüllten Neo4j-Graphen statt. Nach intensivem Produktionsbetrieb und mehreren Ontologie-Kuratierungs-Kampagnen wurde am 15. April 2026 ein zweiter Messlauf durchgeführt – dieses Mal mit einem signifikant dichteren Wissensgraphen und vier neuen, per-Node-gepinnten Expert-Templates.

Graphzustand zum Zeitpunkt des Laufs.

Metrik	Wert
Entity-Nodes	4.962
Synthesis-Nodes	391
Gesamt-Nodes	**5.353**
Kanten	5.909
Ø Kanten/Entity	≈1.19

Neue Templates und Cluster-Parallelisierung. Statt eines einzigen Referenz-Templates wurden *fünf Templates gleichzeitig* auf dem gesamten Cluster gestartet, so dass alle GPU-Knoten parallel beschäftigt waren. Vier der fünf Templates wurden neu angelegt; jedes pinnt seine Experten auf eine bestimmte Hardware-Gruppe:

Template	Planner / Judge	Experten
`moe-reference-30b-balanced`	phi4:14b / gpt-oss:20b	mix N04-RTX
`moe-benchmark-n04-rtx`	phi4:14b / qwen3-coder:30b	alle N04-RTX
`moe-benchmark-n07-n09`	phi4:14b@N07 / gpt-oss:20b@N09	N07-GT + N09-M60
`moe-benchmark-n06-m10`	phi4:14b@N06-01 / phi4:14b@N06-02	N06-M10 ×4
`moe-benchmark-n11-m10`	phi4:14b@N11-01 / phi4:14b@N11-02	N11-M10 ×4

Das Einzel-Template nutzt `MOE_PARALLEL_TESTS=3`, das heißt bis zu drei *single-turn*-Tests laufen gleichzeitig pro Runner. Mit fünf Runnern ergibt das maximal 15 gleichzeitige API-Anfragen – alle GPU-Knoten sind durchgehend beschäftigt.

Ergebnisse: per-Template Score-Übersicht.

Vollständige Messreihe: Score-Entwicklung über die Zeit. Über sechs MoE-Eval-Läufe zwischen dem 10. und 15. April 2026 lässt sich eine konsistente Aufwärtsentwicklung ablesen, die direkt mit dem Wachstum des Wissensgraphen korreliert:

Ursachenanalyse: Warum hat sich das Ergebnis verändert? Die Score-Entwicklung lässt sich auf vier unabhängige Einflussgrößen zurückführen:

1. **Graphdichte (Haupttreiber, +2.4 Punkte gesamt).** Der Ø-Gesamt-Score des Referenz-Templates stieg von 5.2 (Lauf 1, ≈500 Nodes) monoton auf 7.6 (5.353 Nodes). Am stärksten profitierte das *Domain-Routing* (+3.4 Punkte), weil der Planner über GraphRAG-Kontext jetzt genug Domänenwissen zum Thema der Anfrage sieht und den richtigen Experten zuverlässig auswählt. *Multi-Expert Synthesis* verbesserte sich von 0.9 auf 5.7 (+4.8) – der Merger kann Widersprüche auflösen, wenn er Vergleichswissen aus dem Graphen hat.

2. **M10-Hardware-Split (struktureller Bruch).** Vor der letzten Infrastruktur-Änderung liefen die Tesla-M10-Knoten als kombinierter Multi-GPU-Block (4×8 GB = 32 GB VRAM pro Chassis). Das ermöglichte 30B-Modelle auf M10-Hardware. Nach dem Split in vier unabhängige Ollama-Instanzen (je 8 GB) ist der Maximalmodell je Instanz auf ≈7B Q4 begrenzt. Die zuvor existierenden Templates `moe-reference-70b-deep` und `cc-expert-balanced` (30B-Judge) sind damit funktionslos geworden – ein direkter Vorher/Nachher-Vergleich auf diesen Tiers ist nicht mehr möglich. Die vier neuen `moe-benchmark-n06/n11-m10`-Templates dokumentieren dieses Limit: unter dem initialen parallelen Benchmark-Load konnten die alten 30B/70B-Templates keinen einzigen Test abschließen. Die neuen `moe-benchmark-n06/n11-m10`-Templates mit hermes3:8b-Modellen schließen alle 9/9 Tests ab (Ø 3.3–3.6), bestätigen die Praxistauglichkeit von Legacy-Hardware bei reduzierter Modellgröße.

3. **Evaluierungs-Methodenänderung (Scoring-Bereinigung).** Der LLM-Judge-Pfad änderte sich zwischen den Läufen: ältere Runs bewerteten ohne expliziten deterministischen Score (det = 0, Formel: $0.4 \times 0 + 0.6 \times \text{LLM}$); ab dem 15.-April-Lauf wurde

der deterministische Score aus Keyword-Matching und numerischer Toleranz korrekt berechnet. Das erklärt den Sprung bei *Routing-Legal* (4.8 → 8.2): der alte Score war methodisch gedeckelt, nicht wegen schlechterer Antwortqualität.

4. **Nebenläufigkeits-Effekt (Qualitätsverlust unter Last).** Das `n04-rtx`-Template erzielte 6.0 statt der 7.6 des `ref-30b`-Templates, obwohl es auf identischer Hardware läuft. Der Unterschied: ref-30b lief *nach* Abschluss des Parallel-Runs solo; n04-rtx lief *gleichzeitig* mit vier weiteren Templates (15 concurrent requests). Unter Volllast steigt die TTFT deutlich, Timeouts häufen sich (compounding-memory-5turn, multi-expert-synthesis blieben 0), und der 30B-Judge (`qwen3-coder:30b`) hatte selbst lange Wartezeiten. Isolierter Lauf des n04-rtx-Templates würde einen höheren Score erwarten lassen.

Vorher/Nachher: Graph-State-Tracking Memory im Detail.

Test	12. April	
compounding-3turn (ref-30b)	8.2	
compounding-5turn (ref-30b)	0.6	0.0
Ø Precision	8.3	
Ø Gesamt	6.8	

Lesson Learned: Vier unabhängige Faktoren, ein Score

Die Score-Verbesserung von 6.8 → 7.6 ist *nicht* auf eine einzelne Ursache zurückzuführen. Graphdichte, Infrastruktur-Split, Evaluierungsmethode und Nebenläufigkeitseffekte überlagern sich. Für saubere Kausal-Attribution müssen zukünftige Benchmark-Kampagnen diese vier Variablen isolieren: einen Lauf mit altem Graphzustand auf neuer Infrastruktur, einen mit neuem Graph auf alter Infrastruktur, und einen Solo-Lauf pro Template. Erst dann ist eine verzerrungsfreie Aussage über den reinen Graphdichte-Effekt möglich.

14.15.5 Einordnung in externe Benchmarks

MoE Sovereign ist kein Einzelmodell und tritt nicht direkt gegen GPT-5 oder Claude auf einem Leaderboard an. Es ist eine *Orchestrierungsschicht*, die Commodity-Modelle durch deterministisches Routing, MCP-Tools und GraphRAG aufwertet. Die Einordnung erfordert daher eine differenzierte Betrachtung.

GAIA-Benchmark. Der GAIA-Benchmark (General AI Assistants) misst Multi-Schritt-Aufgaben mit Tool-Nutzung – konzeptionell verwandt mit unseren MCP-Precision-Tests. Die Leaderboard-Spitze (Stand April 2026):

#	System	Anbieter	Score
1	OpenAI o1	OpenAI	74,1 %
2	Claude 3.7 Sonnet Thinking	Anthropic	≈56,0 %
3	GPT-4o Mini	OpenAI	44,8 %
4	Gemini 2.5 Pro	Google	33,3 %
5	DeepSeek R1 0528	DeepSeek	27,9 %
6	Qwen3 32B Thinking	Alibaba	12,3 %
7	DeepSeek V3.1 Thinking	DeepSeek	11,5 %

Hinweis zur Benchmark-Integrität – GAIA Level 1

Der im Folgenden berichtete Level-1-Wert von 60 % stammt aus demselben Template wie das Gesamtergebnis 46,7 % (Specification-Gaming- Vorfall, Abschnitt 14.6): sämtliche Experten-Slots waren auf ein kommerzielles Frontier-Modell (`gpt-4.1@AIHUB`) geroutet. Das Teilergebnis kann weder für den souveränen Stack noch für eine einzelne Architekturkomponente als Wirksamkeitsnachweis gewertet werden. Es wird hier aus Dokumentationsgründen aufgeführt. Ein valider souveräner GAIA-Lauf steht aus.

Unsere Backbone-Modelle (DeepSeek-R1:32b, Qwen3:32b, Llama3.3:70b) liegen als Einzelmodelle im Bereich 11–28 %. Im kontaminierten Benchmark-Lauf (nicht-souveränes Template) wurden **60 % auf GAIA Level 1** (6/10 korrekt) erzielt – die qualitativen Beobachtungen zu den Lösungsstrategien sind aber unabhängig vom Souveränitätsproblem dokumentierwert:

- MCP-Precision: Berechnung (Kipchoge-Marathon) und Faktenextraktion (Mercedes Sosa Diskographie) wurden durch MCP-Tools exakt gelöst.
- Multi-Expert-Routing: die Game-Show-Wahrscheinlichkeit und die Doctor-Who-Faktenfrage wurden korrekt an `reasoning` bzw. `research` delegiert.
- Vision-Analyse: die Spreadsheet-Aufgabe (Grundstücksfarben, Graph-Konnektivität) wurde korrekt beantwortet.

Fehlschläge: Das System scheiterte an Aufgaben, die externe Dokumente (PDF, arXiv) herunterladen und durchsuchen müssen (Pie Menus Paper, Fish Bag Volume) – die MCP-Tools bieten noch keine PDF-Analyse. Der reversed-text Test scheiterte an der Kontextverdünnung durch den Merger.

LongMemEval. Der LongMemEval-Benchmark misst Langzeit-Erinnerung über viele Chat-Turns. Über mehrere Messreihen mit zunehmendem Graphzustand erzielt MoE Sovereign einen stabilen Durchschnitt von **81,5 %** und im besten Lauf **91,7 %** – ein empirischer Beleg für den kumulativen Wissenseffekt des GraphRAG-Akkumulationsmechanismus:

Kategorie	Stabil	Best	Δ
Information Extraction	100,0 %	100,0 %	±0
Temporal Reasoning	100,0 %	100,0 %	±0
Abstention	100,0 %	100,0 %	±0
Multi-Session-Reasoning	66,7 %	100,0 %	+33,3
Knowledge Update	66,7 %	100,0 %	+50,0
Durchschnitt	**81,5 %**	**91,7 %**	**+10,2**

Die Kategorien Information Extraction, Temporal Reasoning und Abstention werden stabil mit 100 % bewertet – sie profitieren direkt vom Neo4j-Wissensgraphen und dem nächtlichen Ontology-Healer, der veraltete Relationen korrekt überschreibt. Multi-Session-Reasoning und Knowledge Update verbessern sich mit wachsender Graphdichte von 66,7 % auf 100 %, was den Akkumulationseffekt des RL-Flywheels (Abschnitt 2.5) empirisch belegt. Die Top-Systeme im LongMemEval-Leaderboard – EverMemOS (83 %) und TiMem (76,9 %) – nutzen spezialisierte Memory-Architekturen mit dedizierter Session-Persistenz. Der beste interne Lauf von MoE Sovereign erreichte 91,7 % auf einer LongMemEval-orientierten Testkonfiguration; ob exakt dieselben Datensatz-Splits, Antwort- Normalisierer und Bewertungsregeln wie im offiziellen Leaderboard verwendet wurden, ist nicht vollständig dokumentiert. Ein direkter

Leaderboard-Vergleich ist daraus nicht ohne weiteres ableitbar. Qualitativ belegen die Zahlen jedoch den kumulativen Wissenseffekt des GraphRAG-Akkumulationsmechanismus, obwohl Memory nicht der primäre Optimierungszweck der Architektur ist.

MoE-Eval im Frontier-Vergleich. Der MoE-Eval misst nicht dieselben Fähigkeiten wie GAIA oder LongMemEval – er ist auf die spezifischen Stärken eines Compound-AI-Systems ausgelegt (MCP-Tool-Delegation, deterministisches Routing, GraphRAG-Synthese). Ein direkter Score-Vergleich mit Frontier-Modellen ist daher nicht sinnvoll. Was sich aber ableiten lässt:

System	Typ	MoE-Eval (intern)	GAIA Lv. 1
OpenAI o1	Frontier (Cloud)	n/a	74,1 %
Claude 3.7 Sonnet Th.	Frontier (Cloud)	n/a	≈56 %
GPT-4o Mini	Frontier (Cloud)	n/a	44,8 %
Gemini 2.5 Pro	Frontier (Cloud)	n/a	33,3 %
DeepSeek R1:32b	Open (lokal, 32B)	n/a	27,9 %
Qwen3:32b	Open (lokal, 32B)	n/a	12,3 %
MoE Sovereign ref-30b (Apr. 10, Graph ≈500)	Orchestrator (lokal)	5.2/10	–
MoE Sovereign ref-30b (Apr. 12, Graph ≈2k)	Orchestrator (lokal)	6.8/10	–
MoE Sovereign ref-30b (Apr. 15, Graph 5.353)	Orchestrator (lokal)	**7.6/10**	ungültiger Lauf[†]

[†] Wert aus dem in Abschnitt 14.6 dokumentierten kontaminierten Lauf (nicht-souveräne Modellsubstitution); kein valider Leistungsnachweis für den souveränen Stack.

Die drei Zeilen für MoE Sovereign zeigen: der absolute Score auf dem internen MoE-Eval wächst mit dem Graphen, *ohne dass sich das Basismodell verändert*. Die eingesetzten Backbone-Modelle (phi4:14b, qwen3-coder:30b) erreichen als Einzelmodelle auf GAIA <15 % – der Orchestrator zeigt auf dem internen MoE-Eval eine deutliche Steigerung über die Laufzeit. Für GAIA liegt derzeit kein valider souveräner Vergleichswert vor: Der zuvor berichtete Level-1-Wert von 60 % stammt aus dem kontaminierten Lauf und wird nicht als Leistungsnachweis verwendet.

Kein Leaderboard-Vergleich

Wir betonen ausdrücklich: MoE Sovereign ist *kein* System, das auf einem externen Leaderboard antreten kann oder soll. Die obigen Zahlen sind Orientierungspunkte, keine direkten Vergleiche. Unser Beitrag liegt nicht in einem höheren Score auf einer einzelnen Benchmark, sondern in der *architektonischen Fähigkeit*, Commodity-Modelle durch eine inspizierbare, deterministische Orchestrierungsschicht aufzuwerten – bei vollständiger Datensouveränität und ohne externe Abhängigkeiten.

