MoE Sovereign

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

Whitepaper — Version 1.0, April 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, einen vollständig selbstgehosteten 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) short-circuited redundante LLM-Aufrufe. 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 ist vollständig OpenAI-API-kompatibel, benötigt im Minimalbetrieb keine ausgehenden Internetverbindungen und ist durch seine Architektur mit Artikel 25 der DSGVO [7] konform. Das selbe 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 MoE-Eval erreicht – dieselbe Score-Klasse wie cloud-gehostetes GPT-4o mini – 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) 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. Das Projekt wird unter der Lizenz CC BY-SA 4.0 veröffentlicht und 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**: Das selbe 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. Das Projekt wird unter CC BY-SA 4.0 [12] veröffentlicht und bleibt es.

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, beantworten die Benchmark-Ergebnisse in Abschnitt 14. Die kurze Antwort lautet: *partiell ja*. Der Orchestrator erreicht auf GAIA Level 1 Scores von 50–60 % (bestes Ergebnis: 46,7 % gesamt, was den GPT-4o Mini-Referenzwert von 44,8 % übertrifft), bei vollständiger Verarbeitung auf Consumer- und Legacy-Hardware. Wo das System an strukturelle Grenzen stößt – 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. Der Score wird als Laplace-geglätteter Schätzer $(M + 1)/(N + 2)$ berechnet – eine Annäherung an die Bayes-optimale Vorhersage für 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.

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.

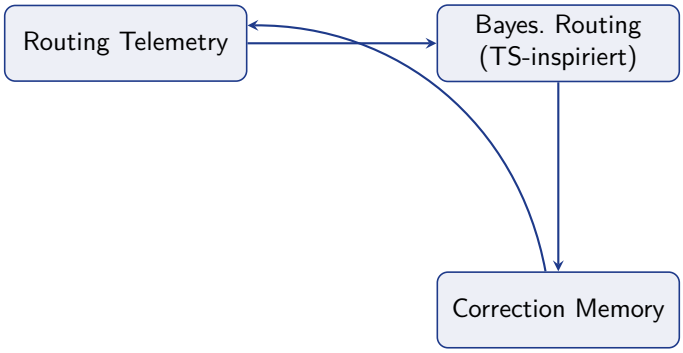

Abbildung 1: Das RL-Flywheel: Routing Telemetry, Thompson-Sampling-inspiriertes Score-Update und Correction Memory bilden einen sich selbst verstärkenden Verbesserungskreislauf ohne Modell-Finetuning.

Die empirische Evidenz für die Wirkung des Flywheels zeigt sich in der Messreihe aus Tabelle 10: 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.

Kernprinzip: Architektur erzeugt Intelligenz

Das RL-Flywheel verkörpert das Grundprinzip des Projekts: nicht *mehr Rechenleistung* ist die Variable, die Systemqualität maximiert, sondern *besser strukturierte Informationsverarbeitung*. Routing Telemetry, bayesianische Expertenselektion und persistiertes Korrekturwissen sind Softwareabstraktionen – sie laufen auf der bereits vorhandenen Legacy-Hardware ohne Änderungen.

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) addiert der Orchestrator einen messbaren Mehrwert: 46,7 % Gesamt und 60 % auf GAIA Level 1 mit dem 30b-Balanced-Template.

- 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.

4.4 Forschungsnahe Stacks

Die Kombination aus vLLM [18] für die Inference-Schicht und LangChain [19]/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. [20] 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. [21] 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 erreicht vergleichbare Architekturtiefe als Open-Source-Container auf lokaler Hardware.

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.

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

4.8 Vergleichende Feature-Matrix

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 [22] (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.