14.15.6 Enterprise-Architektur-Features

Inspiriert durch eine Analyse proprietärer Systeme (Palantir AIP, Databricks Mosaic AI, Glean) wurden vier Enterprise-Architektur- Erweiterungen implementiert und in einem Validierungs-Benchmark verifiziert (6.0/10, keine Regression gegenüber der Baseline).

Confidence-Decay & Self-Healing Knowledge Graph. Jede Relation im Neo4j-Graphen erhält einen *Trust-Score*:

$$\text{trust} = \text{confidence} \times w_{\text{source}} \times \max\left(0.3,\ 1 - \tfrac{\text{Tage}}{365}\right) \times v_{\text{bonus}}$$

wobei $w_{\text{source}} \in \{1.0, 0.9, 0.6\}$ (Ontologie, Healer, extrahiert) und $v_{\text{bonus}} = 1.5$ für verifizierte Relationen. Relationen mit Trust < 0.2, die unbestätigt (`verified = false`) und einmalig (version $= 1$) sind, werden durch folgende Cypher-Query entfernt: `MATCH (a)-[r]->(b) WHERE r.trust_score < 0.2 AND r.verified = false AND r.version = 1 DELETE r`. Die Löschung erfolgt im Phase-3-Linting automatisch und wird im Kafka-Topic `moe.audit` protokolliert. Der nächtliche Ontology Gap Healer führt vor der Lückenforschung ein identisches Cleanup durch (Phase 0). Dies löst das *Wissenskontaminationsproblem*: falsche Assoziationen aus schlechten Queries werden automatisch entfernt, sobald ihr Trust-Score unter die Schwelle fällt.

Ontologie-gestütztes RBAC (Multi-Tenant). Inspiriert durch Palantir AIPs Ontologie-RBAC wird jeder Neo4j-Knoten optional mit einer `tenant_id` versehen. Der neue Permission-Typ `graph_tenant` steuert, welche Graph-Partitionen ein Nutzer sehen kann. Die Cypher-Queries in `query_context()` filtern mit `WHERE e.tenant_id IN $tenant_ids OR e.tenant_id IS NULL`, sodass mandantenübergreifende Isolation auf Datenbankebene erzwungen wird – ohne dass das LLM eingreifen muss.

Inline-Provenance-Tags. Der Merger erhält eine `PROVENANCE_INSTRUCTION`, die Fakten aus dem Wissensgraphen mit `[REF:entity_name]` markiert. Diese Tags werden post-merger extrahiert und als `metadata.sources`-Feld in der API-Response zurückgegeben (backward-kompatibel zum OpenAI-Format). Clients können die Herkunft jeder Aussage bis zum Neo4j-Knoten zurückverfolgen.

Blast-Radius-Estimation & Quarantäne. Bevor ein neues Triple in den Graphen geschrieben wird, prüft `_estimate_blast_radius()` wie viele bestehende Entitäten innerhalb von 2 Hops erreichbar sind. Überschreitet die Reichweite den Schwellwert (Standard: 20), wird das Triple nicht geschrieben, sondern in einer Redis-Quarantäne-Queue (TTL 7 Tage) gespeichert. Eine neue Admin-UI-Seite „Quarantine“ erlaubt die manuelle Freigabe oder Ablehnung. Dies verhindert, dass ein einzelnes fehlerhaftes Triple ganze Wissensbereiche kontaminiert.

Enterprise-Validierung: 6.0/10 — keine Regression

Die vier Features wurden in einem separaten Benchmark-Lauf validiert. Alle neun Tests bestanden, der kombinierte Score blieb bei 6.0/10 – identisch mit der Baseline vor den Erweiterungen. Die Neo4j-Queries für Tenant-Filtering und Blast-Radius-Prüfung fügen < 50 ms Latenz pro Request hinzu – vernachlässigbar gegenüber den 100–600 s Pipeline-Laufzeiten.

14.15.7 Adversarial MCP Tool Security Testing

Um die Robustheit des AST-Whitelisting-Evaluators empirisch zu validieren, wurde eine adversariale Testsuite mit 9 Angriffsversuchen und 1 legitimen Berechnung durchgeführt. Die Angriffe umfassen typische Prompt-Injection-Vektoren, die darauf abzielen, beliebigen Code über das `calculate`-Tool auszuführen:

Angriffsvektor	**Blockiert**
`__import__`-Injection	✓
Attribut-Zugriff auf Module	✓
Lambda-Ausführung	✓
`eval`-Injection	✓
Dunder-Zugriff (`__globals__`)	✓
`compile`-Exploit	✓
`globals()`-Zugriff	✓
Verschachtelte `eval`-Kette	✓
Format-String-Injection	✓
Legitime Berechnung (`2+2`)	× (erlaubt)

Ergebnis: 9/9 Angriffe blockiert, 1/1 legitime Berechnung zugelassen – **100 % AST-Firewall-Effektivität**. Der AST-Evaluator parst jeden Ausdruck vor der Ausführung und erlaubt ausschließlich Knoten aus einer expliziten Whitelist (Literale, arithmetische Operatoren, Funktionsaufrufe aus `SAFE_FUNCTIONS`). Dies bestätigt empirisch die Behauptung, dass deterministische Tool-Ausführung als Halluzinations-Firewall fungiert.

14.15.8 LLM-Rollentauglichkeit: Planner und Judge

Insgesamt wurden 69 LLMs auf fünf Inferenz-Knoten systematisch auf ihre Eignung als *Planner* (JSON-Aufgabenzerlegung) und *Judge* (JSON-Qualitätsbewertung) getestet. Von den 69 Modellen bestehen 42 (61 %) beide Rollen, 6 (9 %) nur den Planner, 7 (10 %) nur den Judge und 14 (20 %) scheitern an beiden. 72 % aller Modelle erzeugen valides Planner-JSON, 78 % valides Judge-JSON.

Zentrale Erkenntnisse. `phi4:14b` erweist sich als bestes Allround-Modell (Planner + Judge in 37,8 s, konsistentes JSON). Modelle mit Thinking-/Reasoning-Modus (`qwen3.5:35b`, `deepseek-r1:32b`) scheitern als Planner, da `<think>`-Tags das JSON-Parsing brechen. `gpt-oss:20b` besteht isolierte Tests, versagt aber in der Pipeline durch Ollama-TTL-Entladung zwischen Expert-Aufrufen. Code-spezialisierte Modelle (`devstral`, `qwen3-coder`) funktionieren als Planner *und* Judge.

14.15.9 Claude Code Profile Benchmark

Drei Referenz-Profile wurden für die Integration mit Claude Code CLI und VS Code erstellt und gegen fünf typische Software-Engineering-Aufgaben getestet (SQL-Injection-Fix, Health-Endpoint, Refactoring, Pytest-Tests, Security-Review):

Ergebnis. Das orchestrierte Profil ist im Schnitt 35 % schneller als das native Profil und verbraucht 17 % weniger Token – ein kontraintuitives Ergebnis, das durch den Akkumulations-Effekt erklärt wird: der GraphRAG-Kontext und die ChromaDB-Cache-Hits aus den vorherigen Benchmark-Runs beschleunigen die Pipeline erheblich. Der Planner kann auf akkumuliertes Wissen zugreifen statt alles von Grund auf zu generieren.

14.15.10 Anmerkung zur Hardware und Reproduzierbarkeit

Die in dieser Arbeit gemessenen Laufzeiten sind *systembedingt* und stellen keine Referenz für andere Installationen dar. Die verwendete Hardware – fünf RTX 3060 mit je 12 GB VRAM, mehrere Tesla-M10- und -M60-Karten mit 8 GB VRAM, zwischen 14 und 128 GB System-RAM pro Knoten – ist *keine* optimale Konfiguration für LLM-Inferenz. Es ist die Konfiguration, die mit einem privaten Budget erreichbar war.

Das gesamte Cluster wurde vom Autor persönlich aufgebaut: ein 48-HE-19"-Serverrack, bestückt mit Servern, die einzeln beschafft, zusammengebaut und für diesen Zweck optimiert wurden. Keine gesponserte Hardware, kein Cloud-Kredit, kein institutionelles Budget. Jeder Benchmark-Wert in diesem Paper wurde auf Servern gemessen, die der Autor mit eigenem Geld gekauft und mit eigenen Händen verkabelt hat.

Das ist kein Nachteil – es ist der Punkt. *Digitale Souveränität* bedeutet, dass man mit dem arbeitet, was man hat, nicht mit dem, was man sich wünscht. Wenn ein System auf fünf gebrauchten RTX 3060 funktioniert und messbare Ergebnisse liefert, dann funktioniert es erst recht auf besserer Hardware. Die Latenzen in diesem Paper sind Obergrenzen, nicht Untergrenzen.

Wer dieses System nachbauen möchte, braucht weder ein Rechenzentrum noch ein Forschungsbudget. Ein einzelner Server mit einer aktuellen GPU (RTX 4090 mit 24 GB VRAM oder besser) und 32 GB RAM reicht für das `solo`-Profil aus. Was zählt, ist nicht die Hardware – sondern die Bereitschaft, die Kontrolle über die eigene Infrastruktur nicht abzugeben. Dieses Whitepaper ist der Beweis, dass das mit bescheidenen Mitteln möglich ist – als Machbarkeitsstudie und als Einladung zum Nachmachen.

Legacy-Hardware als Stresstest, nicht als Kompromiss. Die Tesla-M10-GPUs (8 GB VRAM, DDR3, PCIe 2.0) waren kein Budget-Kompromiss, sondern der härteste verfügbare Integrationstest für die Orchestrierungs- schicht. Ein System, das unter diesen Bedingungen – hohe Latenzen, knappes VRAM, langsame Bus-Bandbreite – deterministisch korrekte Ergebnisse produziert, ist auf modernen Clustern (H100 SXM5, NVLink-Bandbreite >900 GB/s) nicht nur lauffähig, sondern strukturell überlegen: dieselbe Cacheschicht, die auf dem M10 Sekunden einspart, spart auf dem H100 Millisekunden – bei gleichzeitig 50–100× höherem Durchsatz und drastisch steigender Anzahl unterstützter Concurrent Users pro Knoten.

14.16 Fähigkeitsanalyse: MoE Sovereign vs. Cloud-APIs

Eine zentrale Frage für potenzielle Anwender: *Kann ein selbstgehostetes MoE-System eine kommerzielle API (z.B. Anthropic Claude) für produktive Coding-Aufgaben ersetzen?*

Single-Shot-Qualität. Bei einem einzelnen API-Aufruf übertreffen Frontier-Modelle (Claude Opus, GPT-5) lokale 31B-Modelle deutlich bei komplexen Generierungsaufgaben. Unser erster Versuch, ein HTML5-Canvas-Spiel über die MoE-Pipeline zu generieren, produzierte Code mit doppelten `const`-Deklarationen und halluziniertem Text nach dem `</script>`-Tag.

Der Learning Loop verändert die Gleichung. Das Template `cc-expert-70b-deep` kombiniert vier Spezialisten:

- **code_reviewer** (llama3.3:70b): Ursachendiagnose
- **web_researcher** (phi4:14b): SearXNG-Recherche
- **agentic_coder** (Qwen3-Coder-Next, 79.7B): Fixes
- **vision_verifier** (phi4:14b-fp16 / qwen3-vl:8b): Screenshot-Analyse

Dimension	Cloud-API	MoE Sovereign
Erstversuch-Qualität	Hoch	Mittel
Lernfähigkeit	Keine (stateless)	Ja (GraphRAG)
Web-Wissen in Echtzeit	Nein	Ja (SearXNG)
Automatische Verifikation	Nein	Ja (Playwright)
Multi-Perspektive	1 LLM	3–4 Experten + Judge
Vision-Verifikation	Ja (nativ)	Ja (Vision-Experte)
Kosten pro Versuch	$0,50–2,00	$0 (lokal)

Cloud-API = **Sprinter** (schnell, präzise, teuer, stateless). MoE Sovereign = **Marathonläufer** (langsamer pro Schritt, wird mit jeder Iteration besser, bei null Grenzkosten).

14.17 Deployment-Reife

Transparenz über den Teststatus ist für Reproduzierbarkeit essenziell.

Wir halten diese ehrliche Offenlegung für besser als universelle Deployment-Bereitschaft ohne Evidenz zu behaupten.

Warum Trial & Error statt Lehrbuch

Das Wissen, das in diesem Projekt steckt – wie man einen 70B-Judge auf Consumer-GPUs zum Laufen bringt, wie man Ghost-Keys in Valkey diagnostiziert, wie man heterogene GPU-Cluster mit unterschiedlichen VRAM-Größen balanciert – ist nirgendwo in einem Lehrbuch dokumentiert. Es entsteht ausschließlich in der Praxis, durch Ausprobieren, Scheitern, Debuggen und Iterieren. Die KI-Entwicklung der nächsten Jahre wird nicht nur von denjenigen vorangetrieben, die die teuerste Hardware haben, sondern auch von denen, die mit dem Vorhandenen das Bestmögliche herausholen und ihre Erfahrungen öffentlich teilen.

14.18 M10-Gremium-Experiment: Kompensiert Graphdichte kleine LLMs?

Am 15. April 2026 wurde das Template `moe-m10-8b-gremium` getestet: acht Tesla-M10-GPUs (je 8 GB VRAM) als Experten-Knoten, alle mit Modellen zwischen 7 und 9 Milliarden Parametern. Planner und Judge: phi4:14b auf N07-GT ($2 \times$ GT 1060 = 12 GB). Forschungsfrage: *Kann ein dichter Wissensgraph (5.353 Nodes, 5.909 Kanten) die Qualitätslücke zu größeren Modellen schließen?*

Prompt und Scoring

Ein einziger Multi-Domain-Prompt (1.893 Zeichen) verlangt Rechtskompetenz (DSGVO Art. 9, EU AI Act), medizinische Statistik (Sensitivität, Stichprobengröße), Infrastrukturmathematik (10 TB/Tag, 5-Jahres-Archiv) und ML-Grundlagen (Bias-Variance-Trade-off, Regularisierung). Sieben deterministische Checks (Gesamtgewicht 10,5) erlauben eine reproduzierbare Bewertung ohne LLM-Judge.

Ergebnisse

Ursachenanalyse: Context-Overflow auf N07-GT

Mit 5.353 Graph-Nodes injiziert der GraphRAG-Abruf rund 5.000 Token Triplekontext in den Planner-Prompt. phi4:14b auf N07-GT hat ein Kontextfenster von 8.192 Token. Das sättigte das Fenster vollständig: Der Planner beantwortete die Frage in Fließtext statt ein JSON-Routing-Schema zurückzugeben.

Nach 28 Minuten reiner Planner-Zeit wurde lediglich der `legal_advisor`-Experte (sauerkrautlm-7b-hero) aufgerufen. Das Modell antwortete im Kritik-Modus statt direkt auf die Fragen.

Lessons Learned

1. **Graphdichte schadet kleinen Kontext-Fenstern.** Ab ≈3.000 Nodes überschreitet die GraphRAG-Injektion das effektive Instruktionsfolgefenster von phi4:14b (8k). Der Planner braucht ≥16.384 Token, oder der GraphRAG-Abruf muss auf top-$k = 10$ Tripel begrenzt werden.

2. **M10-Experten sind domänenkompetent.** sauerkrautlm-7b-hero lieferte eine kohärente rechtliche Analyse seines Teilbereichs. Das Schwachpunkt war das Routing, nicht das Modell.

3. **Graphdichte kompensiert Context-Overflow nicht.** Graphwissen verbessert Qualität nur, wenn der Planner korrekt routen kann. Ein überlasteter Planner negiert sämtliche Experten- und Graph-Vorteile.

4. **Mitigation:** Planner an Knoten mit großem Kontext- fenster binden (z. B. N04-RTX mit qwen2.5-coder:7b bei 32k Tokens), *oder* GraphRAG-Tiefe bei Legacy-Hardware-Plannern hart begrenzen.

14.19 moe-m10-gremium-deep: Orchestriertes 8-Experten-Ensemble

Der Nachfolger des fehlgeschlagenen `moe-m10-8b-gremium`-Templates (Abschnitt 14.18) adressiert den Context-Overflow durch eine klare Hardware-Trennung: Planner und Judge laufen auf **phi4:14b@N04-RTX** mit 16.384 Token Kontextfenster und Flash Attention; die acht Domänenexperten bleiben auf den Tesla M10 GPUs (je 8 GB VRAM).

Template-Konfiguration

Alle acht Experten-Modelle sind auf Q4_K_M quantisiert ($\leq$5.7 GB VRAM), kein CPU-Offloading. Gesamter Cluster: 88 GB VRAM verteilt auf neun Knoten.

Overnight Stability Benchmark

Am 19.–20. April 2026 wurde das Template in einem 11-stündigen Stabilitätslauf (*overnight benchmark*) mit der MoE-Eval-v2-Suite getestet. Die Suite umfasst 12 zusammengesetzte Szenarien aus sechs Kategorien; jede Epoche läuft alle 12 vollständig durch.

36 Szenarioausführungen, null Fehler. E2 war 25 % schneller als E1 (Ollama-Modelle blieben nach der ersten Epoche im VRAM warm).

Kategorieergebnisse (Durchschnitt über 3 Epochen)

Bekannte Einschränkung: memory-8turn (6.3 $\rightarrow$ 1.8). Das 8-Turn-Memory-Szenario generiert dichte Experten-Antworten, die das 16.384-Token-Kontextfenster des Judges ab Turn 8 füllen. Dies ist ein *Konfigurationslimit*, kein Architekturproblem: das Erhöhen von `OLLAMA_CONTEXT_LENGTH` auf 32k auf N04-RTX würde es beheben. Der 10-Turn-Test verbesserte sich in E3 tatsächlich (4.2 $\rightarrow$ 4.8), weil seine Einzel-Turn-Antworten kürzer sind.

Orchestration-Premium

Der **Orchestration-Premium** beträgt +2.5 bis +2.8 Punkte gegenüber einem einzelnen 7B-Modell auf identischer Hardware. Das Ensemble schließt 60 % des Abstands zwischen einem einzelnen 7B- und einem 30B-System.

Vergleich mit öffentlichen Cloud-Modellen

Tabelle 27 stellt den gemessenen MoE-Eval-Score gegenüber publizierten Benchmarks für Modelle der 7–14B-Klasse.

Einschränkung: MMLU und MT-Bench messen isolierte Einzelmodellfähigkeiten. MoE-Eval misst Compound-AI-Orchestrierungsqualität (Routing, Spezialisierung, GraphRAG-Synthese, Multi-Turn-Memory). Kreuzvergleiche sind richtungsweisend, nicht exakt.

Kernaussage. Ein selbst gehostetes Ensemble aus acht 7–9B-Domänenspezialisten auf Legacy-Tesla-M10-Hardware erreicht 6,11/10 auf dem internen MoE-Eval – bei vollständiger Datensouveränität und ohne Token-basierte Nutzungsgebühren. Ein direkter Qualitätsvergleich zu cloud-basierten Diensten ist aufgrund unterschiedlicher Benchmarks nicht möglich.

14.20 Gesamtbewertung

14.20.1 Externe Einschätzung

Im Rahmen eines KI-gestützten Peer-Review-Prozesses wurde MoE Sovereign qualitativ entlang der Dimensionen Architekturreife, Innovationsgrad, Skalierbarkeit, Reproduzierbarkeit und gesellschaftliche Relevanz bewertet. Hervorgehoben wurde die konsequente Verbindung bekannter Einzelbausteine (MoE-Routing, GraphRAG, RL-Flywheel) zu einer souveränen, lernenden Compound-AI-Infrastruktur sowie die Transparenz der dokumentierten Fehlschläge. Benchmark-bezogene Aussagen aus dieser Evaluation werden in dieser Arbeit nicht als quantitative Belege verwendet, da der zugrundeliegende GAIA-Messlauf nachträglich als ungültig eingestuft wurde (Abschnitt 14.6) und eine KI-generierte Selbstevaluation kein Ersatz für kontrollierte externe Benchmarks darstellt.

14.20.2 Kritische Eigeneinschätzung der Autoren

Selbstbewertungen sind methodisch begrenzt: Blinde Flecken, Optimierungsbias und die unvermeidliche Investition des Autors in das eigene Projekt verzerren das Urteil. Wir halten dennoch eine explizite Gegenüberstellung für wissenschaftlich redlicher als einen bloßen Verweis auf externe Zahlen.

Stärken.

- **Architektur** – Deterministisches Routing, vierschichtige Cache-Hierarchie und das RL-Flywheel sind konzeptuell kohärent und in jedem Schritt nachvollziehbar.
- **Memory-System** – Die Kombination aus Neo4j (GraphRAG), ChromaDB (Semantik-Cache) und Correction-Memory erzeugt nachweisbar akkumulierende Qualität; die LongMemEval-Zahlen belegen das.
- **Deterministische Pipeline** – MCP-Präzisionswerkzeuge mit AST-Whitelist eliminieren Halluzinationen bei rechenintensiven Aufgaben vollständig.
- **Self-Hosted-Fähigkeit** – Ein einziges OCI-Image, drei Profile, vier Wrapper: das Deployment-Modell ist in der Open-Source- Landschaft ungewöhnlich durchdacht.
- **Reproduzierbarkeit** – Alle Benchmark-Ergebnisse sind auf der dokumentierten Hardware reproduzierbar; die Infrastruktur ist vollständig als Code verfügbar.

Schwächen.

- **Reasoning-Grenzen kleiner Modelle** – GAIA Level 3 und tiefes Multi-Step-Reasoning bleiben strukturell schwach. Das ist keine Architektur-Schwäche, sondern eine inhärente Eigenschaft von Modellen unter 30B Parametern; sie lässt sich durch bessere Backbone-Modelle adressieren, aber nicht durch Orchestrierung allein.
- **Infrastrukturkomplexität** – 19 Docker-Services, 51 MCP-Tools, heterogene GPU-Knoten: der Betriebsaufwand ist für eine einzelne Person erheblich. Die Dokumentation ist ausführlich, aber die Lernkurve bleibt steil.
- **Einzel-Entwickler-Bias** – Das Projekt wurde von einer Person konzipiert, implementiert und evaluiert. Blinde Flecken in der Benchmark-Auswahl und Designentscheidungen, die einer breiteren Gemeinschaft nicht intuitiv erscheinen, sind wahrscheinlich vorhanden.

Ehrliche Gesamtschau

Die dokumentierten Stärken – Architektur, Memory-Akkumulation, Reproduzierbarkeit – sind belastbar; die LongMemEval-Zahlen und die MCP-Präzisionsevidenz stützen sie. Was noch fehlt, sind kontrollierte externe Benchmark-Ergebnisse für den souveränen Stack. Ein SLM-Ensemble ist kein Ersatz für Frontier-Modelle auf Aufgaben, die tiefes parametrisches Wissen oder Long-Horizon-Reasoning erfordern. MoE Sovereign besetzt einen anderen Punkt im Design-Raum – und den besetzt es gut.

14.21 Komponentenbeitrag-Analyse (Strukturelle Ablation)

Eine vollständig kontrollierte Ablationsstudie – identische Eingaben, identische Hardware, jeweils eine Komponente entfernt – übersteigt den Rahmen eines Einzelpersonenprojekts. Stattdessen präsentieren wir eine *strukturelle Ablation*: jede Komponente wird anhand dedizierter Benchmark-Evidenz bewertet.

Interpretation. Keine einzelne Komponente erklärt die Gesamtleistung allein. Cache-Hierarchie: größter Latenzeffekt (9,3×). AST-Evaluator: größter Genauigkeitseffekt bei deterministischen Aufgaben. Semantic Memory: einziger Enabler für Recall jenseits des nativen Kontextfensters.

Einschränkungen. Strukturelle Ablation ist kein Ersatz für kontrollierte Experimente. Hardware-Varianz und fehlende identische Wiederholungen bedeuten unquantifizierte Unsicherheit in den Δ-Schätzungen.

14.22 GraphRAG vs. Zero-Shot: 10-Anfragen-Direktvergleich

Evaluationsdesign. Am 2026-07-11 wurden 10 repräsentative Anfragen aus dem operativen Prompt-Pool parallel durch beide Routing-Pfade geschickt:

- **Zero-Shot**: Direkte Weiterleitung an das zuständige T1/T2-Modell ohne Graph-Kontext.
- **GraphRAG**: Neo4j-Subgraph-Retrieval (bis zu 5.353 Nodes, ≈5.000 Graph-Kontext-Token) vor der Modellanfrage.

Scoring: Likert-Skala 0–5, manuell bewertet. Zeitlimit: 300 Sekunden pro Anfrage.

Ergebnisse. Von den drei auswertbaren Anfragen zeigten zwei eine Qualitätssteigerung durch GraphRAG; eine war identisch (f005, je 4,0 Punkte):

- **s001** (Attention-Is-All-You-Need-Zusammenfassung): Zero-Shot 4,0 Punkte in 46,79 s – GraphRAG **5,0 Punkte in 6,73 s**. Latenzgewinn: −40 s (85,6 % schneller). GraphRAG lieferte vorverdichteten Graphkontext aus dem Wissengraphen, sodass das Modell keine eigene Recherche durchführen musste.

- **c003** (Consistency-Trade-offs in verteilten Systemen): Zero-Shot 3,0 Punkte in 39,52 s – GraphRAG **4,0 Punkte in 48,19 s**. Scoregewinn +1,0; GraphRAG erfasste nuanciertere CAP-Theorem-Bezüge aus dem Wissensgraphen, mit minimalem Latenzoverhead (+8,6 s).

Einschränkungen. Fünf Anfragen (f001, f002, f004, c001, s002) erreichten das Zeitlimit auf *beiden* Pfaden. Ursache: Der Host-Rechner (`ki-vm-node05`) befand sich unter extremer Ressourcenlast (Swap-Auslastung 100 %, CPU-Lastmittel >12,0) während paralleler Docker-Build- und Trainingsprozesse. Die Timeouts sind keine GraphRAG-spezifischen Ausfälle.

Die Stichprobengröße (3 auswertbare Paare) ist zu klein für statistische Signifikanz. Dennoch liefert das Experiment erste empirische Belege für den Nutzen des Wissensgraphen: In Szenarien mit hoher Kontextdichte (Architektur-Zusammenfassungen, Trade-off-Analysen) zeigt GraphRAG messbar bessere Ergebnisse bei teils deutlich reduzierter Latenz.

GraphRAG-Nutzen zeigt sich zuerst bei kontextdichten Aufgaben

Bei Anfragen, für die der Wissensgraph dichte und direkt relevante Subgraphen enthält (Architektur-Beschreibungen, Theorem-Referenzen), liefert GraphRAG sowohl Latenz- als auch Qualitätsgewinne. Der 85,6 %-Latenzgewinn bei s001 entstand, weil vorverdichteter Graphkontext das LLM-Reasoning wesentlich abkürzte. Für benchmarkfähige Evidenz ist eine Wiederholung unter kontrollierten Hostbedingungen (kein konkurrierender Swap-Druck) und mit $n \geq 50$ Anfragen notwendig.

14.23 Technische Unterstützung ausgewählter ISO/IEC-27001-Kontrollen

Um den Einsatz in regulierten Unternehmensumgebungen zu unterstützen, ist die Sicherheitsarchitektur von MoE Sovereign und *MoE Codex* an den Sicherheitskontrollen der Norm **ISO/IEC 27001:2022** ausgerichtet. Da eine Software-Infrastruktur keine eigenständige Zertifizierung erlangen kann (dafür sind organisatorische und physische Audits erforderlich), stellt die Plattform technische Mechanismen bereit, die Annex-A-Anforderungen direkt adressieren.

Tabelle 30 zeigt die Zuordnung der implementierten Kontrollen, die technische Nachweisführung und die verbleibenden Aufgaben für den Betreiber.

Verantwortung des Betreibers. Die in Tabelle 30 dokumentierte technische Bereitschaft deckt die Anforderungen auf Software-Ebene ab. Für eine vollständige ISO-27001-Zertifizierung muss der Betreiber bzw. Implementierungspartner diese Kontrollen durch organisatorische Verfahren ergänzen: Erstellung von Sicherheitsleitlinien (A.5.1), Durchführung von Awareness-Schulungen (A.6.3), Absicherung der physischen Serverstandorte (A.7) sowie Definition eines Incident-Response-Prozesses (A.8.9).

14.24 Operative Schutzmaßnahmen für nachtrainierte SLM-Komponenten

Die Einführung qLoRA-nachtrainierter SLMs für die Rollen *Planner* und *Judge* (vgl. Abschnitt 15.10) erzeugt zwei Klassen operativer Risiken, die die Architektur explizit abfangen muss.

14.24.1 Format-Drift-Validator für parakonsistente Ausgaben

Ein neuronales Netz, das auf 90.000 parakonsistenten Beispielen trainiert wurde, lernt prinzipiell, strukturierte vierwertige Wahrheitstypen aus dem Belnap-Dunn-Verband $\{T, F, B, N\}$ zu emittieren. Neuronale Netze sind jedoch inhärent probabilistisch: Jede Temperaturperturbation, Prompt-Variation oder Distributionsverschiebung kann dazu führen, dass das Modell unstrukturierten Fließtext statt der erforderlichen XML-Konflikt-Map ausgibt. Tritt dieser Format-Drift unbemerkt auf, erhält der nachgelagerte parakonsistente Merger fehlerhafte Eingaben, und die gesamte Audit-Kette kollabiert.

Als Gegenmaßnahme wird ein **deterministischer Validator-Layer** zwischen jeden SLM-Aufruf und die Merger-Pipeline geschaltet:

1. **Pydantic-Schema-Validierung.** Die SLM-Ausgabe wird gegen ein striktes `TruthTypeModel`-Schema geparst. Akzeptierte Werte pro Assertion-Slot sind ausschließlich `T`, `F`, `B` und `N`.
2. **Regex-Vorfilter.** Vor dem Pydantic-Parsing bestätigt ein leichtgewichtiger regulärer Ausdruck das Vorhandensein der erwarteten XML-Tags und verwirft Fließtext-Antworten in unter 0,01 ms.
3. **Fail-Safe-Fallback.** Bei jedem Validierungsfehler wird der Assertion der konservative Wahrheitswert N (*unbekannt*) zugewiesen, anstatt eine Vermutung weiterzupropagieren. Dies erhält das Fail-Open-Prinzip: Im Zweifel gesteht das System Unwissenheit ein, anstatt Gewissheit zu halluzinieren.

Die resultierende Pipeline lautet:

$$\text{SLM} \rightarrow \text{Regex-Vorfilter} \rightarrow \text{Pydantic TruthTypeValidator} \rightarrow \{T, F, B, N\} \rightarrow \text{Parakonsistenter Merger}$$

wobei bei jeder Validierungsausnahme N injiziert wird. Diese Garantie ermöglicht es der parakonsistenten Kette, formal korrekt zu bleiben, selbst wenn die zugrunde liegende neuronale Komponente degradiert.

14.24.2 Semantische Entitätsverknüpfung für Graph-Stabilität

Die Ratcliff/Obershelp-Ähnlichkeitsmetrik (Python `SequenceMatcher`, 1988) kanonisiert Entitäts-Strings vor jedem `Neo4j MERGE`-Aufruf und unterdrückt typografische Varianten wirkungsvoll. Ihre Grenzen zeigen sich bei semantisch äquivalenten, aber lexikalisch disjunkten Begriffen: *„Bürgerliches Gesetzbuch"* und *„BGB"* weisen nahezu keine Zeichenüberlappung auf, obwohl sie identische Referenten bezeichnen. Bei echtem Enterprise-Einsatz führt unkontrollierte Synonymie zu Graph-Fragmentierung – für dieselbe Real-World-Entität akkumulieren separate Knoten – und verschlechtert den Retrieval-Recall.

Die Mitigationsstrategie erfolgt in zwei Stufen:

Stufe 1 – Abkürzungslexikon (Nulllatenz). Eine domänenspezifische Nachschlagetabelle in PostgreSQL bildet kanonische Kurzformen auf ihre Volltextäquivalente ab (*BGB* → *Bürgerliches Gesetzbuch*, *NIS2* → *NIS2-Richtlinie*, *DSGVO* → *Datenschutz-Grundverordnung* usw.). Jeder Entitäts-String wird vor dem Ähnlichkeitsvergleich über diese Tabelle aufgelöst. Das Lexikon ist durch den Betreiber ohne Code-Änderungen erweiterbar.

Stufe 2 – Lokaler Cross-Encoder (CPU, ≈2 ms). Für Strings, die nach Stufe 1 unaufgelöst bleiben, berechnet ein quantisiertes Modell *cross-encoder/ms-marco-MiniLM-L-6-v2* (INT8 ONNX, ≈6 MB) einen semantischen Ähnlichkeits-Score. Paare, die einen einstellbaren Schwellenwert θ (Standard: 0,85) überschreiten, werden zur kanonischen Form zusammengeführt. Mit ONNX Runtime auf der CPU beträgt die Inferenzlatenz 1–3 ms – vernachlässigbar gegenüber dem Neo4j-Schreibvorgang selbst.

Diese zweistufige Pipeline gewährleistet die strukturelle Integrität des Wissensgraphen bei wachsendem Corpus, ohne eine GPU-Abhängigkeit oder einen Netzwerkaufruf an einen externen Embedding-Service einzuführen.

15 Diskussion und Ausblick

Dieser Abschnitt reflektiert offene Fragen, bekannte Limitationen und die konkrete Forschungsagenda der nächsten Monate. Wir begreifen das Whitepaper nicht als Abschlussbericht eines fertigen Systems, sondern als Einladung zur Mitarbeit an einem Infrastruktur-Projekt, das in klarer Richtung wächst, aber noch viele offene Flanken hat.

15.1 Bekannte Limitationen

Wir benennen die vier Bereiche, in denen das aktuelle System und dieses Whitepaper anerkannte Lücken aufweisen; jede ist eine explizite Engineering- oder Forschungspriorität.

Benchmark-Reproduzierbarkeit. Der einzige vollständig dokumentierte GAIA-Lauf ($14/30 = 46{,}7\,\%$) ist als ungültig eingestuft (Specification-Gaming-Vorfall, Abschnitt 14.6); ein valider souveräner Messlauf steht noch aus. Sobald ein gültiger Lauf vorliegt, gilt: GAIA-Runs umfassen 30 Fragen (10 je Schwierigkeitsstufe), was ein 95 %-Konfidenzintervall von rund ±18 Prozentpunkten ergibt – jedes Einzelergebnis ist daher eine Schätzung, kein statistisch gesicherter Mindestwert. Per-question-Outputs, Prompt-Seeds und Dependency-Hashes für ein reproduzierbares Artefakt sind noch nicht veröffentlicht; der Evaluierungsharness (`benchmarks/gaia_runner.py`) ist öffentlich, das formale Artefakt-Release mit fixierten Seeds ist geplant.

Cache-Sensitivitätsanalyse. Die genannte Einsparung von 40–80 % der LLM-Aufrufe basiert auf Produktionsbeobachtungen bei konversationellen Workloads. Kein systematischer Sweep über TTL-Konfigurationen, Cosinus-Distanz-Schwellwerte oder Workload-Typen wurde veröffentlicht. Betreiber sollten diesen Bereich als workload-abhängigen Schätzwert, nicht als garantierte Einsparung, verstehen und eigene Deployments über die Prometheus-Metrik `cache_hit_total` instrumentieren.

DSGVO: Designaussage vs. Rechtszertifizierung. Die in Abschnitt 13 beschriebenen Privacy-by-Design- Eigenschaften sind *architektonische* Aussagen, die im Quellcode nachprüfbar sind. Ob ein konkretes Deployment rechtlich DSGVO-konform ist, erfordert eine Datenschutz-Folgenabschätzung (DSFA, Art. 35 DSGVO) und die Prüfung durch einen qualifizierten Datenschutzbeauftragten. Dieses Whitepaper stellt keine Rechtsberatung dar.

Template-Routing-Konfliktauflösung. Der aktuelle Planner verwendet einen JSON-strukturierten Task-Graphen mit `depends_on`-Semantik für sequenzielle Ketten. Die Konfliktbehandlung, wenn sich zwei Experten-Outputs widersprechen, ist an das Judge-Modell delegiert; eine formale Spezifikation des Judge-Tiebreaker-Verhaltens unter adversarialen Eingaben wurde noch nicht veröffentlicht. Dies ist ein aktiver Entwicklungsbereich.

15.2 Komplexitäts-Pre-Routing

Die aktuelle Architektur klassifiziert jede Anfrage über den Planner in eine oder mehrere Expertenkategorien und aktiviert dann die entsprechenden Modelle. Was noch fehlt, ist ein vorgeschaltetes *Komplexitäts-Pre-Routing*: eine billige, deterministische Abschätzung, ob eine Anfrage überhaupt ein grosses Modell benötigt, oder ob ein kleines, lokal lauffähiges Modell (im Extremfall ein 3-Milliarden-Parameter-Modell auf einer CPU) ausreicht. Die Heuristik müsste auf Merkmalen wie Eingabelänge, Anzahl der verlangten Expertenkategorien, Vorhandensein eines L1-Cache-Hits und – wenn verfügbar – einer impliziten Komplexitäts-Einstufung aus dem Planner-Output beruhen.

Das Ziel ist doppelgleisig: Kosten (Strom, Latenz) senken und die Abhängigkeit von schweren GPUs reduzieren. Wir erwarten, dass ein wohldefinierter Pre-Routing-Schritt die Zahl der tatsächlichen GPU-Inferenzen bei einfachen Workloads (Übersetzung kurzer Texte, Formularausfüllung, Small-Talk) drastisch senken kann.

15.3 Verteilte Multi-Knoten-Inferenz

Die jetzige Installation geht davon aus, dass jedes Modell auf einem Inference-Endpunkt einmal komplett vorhanden ist (Ollama-Instanz). Für sehr große Modelle, die die VRAM-Kapazität eines einzelnen Nodes übersteigen, reicht das nicht. Wir sehen drei Wege, die in zukünftigen Versionen exploriert werden sollen:

1. **Pipeline-Parallelismus** über mehrere GPUs auf demselben Host (mit vLLM [22] oder ähnlichen Engines).

2. **Tensor-Parallelismus** über mehrere Hosts hinweg – hier existieren produktionsreife Open-Source-Umgebungen, die wir in die Inference-Backend-Abstraktion integrieren müssen.

3. **Föderierte Inferenz**: mehrere unabhängig betriebene Orchestrator-Installationen teilen sich eine gemeinsame Template-Registry und eine gemeinsame *Cross-Tenant Cache-Bridge*. Dies ist das langfristig interessanteste Szenario, weil es dem Geist der digitalen Souveränität am nächsten kommt: keine einzelne Installation wird zum Bottleneck.

15.4 Föderierte Wissensgraphen

Die in v1.0 dieses Papers skizzierte Architektur für föderierte Wissensgraphen wurde als *MoE Libris* (Abschnitt 8.8) realisiert. Das Hub-and-Spoke-Protokoll, die Pre-Audit-Pipeline, die graduierte Missbrauchsprävention und die Trust-Floor-Integration sind implementiert und betriebsbereit.

Offene Forschungsfragen bleiben: (1) Cross-Hub-Topologie – die aktuelle Implementierung unterstützt einen einzelnen Hub pro Knoten; Mesh- oder hierarchische Multi-Hub-Föderation erfordert eine Konfliktlösungsstrategie für Triples, die von verschiedenen Hubs mit divergierenden Trust-Scores eintreffen; (2) formale Verifikation der Vertrauenspropagation über Föderationsgrenzen hinweg, insbesondere ob die Trust-Floor-Deckelung (Standard 0.5) ausreicht, um transitive Vertrauensinflation in Netzwerken mit vielen teilnehmenden Knoten zu verhindern; (3) kryptographische Bundle-Signierung (Ed25519) zur Manipulationserkennung im Transit, die spezifiziert, aber noch nicht implementiert ist.

15.5 Compliance-Erweiterung: MoE Codex

Die EU-Compliance- und Audit-Anwendungsfälle – Datenherkunft, Freigabe-Workflows, versionierte Daten-Bundles und Drift-Erkennung – wurden als *MoE Codex* (Abschnitt 5.7) realisiert. Der vollständige Stack aus fünfundzwanzig Modulen über die Tracks D.0 bis D.5 ist betriebsbereit und als separates Apache-2.0-Repository mit vollständiger Compliance-Dokumentation für DSGVO Art. 32/35, EU KI-Verordnung Risikoklassen, BSI-Grundschutz und NIS2 veröffentlicht.

Die Tracks D.1 bis D.5 sind implementiert und freigegeben. Die zuvor offenen Fragen – Kestra-basierte Pipeline-Orchestrierung, OpenSearch-föderierte Suche und geospatiale Untersuchungswerkzeuge – sind nun operativ. Verbleibende offene Fragen: LLM-gestützte semantische Drift-Narration jenseits statistischer Schwellwerte; ArgoCD GitOps für automatisierte Multi-Environment-Deployments; und KeplerGL-Ontologie- Bindung für objekt-überlagerte geospatiale Untersuchungen.

15.6 SLM-Skalierungsgrenzen unter GraphRAG (Wikimedia-Benchmarks)

Um systematisch zu untersuchen, wie sich extern strukturiertes Wissen auf die Fähigkeiten extrem kleiner Modelle auswirkt, ist der Import großer Wikimedia-Graph-Dumps in die Neo4j-Datenbank geplant. Ziel ist die Evaluation, ob Small Language Models (SLMs) bis zu 9B Parametern durch den Zugriff auf ein GraphRAG-System signifikant bessere und halluzinationsärmere Antworten liefern können als im reinen Zero-Context-Betrieb.

Diese Forschungsarbeit soll die absolute Untergrenze der logischen Argumentationsfähigkeit von SLMs unter GraphRAG bestimmen. Hierzu wird ein systematischer Benchmark über eine absteigende Parameterreihe durchgeführt: 9B, 7B, 4B, 3B, 1.5B und 0.3B. Durch Messung des Qualitätsabfalls pro Stufe soll die Grenze ermittelt werden, an der die inhärente logische Fähigkeit eines Modells nicht mehr ausreicht, um die bereitgestellten Graph-Zusammenhänge sinnvoll zu interpretieren und in kohärente Antworten zu überführen.

15.7 Energie-Reporting pro Anfrage

Eine Eigenschaft, die wir in vielen Privacy-by-Design-Katalogen vermissen, ist *Energie-Transparenz*. Jede Anfrage kostet Strom; jede Strommenge kostet CO_2. Ein zukünftiges Feature, das wir für besonders wichtig halten, ist eine Prometheus-Metrik `moe_energy_joules_total`, die pro Anfrage die geschätzte Energiemenge ausweist, auf Basis der GPU-Auslastung, der Modell-Parameter und der Anzahl der verarbeiteten Token. Die Formeln sind in der Literatur verfügbar; die Integration in die Observability-Pipeline ist Arbeit für die nächsten Monate.

15.8 Mehrsprachige User-Interfaces

Das Admin-UI ist aktuell in vier Sprachen lokalisiert (Deutsch, Englisch, Französisch, Chinesisch). Die verbleibende Arbeit betrifft die konsistente Terminologie über alle Sprachen und die Erweiterung um weitere europäische Sprachen (Spanisch, Italienisch, Niederländisch, Polnisch), sobald Community-Beiträge zur Verfügung stehen.

15.9 Hyper-Skalierung auf Enterprise-Hardware

Die schlanke Architektur von MoE Sovereign ist portabel: dieselben Prinzipien, die auf Tesla-M10-Clustern funktionieren, skalieren auf H100-Systemen nicht linear, sondern *super-linear*. Drei Effekte treten gleichzeitig ein:

1. **Nahezu-Null-Latenz für triviale Anfragen.** Die L0- und L1-Cache-Schichten (Redis + ChromaDB) liefern Cache-Hits in unter 5 ms – unabhängig von der Inferenz-Hardware. Auf H100-Systemen bedeutet das: der Großteil der Anfragen im Produktivbetrieb wird überhaupt keine GPU-Ressourcen verbrauchen.

2. **100 % Frontier-Compute für echtes Reasoning.** Weil triviale und moderate Anfragen durch Caching und leichte Tier-1-Modelle abgefangen werden, steht die volle H100-Rechenleistung exklusiv für Tier-2/Tier-3-Tasks zur Verfügung – komplexes Multi-Hop-Reasoning, Syntheseaufgaben, Agentic Loops. Kein Compute wird für Anfragen verschwendet, die ein gecachtes Muster wiederholen.

3. **Massive Concurrent-User-Kapazität.** Nicht-optimierte LLM-Stacks reservieren einen kompletten Inference-Slot pro Request, unabhängig von dessen Komplexität. MoE Sovereign verteilt Last nach tatsächlichem Bedarf: Cache-Hits, Tier-1-Modelle und Deterministic-Tool-Calls erzeugen keinen GPU-Druck. Das Resultat ist eine deutlich höhere Anzahl von Concurrent Users pro Knoten im Vergleich zu monolithischen Inference-Deployments.

Das Apollo-11-Prinzip lautet: Maximale Präzision bei minimalem Ressourcenverbrauch – erreichbar durch Architektur, nicht durch Hardware-Investition. Der Übergang auf H100-Systeme multipliziert die Kapazität, ohne die Architektur zu ändern.

15.10 Gefördertes SLM-Distillation-Programm (EuroHPC LUMI-G)

Im Juni 2026 wurde der Antrag EHPC-DEV-2026D06-XXX im Rahmen des EuroHPC-Development-Access-Programms bewilligt (Award-Bescheid erhalten am 2026-06-05): 4.500 Node-Stunden (18.000 GPU-Stunden) auf der **LUMI-G**-Partition (AMD MI250X, 512 GB HBM2e pro physischem Knoten – 4× MI250X-Module à 128 GB –, ROCm-Stack, 2 TB Storage) über einen Zeitraum von sechs Monaten. Die Förderung finanziert ein systematisches Distillation-Programm, das die fünf wirkungsvollsten LLM-abhängigen Entscheidungspunkte im Orchestrierungs-Graphen adressiert und Cloud-geroutete Inferenz dort durch lokal ausführbare Small Language Models (SLMs) ersetzt, wo die Distillation eine ausreichende Entscheidungsqualität erhält.

Tabelle 31 fasst die fünf Distillation-Ziele zusammen. Der Haupthebel ist der `planner_node` (Abschnitt 5, der GAIA-artige Task-Planner): Ziel ist ein `Qwen3.5-4B`-Modell, quantisiert als GGUF Q4_K_M / Q5_K_M, das mindestens 90 % der GAIA-Planqualität des Qwen3.6-35B-A3B-Lehrermodells bei rund 1/10 der Inferenzkosten und 200–400 ms CPU-Latenz erreicht. Die übrigen vier Ziele distillieren unterstützende Entscheidungen – Komplexitätsabschätzung, semantisches Routing, RL-basiertes Experten-Routing und Node-Ranking – in Modelle mit unter 10 ms Inferenzzeit, die dauerhaft ohne GPU-Konkurrenz laufen können.

Die bewilligten 18.000 GPU-Stunden sind auf fünf Phasen verteilt: synthetische Datengenerierung (2.500 h, Monate 1–2), Encoder- und Reward-Model-Training (800 h, Monat 2), Planner-SFT, DPO und Ablationen (9.000 h, Monate 2–4), Offline-RL für die Routing-Policy (3.500 h, Monate 4–5) und ein abschließender RLHF-Durchlauf (2.200 h, Monate 5–6). Der Rollout wird über das Feature-Flag `PLANNER_MODE` mit drei Einstellungen gesteuert – `llm`, `slm_local` und `hybrid` –, wobei `hybrid` bei Unterschreiten eines konfigurierbaren Konfidenz-Schwellwerts auf das Cloud-Lehrermodell zurückfällt. Das spiegelt dieselbe Fail-Open-Philosophie, die bereits beim Experten-Routing (Abschnitt 6) angewendet wird: Das SLM ist eine Kosten- und Latenzoptimierung, niemals eine harte Abhängigkeit.

Bis zum 19. Juli 2026 wurden in der Implementierungsphase insgesamt **437 GPU-Stunden** tatsächlich verbraucht (≈2,4 % des bewilligten Gesamtbudgets von 18.000 GPU-Stunden). Dieser Verbrauch teilt sich auf in ≈96 h für Job 19832821 (12 h-Limit, bis Checkpoint-704), ≈172 h für Resume-Job 19926224 (vollständiger Trainingsdurchlauf, alle 2112 Schritte), ≈54 h als direkter Verlust durch ROCm/BitsAndBytes-Treiberkonflikte und Modell-Serialisierungsfehler, ≈78 h für die W4A16-GPTQ-Quantisierung (Job 19978657, AMD MI250X) sowie ≈37 h für Merge-Job 19964923 und GGUF-Konvertierung.

Das Training des parakonsistenten Judge-Modells (`Qwen/Qwen3.6-35B-A3B`) wurde am 16. Juli 2026 um 21:15 Uhr erfolgreich abgeschlossen. **Job 19832821** (12 h-Limit) hatte am 13. Juli 2026 nach Schritt 781 von 2112 (37 % der zweiten Epoche) durch das Wall-Clock-Limit terminiert und Checkpoint-704 (Epoche 1,0, Loss 0,485, Token-Accuracy 85,7 %) gesichert. **Resume-Job 19926224** (24 h-Limit) setzte das Training ab Checkpoint-704 fort und schloss alle verbleibenden Schritte ab. Endmetriken (Epoche 3): **Loss 0,3032**, **Token-Accuracy 86,72 %**, Entropie 0,4447, Gradientennorm ≈0,08. Das Modell wurde anschließend in nativer `bfloat16`-Präzision zusammengeführt (Merge-Job 19964923, 17. Juli 2026), als W4A16-GPTQ quantisiert (Job 19978657, 9 h 41 min, 18. Juli 2026) und als GGUF Q4_K_M konvertiert (19. Juli 2026, `-no-mtp`-Flag erforderlich, vgl. Episode 10). Das bereinigte Modell läuft unter dem Alias `sovereign-judge:35b-q4km` auf dem lokalen Ollama-Server mit `num_ctx=262144`.

Status des Judge-Fine-Tunings

Training, Merge, Quantisierung und lokale Bereitstellung sind abgeschlossen. Eine kontrollierte Downstream-Evaluation gegen das unveränderte Basismodell über die Artefaktstufen BF16, GPTQ und GGUF steht noch aus. Aus den Trainingsmetriken (Loss 0,3032, Token-Accuracy 86,72 %) allein wird keine Verbesserung der Judge-Qualität abgeleitet; diese erfordert eine separate Evaluation auf T/F/B/N-Klassifikationsaufgaben und End-to-End-Orchestrierungsläufen.

vLLM-Inferenz-Performance auf AMD MI250X. Als Grundlage für die Datensatz-Synthese der IMoE-Routing-Daten wurde **vLLM v0.20.1** mit `Qwen/Qwen2.5-32B-Instruct` (TP=2, bfloat16) auf einem physischen LUMI-G-Host unter Nutzung von 2 GCDs (ein MI250X-Modul, 128 GB HBM2e von 512 GB knotenweitem Gesamt-HBM) betrieben. Die gemessenen Leistungswerte sind in Tabelle 32 aufgeführt.

Die Startzeit von 265 Sekunden ist für HPC-Batch-Jobs akzeptabel (SLURM-Overhead dominierende Variable bei kurzen Jobs), für interaktive Produktivsysteme aber grenzwertig. Der Decode-Durchsatz von ≈22 tok/s entspricht dem erwarteten Wert für ein 32B-Modell bei TP=2 auf dieser Hardware. Die fehlende Flash-Attention-2-Unterstützung (ROCm-Stack) schlägt sich in rund 10–15 % Overhead gegenüber CUDA-äquivalenten Konfigurationen nieder.

Parallel dazu wächst der zugrunde liegende Wissensgraph rapide (46× Entitäten, 56× Relationen in 16 Tagen Anfang 2026), und der nächste Skalierungsmeilenstein ist die erste Multi-Hub-MoE-Libris-Föderation (siehe Abschnitt zu föderierten Wissensgraphen), die die föderierte Architektur in der oben genannten Größenordnung von 25k–100k Relationen validieren wird.

15.11 Warum wir nicht monetarisieren

Eine häufig gestellte Frage ist, ob das Projekt nicht zumindest im Enterprise-Segment monetarisiert werden sollte – etwa durch Support-Verträge oder Managed-Offerings. Unsere Antwort ist: wir haben uns bewusst gegen diesen Weg entschieden. Der Grund ist nicht Ideologie, sondern Architektur: *sobald ein Projekt Monetarisierung als Ziel einführt, beginnt die technische Architektur sich danach auszurichten.* Features werden in eine Free- und eine Pro-Ebene geteilt; Security-Fixes erscheinen mit Verzögerung in der Free-Edition; Dokumentation wird strategisch auf den Verkaufspfad optimiert. Diese Effekte sind in der Open-Source-Welt hinreichend dokumentiert.

Wir schliessen Kooperationen, Spenden oder Forschungs-Grants nicht aus. Wir schliessen eine Enterprise-Edition aus. Wer produktionsreifen Support braucht, ist eingeladen, ihn auf Basis der dokumentierten Architektur selbst anzubieten – als Drittanbieter, ohne Verzerrung des Upstream-Projekts.

15.12 Offene Einladung

Dieses Whitepaper schliesst mit einer expliziten Einladung an drei Gruppen:

- **Forscherinnen**: das Projekt eignet sich als Experimentierplattform für Routing-Algorithmen, Cache-Strategien, Hybrid-RAG-Designs und Federated-Learning-Prototypen. Die Architektur ist dokumentiert, der Code ist Open Source, die Testsuite erlaubt eine schnelle Iteration.

- **Praktikerinnen in regulierten Domänen**: wer einen konkreten, nachvollziehbaren Weg zur DSGVO-kompatiblen LLM-Infrastruktur sucht, findet hier einen Startpunkt. Wir sind ausdrücklich an Feedback und realistischen Anforderungen aus der Praxis interessiert.

- **Beitragende aus der Community**: Issues, Pull-Requests, Übersetzungen, Testberichte, Dokumentations-Verbesserungen – alles ist willkommen. Wir betreiben das Projekt nach dem „Bazaar"-Modell [47]: schnell veröffentlichen, früh iterieren, transparent scheitern, öffentlich lernen.

16 Fazit

Wir haben mit MoE Sovereign ein Framework vorgestellt, das versucht, einen spezifischen Punkt im Design-Raum moderner LLM-Infrastruktur zu besetzen: *deterministisch geroutet, lokal gehostet, mit compoundender Wissensbasis, über heterogene Hardware und verschiedene Deployment-Schichten hinweg einheitlich betriebsfähig, und bewusst nicht-kommerziell.* Dieser Punkt war unseres Wissens nach bisher nicht besetzt, und wir halten ihn für architektonisch lohnenswert und gesellschaftlich notwendig.

Die technischen Hauptbeiträge lassen sich in einer Handvoll Sätze zusammenfassen: Expert-Templates ersetzen gelernte Router und machen Routing-Entscheidungen auditierbar; eine vierschichtige Cache-Hierarchie trennt semantische Ähnlichkeit von Plan-Äquivalenz, Graph-Kontext und historischer Zuverlässigkeit; ein GraphRAG-Ansatz mit kategoriespezifischen Entity-Filtern verhindert systematisch Cross-Kontamination zwischen Fachdomänen; ein MCP-Server mit einer AST-Whitelist und 23 deterministischen Werkzeugen entzieht präzisionskritische Operationen der probabilistischen Modell-Logik; und ein Universal-Deployment-Modell mit einem einzigen OCI-Image, drei Profilen und vier Wrappern hält den Bogen vom Hobby-LXC bis zum OpenShift-Cluster ohne Code-Fork.

Wir haben im Text offen über elf dokumentierte Fehler- und Lernkurven-Episoden berichtet: von der Rendering-Regression und der SymPy-Injection-Lücke über den SQLite-Concurrent-Access-Bug und den zu cleveren ersten Scheduler bis hin zum Specification-Gaming-Vorfall beim GAIA-Benchmark, dem DeepSpeed-ZeRO-3-Multimodal-Fix auf AMD MI250X, dem ChromaDB-Cache-Miss, dem PEFT-Versionsmismatch, dem GGUF-`--no-mtp`-Flag und dem wirkungslosen Planner-SFT durch Sequenzlängen-Truncation. Die meisten sind behoben; wo die Behebung noch aussteht, ist der Analysestand dokumentiert. Die Offenlegung dieser Episoden dient einem doppelten Zweck: Contributorinnen vor dem gleichen Fehler zu warnen und die wissenschaftliche Integrität des Papers zu wahren.

Der letzte, entscheidende Punkt ist die Haltung. *Digitale Souveränität* ist kein Marketing-Slogan, sondern ein technisch erreichbarer Zustand: jede Komponente lokal, jede Abhängigkeit benannt, jeder Datenfluss dokumentiert, jede Entscheidung reversibel, jeder Fehler öffentlich. Wir glauben, dass Open-Source-Projekte dieser Art kein Geschäftsmodell brauchen, um relevant zu werden – sie brauchen Klarheit, Ehrlichkeit und ein belastbares Fundament, auf dem andere weiterbauen können.

MoE Sovereign wird ab dem Veröffentlichungsdatum dieses Papers frei zugänglich sein: der Quellcode unter Apache 2.0, dieses Whitepaper unter CC BY-SA 4.0 [12] – auf Philipp Horns Entwicklerseite und im dazugehörigen Git-Repository. Die Einladung zur Mitarbeit ist offen und explizit. Wer sich dem Projekt anschliessen möchte – als Leserin, als Anwenderin, als Contributor – ist eingeladen, den gleichen Weg zu gehen, den wir beschreiten: sorgfältig bauen, transparent dokumentieren, ehrlich lernen, kollektiv besitzen.

Mit *MoE Libris* (Abschnitt 8.8) hat MoE Sovereign den Schritt vom alleinstehenden Export-/Import-Mechanismus für Community-Wissens-Bundles zu einem vollständig operativen **föderierten Wissensaustausch** vollzogen. MoE Libris implementiert ein Hub-and-Spoke-Föderationsprotokoll – bilateraler Handshake, zweistufige Pre-Audit-Pipeline, graduierte Missbrauchsprävention, Auditierungswarteschlange und Trust-Floor-gedeckelter Import –, das souveränen Knoten den Austausch von Domänenwissen ermöglicht, ohne proprietäre Daten preiszugeben. Der daraus resultierende **Netzwerkeffekt** bedeutet, dass jede neue Installation die kollektive Intelligenz aller Teilnehmer bereichert. Die Architektur orientiert sich explizit am Fediverse-Modell: unabhängige Instanzen föderieren freiwillig über ein standardisiertes Protokoll, und kein einzelner Hub hat Autorität über einen Knoten. Dies ist die architektonische Grundlage für dezentrale KI, die mit der Community skaliert – nicht mit Compute-Budgets.

Die Quintessenz in einem Satz

Souveräne KI-Infrastruktur ist kein Produkt, das man kauft, und kein Ziel, das man erreicht – sie ist eine Praxis, die man jeden Tag neu bestätigt, indem man die Kontrolle nicht abgibt.

Philipp Horn
Ascheberg, Juli 2026

Tabelle 5: Wissenschaftliche Quellen direkt implementiert in MoE Sovereign (gesamter Quellcode).

Autoren	**Jahr**	**Übernommenes Konzept**	**Implementierung in MoE Sovereign**
de Vries, Andreas [29]	2007	Algebraische Gitterhierarchie: einheitlicher Rahmen, der parakonsistente, fuzzy-, Quanten-, intuitionistische und Boolesche Logiken in eine gemeinsame Ordnungsstruktur einbettet; algebraische Motivation für widerspruchstolerantes Schließen	`pipeline/state.py`: `conflict_registry`; `graph/synthesis.py`: `resolve_conflicts_node` (algebraische Motivation)
Belnap, Nuel D. [30]	1977	Vierwertige Semantik $\{T, F, B, N\}$ (*Wahr*, *Falsch*, *Beide*, *Keines*): Widersprüche werden als B lokalisiert, ohne *ex contradictione quodlibet* auszulösen	`resolve_conflicts_node`: formale Grundlage für Konfliktklassifikation (Strategie A Dismiss vs. Strategie B Arbitration); $\{T, F, I, U\}$-Wahrheitswertelabels im 90 000-Sample-Trainingsdatensatz
Gödel, Kurt [31]	1932	Gödel-T-Norm $T_G(a, b) = \min(a, b)$: konservativste Fuzzy-Konjunktion	Standard-Konjunktion in `fuzzy_router_node`: kombiniert `vector_confidence` und `graph_confidence` per min; Ergebnis ist stets durch das schwächere Signal begrenzt
Łukasiewicz, Jan [32]	1920	Łukasiewicz-T-Norm $T_Ł(a, b) = \max(0, a+b-1)$: tolerante Fuzzy-Konjunktion	Alternative Konjunktion in `pipeline/logic_types.py` (`lukasiewicz_tnorm`); per Template-Konfiguration wählbar
Kolmogorov, Andrei N. [33]; Chaitin, Gregory J. [34]	1965 / 1966	Algorithmischer Informationsgehalt (Kolmogorov-Komplexität): zlib-Länge als praktische obere Schranke	`complexity_estimator.py`: `_aic_compressibility()` nutzt zlib-Kompressionsrate zur Klassifikation von Anfragen in `trivial`/`moderate`/`complex` ohne LLM-Aufruf
Ratcliff, John W.; Metzener, David E. [35]	1988	Ratcliff/Obershelp-Algorithmus: iterative longest-common-substring-Ähnlichkeit	`graph_rag/manager.py`: `_fuzzy_resolve_entity_name()` via Python `SequenceMatcher` (Schwellwert 0,82) kanonisiert Entitätsnamen vor Neo4j-MERGE und verhindert Duplikate
Anthropic [6]	2024	Model Context Protocol (MCP): Standard für deterministische Tool-Exposition an LLM-Agenten	`mcp_server/server.py`: 51 deterministische Werkzeuge hinter einer AST-Whitelist (Arithmetik, Hashing, Subnetz, Datum, Regex, PPTX, Wikidata, PubMed, Wayback, Schach…); präzisionskritische Operationen außerhalb probabilistischer Modelllogik

Tabelle 8: AMD MI250X vs. NVIDIA A100/H100: empirische Trainingsunterschiede beim LLM-Fine-Tuning auf LUMI-G.

Aspekt	**NVIDIA A100/H100 (CUDA)**	**AMD MI250X (ROCm/LUMI-G)**	**Implikation**
4-bit QLoRA (BitsAndBytes)	✓ Stabil, weit verbreitet	× Instabil (kein stabiler ROCm-Build)	Auf AMD: native BF16 + ZeRO-3 statt QLoRA
Flash Attention 2	✓ Nativ (`flash_attn`)	× Nicht verfügbar	`attn_implementation=ëager"` erzwungen (≈10–15 % Overhead)
ZeRO-3 + multimodale Modelle	Selten nötig (QLoRA ausreichend)	Primäre Strategie für >30 B	Vision Tower *vor* LoRA-Wrap einfrieren
`device_map=äuto"`	✓ Standard	× OOM durch unbalancierte Layer-Verteilung	`device_map=None` + ZeRO-3 erzwingen
`torch_dtype`	FP16 oder BF16 stabil	Nur BF16 stabil	`torch_dtype=torch.bfloat16`
Transformers-Modell-Support	Immer aktuell (CUDA-First)	Container-Image oft 2–4 Wochen verzögert	Neue Architekturen per venv nachrüsten

Tabelle 9: Reale Token-Verteilung dreier Trainingsbeispiele (`Qwen3-8B`-Tokenizer) gegen `max_seq_len=1536`.

Beispiel	**System-Prompt**	**System+User**	**Assistant-JSON beginnt bei**
Sample 0	2 509 Token	2 527 Token	Token-Position ≈2 536
Sample 1	2 514 Token	2 538 Token	Token-Position ≈2 547
Sample 2	2 495 Token	2 513 Token	Token-Position ≈2 522

Tabelle 10: Drei divergierende Planner-Prompts im deployten System.

Quelle	**Länge**	**Charakteristik**
Trainings-Prompt (`PLANNER_SYSTEM_PROMPT`)	≈2 500 Tok.	16 Kategorien inkl. `research` und `dynamic`; Pipeline-Beschreibung, 22-Tool-Katalog, 9 Beispiele
Code-Fallback (`DEFAULT_PLANNER_ROLE`)	415 Zeichen	generischer 3-Satz-Stub, aktiv nur wenn kein Template hinterlegt ist
Produktiv genutztes Template (GAIA-Benchmark-Profil)	13 987 Zeichen	`web_researcher` statt `research`; neue Kategorie `memory_recall` (im Training unbekannt); *keine* `dynamic`-Kategorie

Template	**Wall (s)**	**in tok**	**out tok**	**T1**	**T2**	**Pipeline**
`tpl-ref-8b`	523.5	7147	3435	2	1	complete
`tpl-ref-30b`	360.6	4282	6469	2	2	complete
`tpl-ref-70b`	1414.1	4005	5910	2	2	complete

Tabelle 11: Baseline-Benchmark: drei Referenz-Templates gegen den selben Multi-Domain-Prompt (DSGVO + Math + Technical). Alle Läufe gingen durch den vollständigen Orchestrator-Pipeline (Planner → parallele T1-Experten → Self-Correction → GraphRAG-Ingest → Merger → Judge).

Run	Score	L1	L2	L3
1 – Baseline dieser Iteration	11/30=37%	6/10	3/10	2/10
2 – **BEST**: Planner-Regeln	**14/30=47%**	6/10	5/10	3/10
3 – Tool-Gruppen-Fix	10/30=33%	5/10	4/10	1/10
4 – Cache durch Pre-Warm vergiftet	11/30=37%	6/10	3/10	2/10
5 – Confidence-Gate-Fix revertiert	11/30=37%	6/10	4/10	1/10
GPT-4o Mini (Referenz)	44,8% gesamt			

Tabelle 12: GAIA Validation Set – 5 Runs, 2026-04-25. Level 1–3 je 10 Fragen. Gemessenes Ergebnis: **14/30 = 46,7%** mit Template `moe-aihub-free-gremium-deep-wcc`. *Hinweis: nachträglich als Specification Gaming eingestuft (Abschnitt 14.6); kein valides Sovereign-Stack-Ergebnis.* GPT-4o Mini Referenz: 44,8%.

Konfiguration	**Score**	**L1**	**L2**	**L3**	**Besonderheit**
gpt-oss-120b (Best)	14/30 = 46,7%	6/10	5/10	3/10	Baseline Best
qwen-3.5-122b (Planner+Judge)	9/30 = 30,0%	5/10	3/10	1/10	−16,7 pp

Tabelle 13: Judge-Experiment: Modell-Vergleich auf dem GAIA Validation Set (30 Fragen, Levels 1–3). `qwen-3.5-122b-sovereign` als Planner+Judge vs. Baseline `gpt-oss-120b-sovereign`.

Test	**Kategorie**	**Det.**	**LLM**	**Komb.**
Subnet-Berechnung	MCP-Precision	7.0	10.0	**8.8**
Arithmetik + Einheiten	MCP-Precision	10.0	9.0	**9.4**
Datumsrechnung (Schaltjahr)	MCP-Precision	10.0	10.0	**10.0**
3-Turn Memory	Persistentes Graph-State-Tracking	5.5	9.0	**7.6**
5-Turn Memory	Persistentes Graph-State-Tracking	2.3	0.0	0.9
Rechtsfrage (BGB)	Domain-Routing	0.0	8.0	**4.8**
Medizinfrage (Hashimoto)	Domain-Routing	9.4	0.0	3.8
Code-Review (SQL Inj.)	Domain-Routing	7.6	10.0	**9.0**
Multi-Expert-Synthese	Multi-Expert	2.3	0.0	0.9
Durchschnitt				**6.1**

Tabelle 14: MoE-Eval v1: kognitive Benchmark-Ergebnisse mit dem 30b-Balanced-Template. Det. = deterministischer Score (0–10), LLM = `gpt-oss:20b` Judge-Score via direktem Ollama-Call, Komb. = $0.4 \times$ Det. $+ 0.6 \times$ LLM.

Template	Wall-clock (s)		Δ	Ext. Score		Δ
	Run 1	Run 2		Run 1	Run 2	
`tpl-ref-8b`	523.5	578.2	+10 %	8.0	8.0	±0
`tpl-ref-30b`	360.6	**304.4**	**−16 %**	—	**9.0**	—
`tpl-ref-70b`	1414.1	**641.5**	**−55 %**	—	**9.0**	—

Tabelle 15: Vorher/Nachher-Vergleich: Baseline-Benchmark Run 1 vs. Run 2. Run 2 lief unmittelbar nach Run 1 auf derselben Infrastruktur ohne Code-Änderungen. Die Verbesserungen beim 30b (−16 %) und 70b (−55 %) Template spiegeln den Akkumulations-Effekt wider: Few-Shot-Korrekturen aus dem Critic-Node, ein dichterer Neo4j-Graph (+20 Triples) und Modell-Warmth reduzieren die Pipeline-Durchlaufzeit. Die Qualitäts-Scores (External Judge: `llama3.3-70b-ctx4k`) bleiben stabil oder verbessern sich: das 30b-Template erreicht 9.0/10, das 70b-Template ebenfalls 9.0/10.

Template	Prec.	Comp.	Rout.	M-E	∅
`ref-30b`	9.6	4.5	8.4	5.7	7.6
`n04-rtx`	7.6	4.5	5.9	4.9	6.0
`n07-n09`	6.0	0.0	7.8	0.0	4.6
`n06-m10`	1.9	4.2	5.3	0.0	3.3
`n11-m10`	3.5	1.8	5.3	1.9	3.6

Tabelle 16: Dense-Graph-Run (15. April 2026): Kategorie-Durchschnitte (0–10) nach Abschluss der Evaluierung. Prec. = MCP-Precision, Comp. = Graph-State-Tracking Memory, Rout. = Domain-Routing, M-E = Multi-Expert Synthesis.

Datum / Template	Graph	Prec.	Comp.	Rout.	M-E	∅
10. Apr. ref-30b (Lauf 1)	≈500	7.6	4.1	5.0	0.9	5.2
10. Apr. ref-30b (Lauf 2–4)	≈800	9.3	3.9	5.8	0.9	6.0
12. Apr. ref-30b	≈2.000	8.3	4.4	7.6	5.1	6.8
15. Apr. ref-30b	5.353	9.6	4.5	8.4	5.7	**7.6**
15. Apr. n04-rtx (RTX-only)	5.353	7.6	4.5	5.9	4.9	6.0
15. Apr. n07-n09 (GT1060+M60)	5.353	6.0	0.0	7.8	0.0	4.6
15. Apr. n06-m10 (M10×4)	5.353	1.9	4.2	5.3	0.0	3.3
15. Apr. n11-m10 (M10×4)	5.353	3.5	1.8	5.3	1.9	3.6

Tabelle 17: Vollständige MoE-Eval-Messreihe April 2026. Prec. = MCP-Precision, Comp. = Graph-State-Tracking Memory, Rout. = Domain-Routing, M-E = Multi-Expert Synthesis. Alle Scores 0–10, kombiniert aus deterministischem Score (40 %) und LLM-Judge-Score phi4:14b (60 %).

Modell	Rolle	Planner	Judge
`phi4:14b`	Planner + Judge	✓	✓
`devstral:24b`	Planner + Judge	✓	✓
`qwen3-coder:30b`	Planner + Judge	✓	✓
`gemma3:27b`	Planner + Judge	✓	✓
`mistral-small:24b`	Planner + Judge	✓	✓
`nomic-embed-text`	Embedding	×	×
`llama-guard3:8b`	Guard	×	×
`x/z-image-turbo`	Bildgenerierung	×	×
`deepseek-r1:32b`	Reasoning	×	×
`gpt-oss:20b`	TTL-Fehler	×	×

Tabelle 18: Top-5 geeignete und 5 gescheiterte Modelle aus dem Rollentauglichkeitstest (n=69).

Aufgabe	Native	Reasoning	Orchestrated
SQL-Injection-Fix	435 s / 14.7K	510 s / 14.9K	**315 s / 9.2K**
Health-Endpoint	426 s / 11.0K	**320 s / 10.1K**	481 s / 12.8K
DB-Refactoring	468 s / 12.3K	326 s / 10.5K	**188 s / 8.4K**
Pytest-Tests	394 s / 11.0K	322 s / 9.4K	**193 s / 9.8K**
Security-Review	361 s / 9.8K	354 s / 10.8K	**179 s / 8.7K**
Durchschnitt	417 s / 11.8K	366 s / 11.1K	**271 s / 9.8K**

Tabelle 19: Claude Code Profile Benchmark: Latenz (Sekunden) / Token-Verbrauch pro Profil und Aufgabe. Native = direktes LLM ohne Pipeline, Reasoning = mit Thinking-Node, Orchestrated = volle MoE-Pipeline (Planner → Experten → Merger → Judge → GraphRAG). Alle Tests auf `gemma4:31b` (N04-RTX).

Tabelle 20: Deployment-Reife (Stand April 2026).

Ziel	Status	Hinweise
Docker Compose	Getestet	Primäre Methode. Produktiv auf 5-Knoten-GPU-Cluster.
LXC / Proxmox	Getestet	Docker-in-LXC mit `nesting=1, fuse=1`.
Podman (rootless)	Geplant	Vorbereitet, noch nicht validiert.
K3s	Geplant	Helm-Chart vorbereitet. Longhorn empfohlen.
Kubernetes	Ungetestet	Manifeste bereitgestellt; Community-Validierung willkommen.
OpenShift	Ungetestet	SCC-Konfiguration dokumentiert, mangels Zugang nicht validiert.

Tabelle 21: Vergleich: `moe-reference-30b-balanced` vs. `moe-m10-8b-gremium`, Multi-Domain-Challenge-Prompt, 2026-04-15.

Metrik	ref-30b	m10-gremium
Deterministischer Score	**6,67 / 10**	**4,29 / 10**
Laufzeit	528 s	2.542 s
Tokens (Eingabe)	15.875	31.926
Tokens (Ausgabe)	14.615	8.172
Aufgerufene Experten	mehrere	1 (legal_advisor)
Planner-Fehlversuche	0	2

Tabelle 22: Planner-Versuche im m10-gremium-Lauf.

Versuch	Dauer	Ergebnis
1	≈11 min	Fließtext – parse error, retry
2	≈8 min	Fließtext – fallback gesetzt
3	≈9 min	**Gültiges JSON** (nur legal_advisor geroutet)

Tabelle 23: Template `moe-m10-gremium-deep` – Komponentenübersicht.

Rolle	Modell	Knoten	Auswahlgrund
Planner	phi4:14b	N04-RTX	16k-Kontext, Flash Attention, nur Routing – kein GraphRAG
Judge	phi4:14b	N04-RTX	16k-Kontext, empfängt ≤12.000 Chars GraphRAG
code_reviewer	qwen2.5-coder:7b	N06-M10-01	SWE-bench SOTA 7B (Alibaba)
math	mathstral:7b	N06-M10-02	MATH-Benchmark SOTA 7B (Mistral AI)
medical_consult	meditron:7b	N06-M10-03	MedQA über GPT-3.5 (EPFL 2023)
legal_advisor	sauerkrautlm-7b-hero	N06-M10-04	Bestes Deutsch-Rechts-7B, 32k Kontext
reasoning	qwen3:8b	N11-M10-01	GPQA-Anführer <8B (Alibaba 2025–2026)
science	gemma2:9b	N11-M10-02	71.3 % MMLU (Google)
translation	qwen2.5:7b	N11-M10-03	Stärkstes westeuropäisches Multilingual 7B
technical_support	qwen2.5-coder:7b	N11-M10-04	Strukturierte Ausgabe + MCP-Tool-Calling

Tabelle 24: Epochenübersicht – Run `overnight_20260419-225041`.

Epoche	Dauer	Szenarien	RC	Score
E1	4h 11min	12 / 12	0	6.53 / 10
E2	3h 5min	12 / 12	0	5.78 / 10
E3	3h 36min	12 / 12	0	6.03 / 10
Ø 3 Epochen	**3h 37min**	—	—	**6.11 / 10**

Tabelle 25: Kategoriescores `moe-m10-gremium-deep` – 3-Epochen-Durchschnitt.

Kategorie	Ø Score	Bemerkung
Domain Routing	7.80 / 10	routing-code 9.4→9.2, routing-medical stabil
Precision (MCP)	7.95 / 10	precision-math 10.0→8.0, subnet stabil
Knowledge Healing	5.50 / 10	healing-novel +1.5 Pkt. über Epoche 1–3
Multi-Expert	5.20 / 10	synthesis-cross 4.8→5.4 (steigend)
Causal Reasoning	4.50 / 10	causal-surgery 3.6→4.2
Context/Memory	4.20 / 10	memory-8turn strukturelles Problem (s.u.)

Tabelle 26: Nativ vs. Orchestriert – M10-Hardware, MoE-Eval-Scores.

Modus	Template	Score	Anmerkung
Nativ (pro GPU)	moe-benchmark-n06-m10	3.3 / 10	1× 7–8B, kein Routing
Nativ (pro GPU)	moe-benchmark-n11-m10	3.6 / 10	1× 7–8B, kein Routing
Orchestriert	**moe-m10-gremium-deep**	**6.11 / 10**	**8 Domänenspezialisten + phi4:14b**
Orchestriert	moe-reference-30b	7.60 / 10	phi4:14b + 30B-Judge, RTX
Orchestriert	moe-aihub-sovereign	9.00 / 10	120B+122B, H200 (Cloud)

Tabelle 27: MoE-Eval-Score vs. vergleichbare Modelle (7–14B-Klasse).

System	Typ	Größe	MMLU	MT-Bench	Datensouveränität
GPT-4o mini (API)	Cloud	∼8B	82 %	8.8	×
Claude Haiku 3.5 (API)	Cloud	∼8B	∼80 %	∼8.5	×
Llama 3.1 8B (einzeln)	Lokal	8B	73 %	8.2	✓
Qwen2.5 7B (einzeln)	Lokal	7B	74 %	8.4	✓
phi4:14b (einzeln)	Lokal	14B	84 %	9.1	✓
moe-m10-gremium-deep	**Lok. Ensemble 8×7–9B**		–	–	✓
	MoE-Eval: 6.11 / 10 (gemessen)				

Tabelle 28: Strukturelle Ablation: Komponentenbeitrag aus gezielter Benchmark-Evidenz.

Komponente	Metrik	Δ (ohne)	Evidenz
MCP AST-Werkzeuge	Mathe-Genauigkeit	≈ −40 Pp	10/10 mit AST; ∼60 % LLM-Halluzinationsrate ohne deterministischen Aufruf [46]
Tier-2 Semantic Memory	Recall bei Tiefe > 50	≈ −100 %	MRCR-lite v2: Score 1,000 mit T2; Kollaps auf 0 ohne T2
L2/L3 Cache-Hierarchie	Medianlatenz (Epoche 5)	+830 %	Akkumulierungs-Benchmark: 707 s (kalt) vs. 76 s (warm)
Neo4j GraphRAG	Ø Score / Ø Latenz	−15 % Score	10-Anfragen-Direktvergleich (2026-07-11): GraphRAG 1,30 vs. Zero-Shot 1,10 (+18 %); GraphRAG gewinnt 2/3 auswertbare Paare; Latenz −3 %; 5/10 Timeouts wegen Host-Ressourcenmangels (siehe § 14.22)
Zwei-Tier-Eskalation	Kosten & Latenz	+30–50 %	T2-Fallback-Rate ≈18 %; ohne Routing alle Anfragen auf T2
Corrective RAG Gate	Kontextpräzision	Noch nicht gemessen	v2.5-Feature; qualitative Verbesserung bestätigt
Episodisches Gedächtnis	Routing-Konvergenz	Noch nicht gemessen	v2.5-Feature; Longitudinaldaten erforderlich

Tabelle 29: GraphRAG vs. Zero-Shot: zusammengefasste Benchmark-Ergebnisse (10 Anfragen, 2026-07-11).

Metrik	Zero-Shot	GraphRAG	Delta
Ø Score (0–5)	1,10	1,30	+0,20 (+18,2 %)
Ø Latenz	161,27 s	155,84 s	−5,43 s (−3,4 %)
Siege (höherer Score)	0	2	GraphRAG +2
Timeouts (>300 s)	5	5	—

Tabelle 30: ISO/IEC 27001:2022 Kontroll-Mapping und Plattform-Bereitschaft.

Annex A Control	Status	Technischer Mechanismus	Offene Betreiber-Aufgaben
A.5.9 / A.5.12 Asset- & Daten-Klassifizierung	Vorbereitet	Metadaten-Registry im *Data Catalog* mit automatischer DSGVO-Klassifizierung und Aufbewahrungs-Tags.	Definition der organisatorischen Klassifizierungs-richtlinie.
A.8.2 / A.8.3 Zugriffsteuerung & Privilegien	Operativ	Attributbasierte Zugriffskontrolle (ABAC), erzwungen durch Rego-Richtlinien des *Open Policy Agent (OPA)*.	Integration des Identity-Providers (LDAP/OIDC).
A.8.12 Schutz vor Datenabfluss (DLP)	Operativ	Local-First- und Air-Gap-Deployment; Outbound-Filter in `federation/` blockieren kritische Domänen.	Netzwerk-Perimeter-Schutz und DLP-Protokolle.
A.8.16 Protokollierung & Überwachung	Operativ	Auditierbare OpenLineage-Metadatenströme in *Marquez*; Prometheus-Host- und Node-Metriken.	SIEM-Integration und Eskalations-regeln.
A.8.20 / A.8.22 Netzwerksicherheit & Segregation	Vorbereitet	Isolierte Bridge-Netzwerke in Docker; containerisierte Modell-Runtimes (LXC/Podman).	Firewall-Konfiguration und VLAN-Routing.
A.8.25 / A.8.28 Sichere Entwicklung & Änderung	Operativ	Git-artige Branching- und Freigabe-Workflows für Wissens-Bundles via *lakeFS*.	Unabhängige Code-Reviews und S-SDLC-Prozesse.

Tabelle 31: EuroHPC-LUMI-G-Distillation-Ziele: Komponente, Zielmodell und Latenz-/Qualitätsziel.

Komponente	**Zielmodell**	**Ziel**
`planner_node`	Qwen3.5-4B (GGUF Q4_K_M / Q5_K_M)	≥90 % der GAIA-Planqualität des 35B-Lehrermodells bei ∼1/10 der Kosten, 200–400 ms CPU
`complexity_estimator`	DeBERTa-v3-small (ONNX INT8)	3–6 ms Inferenz
`semantic router`	multilingualer MiniLM-L12-v2 + Triplet Loss + FAISS	8–12 ms Inferenz
`RL-Routing-Policy`	3-Layer-MLP (Offline-RL)	<1 ms Inferenz
`node ranker`	XGBoost (ONNX)	0,3 ms Inferenz

Tabelle 32: vLLM-Performance: Qwen2.5-32B-Instruct auf 2× AMD MI250X GCD (TP=2, bfloat16).

Phase / Metrik	**Wert**	**Anmerkung**
Gewichte laden (Lustre)	146,89 s	61,03 GiB Safetensors von Scratch-FS
Torch-Kompilierung (AOT)	17,95 s	Dynamo-Bytecode-Cache
CUDA-Graph-Capture (51 Größen)	50,00 s	0,83 GiB je Rank
Gesamtstartzeit	264,97 s	Bis API-Status 200
Prefill-Durchsatz	548,0 tok/s	Prompt-Verarbeitung
Decode-Durchsatz	17,1–29,1 tok/s	Ø 22,8 tok/s
Gewichte je GCD	30,77 GiB	HBM2e-Auslastung
KV-Cache je GCD	27,2 GiB	Kapazität: 222 768 Tokens

Literatur

[1] N. Shazeer u. a. „Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer“. In: *International Conference on Learning Representations (ICLR)* (2017). URL: https://arxiv.org/abs/1701.06538.

[2] LangChain AI. *LangGraph: Building Stateful, Multi-Actor Applications with LLMs*. https://github.com/langchain-ai/langgraph. Accessed 2026-04-09. 2024.

[3] Chroma, Inc. *ChromaDB: The AI-native Open-Source Embedding Database*. https://www.trychroma.com/. Accessed 2026-04-09. 2023.

[4] The Linux Foundation. *Valkey: An Open Source High-Performance Key-Value Store*. https://valkey.io/. Fork of Redis OSS 7.2.4 after the licence change; accessed 2026-04-09. 2024.

[5] Neo4j, Inc. *Neo4j Graph Database*. https://neo4j.com/. Community Edition, version 5; accessed 2026-04-09. 2024.

[6] Anthropic. *Model Context Protocol Specification*. https://modelcontextprotocol.io/specification. Accessed 2026-04-09. 2024.

[7] European Parliament and Council. *Regulation (EU) 2016/679: General Data Protection Regulation (GDPR)*. https://eur-lex.europa.eu/eli/reg/2016/679/oj. Article 25 (Data protection by design and by default) and Article 32 (Security of processing). 2016.

[8] Ollama. *Ollama: Get Up and Running with Large Language Models Locally*. https://ollama.com/. Accessed 2026-04-09. 2024.

[9] Zylon AI. *PrivateGPT: Interact Privately with Your Documents Using the Power of LLMs*. https://github.com/zylon-ai/private-gpt. Accessed 2026-04-09. 2023.

[10] E. Di Giacomo. *LocalAI: Self-Hosted, Community-Driven, Drop-In Replacement for OpenAI*. https://github.com/mudler/LocalAI. Accessed 2026-04-09. 2023.

[11] Open Container Initiative. *OCI Image Format Specification, v1.1.0*. https://github.com/opencontainers/image-spec. Accessed 2026-04-09. 2024.

[12] Creative Commons. *Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)*. https://creativecommons.org/licenses/by-sa/4.0/. Licence under which this whitepaper is released. 2013.

[13] A. Q. Jiang u. a. „Mixtral of Experts“. In: *arXiv preprint*. 2024. eprint: 2401.04088. URL: https://arxiv.org/abs/2401.04088.

[14] D. J. Russo u. a. „A Tutorial on Thompson Sampling“. In: *Foundations and Trends in Machine Learning* 11.1 (2018), S. 1–96. DOI: 10.1561/2200000070.

[15] G. Mialon u. a. *GAIA: A Benchmark for General AI Assistants*. 2023. eprint: 2311.12983. URL: https://arxiv.org/abs/2311.12983.

[16] European Commission. *European Strategy for Data: Common European Data Spaces*. https://digital-strategy.ec.europa.eu/en/policies/strategy-data. Accessed 2026-04-09. 2020.

[17] Gaia-X European Association for Data and Cloud AISBL. *Gaia-X: A Federated and Secure Data Infrastructure*. https://gaia-x.eu/. Accessed 2026-04-09. 2022.

[18] S.-Q. Yan, J.-C. Gu, Y. Zhu und Z.-H. Ling. *Corrective Retrieval Augmented Generation*. Introduces a lightweight retrieval evaluator that scores retrieved documents for relevance before injection into the generation prompt and triggers web-search fallback for low-confidence retrievals. Forms the scientific basis for the Corrective RAG Gate in `graph_rag/manager.py`. 2024. arXiv: 2401.15884 [cs.CL]. URL: https://arxiv.org/abs/2401.15884.

[19] B. J. Chan, C.-T. Chen, J.-H. Lai und D.-B. Wu. *Don't Do RAG: When Cache-Augmented Generation is All You Need for Knowledge Tasks*. Demonstrates empirically that for stable, bounded knowledge domains, pre-loading authoritative context outperforms retrieval in accuracy, consistency, and latency. Forms the scientific basis for the CAG Compliance Layer in `compliance_cag.py`. 2024. arXiv: 2412.15605 [cs.CL]. URL: https://arxiv.org/abs/2412.15605.

[20] J. S. Park u. a. *Generative Agents: Interactive Simulacra of Human Behavior*. Introduces memory streams for LLM-based agents: experiences are stored in a structured log, retrieved by relevance and recency, and summarised into higher-level reflections. Forms part of the scientific basis for the episodic memory layer in `episodic_memory.py`. 2023. arXiv: 2304.03442 [cs.HC]. URL: https://arxiv.org/abs/2304.03442.

[21] C. Packer u. a. *MemGPT: Towards LLMs as Operating Systems*. Presents a tiered memory architecture for LLMs analogous to OS virtual memory: main context (fast, limited) backed by external storage (slow, unlimited). The three-tier Hot/Warm/ Cold memory design in MoE Sovereign follows this model. 2023. arXiv: 2310.08560 [cs.AI]. URL: https://arxiv.org/abs/2310.08560.

[22] W. Kwon u. a. *vLLM: Easy, Fast, and Cheap LLM Serving with PagedAttention*. 2023. eprint: 2309.06180. URL: https://arxiv.org/abs/2309.06180.

[23] H. Chase. *LangChain*. https://github.com/langchain-ai/langchain. Accessed 2026-04-09. 2022.

[24] W. Fedus, B. Zoph und N. Shazeer. „Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity". In: *Journal of Machine Learning Research* 23.120 (2022), S. 1–39. URL: http://jmlr.org/papers/v23/21-0998.html.

[25] D. Edge u. a. *From Local to Global: A Graph RAG Approach to Query-Focused Summarization*. Microsoft Research. 2024. eprint: 2404.16130. URL: https://arxiv.org/abs/2404.16130.

[26] Apache Software Foundation. *KRaft: Apache Kafka without ZooKeeper*. https://kafka.apache.org/documentation/#kraft. Production-ready since Kafka 3.3; accessed 2026-04-09. 2022.

[27] P. Lewis u. a. „Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2020. URL: https://arxiv.org/abs/2005.11401.

[28] E. Tulving. „Episodic and Semantic Memory". In: *Organization of Memory*. Hrsg. von E. Tulving und W. Donaldson. Foundational taxonomy distinguishing episodic memory (past experiences with temporal context) from semantic memory (factual world knowledge without personal context). Motivates the three-tier memory architecture in MoE Sovereign. Academic Press, 1972, S. 381–403.

[29] A. de Vries. „Algebraic hierarchy of logics unifying fuzzy logic and quantum logic“. In: *arXiv preprint* (2007). arXiv:0707.2161 [math.LO]. Establishes a unified algebraic lattice hierarchy that places paraconsistent, fuzzy, quantum, intuitionistic, and Boolean logics in a common order structure. Motivates the contradiction-tolerant reasoning strategy in `resolve_conflicts_node`. The concrete $\{T, F, B, N\}$ four-valued semantics is due to Belnap (1977).

[30] N. D. Belnap. „A Useful Four-Valued Logic“. In: *Modern Uses of Multiple-Valued Logic*. Hrsg. von J. M. Dunn und G. Epstein. Introduces the four-valued semantics $\{T, F, B, N\}$ (*True*, *False*, *Both*, *Neither*) for paraconsistent logic: contradictions are localised as B without triggering *ex contradictione quodlibet*. Applied in `resolve_conflicts_node` as the formal basis for conflict classification; four-valued truth labels (T, F, I, U) appear in the paraconsistent training dataset compiled on LUMI-G. Dordrecht: D. Reidel, 1977, S. 5–37.

[31] K. Gödel. „Zum intuitionistischen Aussagenkalkül“. In: *Anzeiger der Akademie der Wissenschaften in Wien* 69 (1932). Introduces the Gödel t-norm $T_G(a, b) = \min(a, b)$, the most conservative conjunction in fuzzy logic. Used in `fuzzy_router_node` as the default routing conjunction., S. 65–66.

[32] J. Łukasiewicz. „O logice trójwartościowej“. In: *Ruch Filozoficzny* 5 (1920). Introduces three-valued logic and the Łukasiewicz t-norm $T_{\text{Ł}}(a, b) = \max(0, a+b-1)$. Implemented alongside the Gödel t-norm in `pipeline/logic_types.py` as a selectable routing conjunction., S. 170–171.

[33] A. N. Kolmogorov. „Three Approaches to the Quantitative Definition of Information“. In: *Problems of Information Transmission* 1.1 (1965). Defines algorithmic information content (Kolmogorov complexity) as the length of the shortest description of a string. Applied in `complexity_estimator.py` via zlib compressibility as a practical upper bound., S. 1–7.

[34] G. J. Chaitin. „On the Length of Programs for Computing Finite Binary Sequences“. In: *Journal of the ACM* 13.4 (1966). Establishes that Kolmogorov complexity is computable via compression length as an upper bound. Co-basis with Kolmogorov 1965 for the zlib-based complexity estimator., S. 547–569.

[35] J. W. Ratcliff und D. E. Metzener. „Pattern Matching: The Gestalt Approach“. In: *Dr. Dobb's Journal* 13.7 (1988). Describes the Ratcliff/Obershelp string-similarity algorithm (longest common substring, iterative). Implemented in `graph_rag/manager.py` as `_fuzzy_resolve_entity_name` via Python's `SequenceMatcher`., S. 46–72.

[36] Y. Bai u. a. *Constitutional AI: Harmlessness from AI Feedback*. 2022. eprint: 2212.08073. URL: https://arxiv.org/abs/2212.08073.

[37] A. Madaan u. a. *Self-Refine: Iterative Refinement with Self-Feedback*. 2023. eprint: 2303.17651. URL: https://arxiv.org/abs/2303.17651.

[38] P. Horn. *Fine-Tuning Large Language Models on AMD MI250X: Practical Lessons from LUMI-G Sovereign AI Training*. Technical Report, EuroHPC Grant EHPC-DEV-2026D06, August 2026. Companion paper documenting the Judge and Planner fine-tuning campaigns on LUMI-G; covers ZeRO-2/3 memory analysis, OOM root causes, SLURM dependency chains, and the teacher-student distillation pipeline. 2026.

[39] Containers. *Rootless Podman: Running Containers as a Non-Root User*. https://github.com/containers/podman/blob/main/docs/tutorials/rootless_tutorial.md. Accessed 2026-04-09. 2024.

[40] Broadcom. *Bitnami Helm Charts*. https://github.com/bitnami/charts. Accessed 2026-04-09. 2024.

[41] Red Hat. *OpenShift Security Context Constraints (SCC)*. https://docs.openshift.com/container-platform/latest/authentication/managing-security-context-constraints.html. Accessed 2026-04-09. 2024.

[42] W3C. *Trace Context, Level 1 (W3C Recommendation)*. https://www.w3.org/TR/trace-context/. Defines the `traceparent` HTTP header for distributed trace propagation. 2021.

[43] Grafana Labs. *Grafana Alloy: A Flexible Distribution of OpenTelemetry Collector*. https://grafana.com/docs/alloy/. Successor to Grafana Agent; accessed 2026-04-09. 2024.

[44] Prometheus Authors. *Prometheus: Monitoring System and Time Series Database*. https://prometheus.io/. CNCF graduated project; accessed 2026-04-09. 2024.

[45] V. Krakovna u. a. *Specification Gaming: the Flip Side of AI Ingenuity*. DeepMind Blog, 6 April 2020. Canonical survey of specification-gaming examples across reinforcement learning and optimisation systems. 2020. URL: https://deepmind.google/discover/blog/specification-gaming-the-flip-side-of-ai-ingenuity/.

[46] T. Schick u. a. „Toolformer: Language Models Can Teach Themselves to Use Tools“. In: *Advances in Neural Information Processing Systems* 36 (2024). arXiv:2302.04761.

[47] E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media, 1999. ISBN: 978-0596001087.

Anhang

A Glossar

Persistentes Graph-State-Tracking
: Der Prozess, durch den der Wissensgraph im laufenden Betrieb mit neuen Entitäten und Relationen angereichert wird, während jede Änderung versioniert und inspizierbar bleibt.

Deterministisches Routing
: Eine Routing-Entscheidung, die bei gleicher Eingabe und gleicher Konfiguration reproduzierbar zum gleichen Ergebnis führt – im Gegensatz zu gelerntem, probabilistischem Routing.

Expert-Template
: Eine versionierte JSON-Konfiguration, die pro Expertenkategorie eine Liste von Modellen mit Rollen-Tags (`primary`, `fallback`, `always`) und einem System-Prompt definiert.

Judge-Node
: Der letzte Knoten der LangGraph-Pipeline, der die gesammelten Expertenantworten bewertet und einen Self-Evaluation-Score vergibt.

LangGraph
: Ein Open-Source-Framework von LangChain AI für stateful Multi-Actor-Workflows auf Basis von LLMs [2].

Merger-Node Der Knoten in der LangGraph-Pipeline, der Antworten mehrerer Experten zusammenführt. Er ist auch der Emitter von `<SYNTHESIS_INSIGHT>`-Blöcken für den GraphRAG-Ingest.

MCP *Model Context Protocol*. Ein Protokoll von Anthropic für den Aufruf externer, prozessgetrennter Werkzeuge durch LLMs [6].

Quadlet Eine systemd-native Unit-Datei für die Verwaltung rootless Podman-Container ab Podman 4.4.

Rootless Podman Eine Container-Runtime, die ohne Root-Privilegien und ohne Daemon-Prozess arbeitet [39].

Traceparent Der HTTP-Header aus dem W3C-Trace-Context-Standard [42] zur kausalen Korrelation verteilter Anfragen.

Valkey Ein Fork der Redis-OSS-Version 7.2.4, der nach der Lizenzänderung von Redis durch die Linux Foundation hervorgebracht wurde [4].

B Expert-Katalog

Die aktuelle Installation kennt die folgenden 15 Expertenkategorien. Jede Kategorie ist durch eine Markdown-Datei im Repository `docs/experts/` dokumentiert und hat mindestens einen `primary`-Eintrag im Standard-Template.

Kategorie	Zweck
`general`	Universalassistenz für unspezifische Anfragen.
`math`	Mathematik, Gleichungen, Einheiten, Statistik.
`technical_support`	IT, DevOps, Debugging, Systemadministration.
`code_reviewer`	Code-Review mit Sicherheitsfokus.
`creative_writer`	Texterstellung, stilistische Überarbeitung.
`medical_consult`	Medizinische Information mit Disclaimer-Modus.
`legal_advisor`	Deutsches Recht (BGB, StGB u. a.) mit Paragraphenbezug.
`translation`	Multilinguale Übersetzung.
`reasoning`	Komplexe Logik, strategische Überlegungen.
`vision`	Bild- und Screenshot-Analyse.
`data_analyst`	Statistik, Pandas, explorative Analysen.
`science`	Chemie, Biologie, Physik.
`agentic_coder`	Agentic-Coding-Workflows auf ganzen Repositories.
`judge`	Antwort-Synthese und -Validierung.
`planer`	Mehrschrittige Planung und Dekomposition.

C Umgebungsvariablen-Referenz (Auszug)

Die folgende Auswahl zeigt die wichtigsten Konfigurations-Stellschrauben. Die vollständige Referenz liegt im Repository unter `docs/reference/`.

Variable	Bedeutung
`MOE_PROFILE`	`solo` / `team` / `enterprise`.
`KAFKA_URL`	Bootstrap-Adresse des Kafka-Clusters.
`REDIS_URL`	Valkey-Verbindungs-URL.
`POSTGRES_CHECKPOINT_URL`	LangGraph-Checkpoint-DB-URL.
`MOE_USERDB_URL`	Admin-DB-URL (Users, Budgets, Audit).
`CHROMA_HOST`	ChromaDB-Endpunkt.
`NEO4J_URI`	Neo4j-Bolt-URL.
`MCP_URL`	MCP-Server-URL.
`INFERENCE_SERVERS`	JSON-Liste der Inference-Backends.
`EXPERT_TEMPLATES`	JSON-Liste der Expert-Templates.
`MOE_LOGS_DIR`	Writable logs-Pfad im Container.
`MOE_CACHE_DIR`	Writable cache-Pfad im Container.
`MOE_EXPERTS_DIR`	Read-only Expert-Markdown-Verzeichnis.
`JWT_SECRET`	Schlüssel für Mandanten-Token.
`LOKI_URL`	Optional: Remote-Log-Senke.
`TEMPO_URL`	Optional: Remote-Trace-Senke.

D Lizenz und Third-Party-Notices

Die begleitende Software steht unter der **Apache License 2.0**. Dieses Whitepaper-Dokument steht unter *Creative Commons Attribution-ShareAlike 4.0 International* (CC BY-SA 4.0) [12]. Eine vollständige Liste der verwendeten Third-Party-Bibliotheken mit ihren jeweiligen Lizenzen liegt im Repository unter `THIRD_PARTY_NOTICES.md` vor. Die wichtigsten dort aufgeführten Komponenten sind: LangChain und LangGraph (MIT), FastAPI (MIT), Pydantic (MIT), Valkey (BSD-3-Clause), ChromaDB (Apache-2.0), Neo4j Community Edition (GPLv3), Apache Kafka (Apache-2.0), PostgreSQL (PostgreSQL License), Grafana OSS (AGPLv3), Prometheus (Apache-2.0), Caddy (Apache-2.0), Ollama (MIT), Authentik (MIT), mkdocs-material (MIT).

Hinweis zum Lizenzwechsel (v1.0 → v1.1): Die Software wurde initial unter CC BY-SA 4.0 veröffentlicht. Ab v1.1 steht sie unter Apache 2.0, das klarere Patent-Grant-Formulierungen enthält und eine breitere Kompatibilität mit kommerziellen Integrationen bietet, während der Open-Source-Charakter des Projekts erhalten bleibt.

E Kontakt

Herausgeber, Initiator und Ideenhalter:

Name	Philipp Horn
Anschrift	Benediktus-Kirchplatz 15, 59387 Ascheberg, Deutschland
E-Mail	kontakt@philipp-horn.dev
GitHub	https://github.com/h3rb3rn/moe-sovereign
Website	https://moe-sovereign.org
LinkedIn	https://www.linkedin.com/in/philipp-horn-dev/

Für die inhaltlichen Aussagen dieses Whitepapers übernimmt der Herausgeber die Verantwortung. Für die technischen Empfehlungen gilt der übliche *as is*-Disclaimer: jede Betreiberin

ist verpflichtet, die für die eigene Installation geltenden rechtlichen, regulatorischen und technischen Rahmenbedingungen eigenverantwortlich zu prüfen.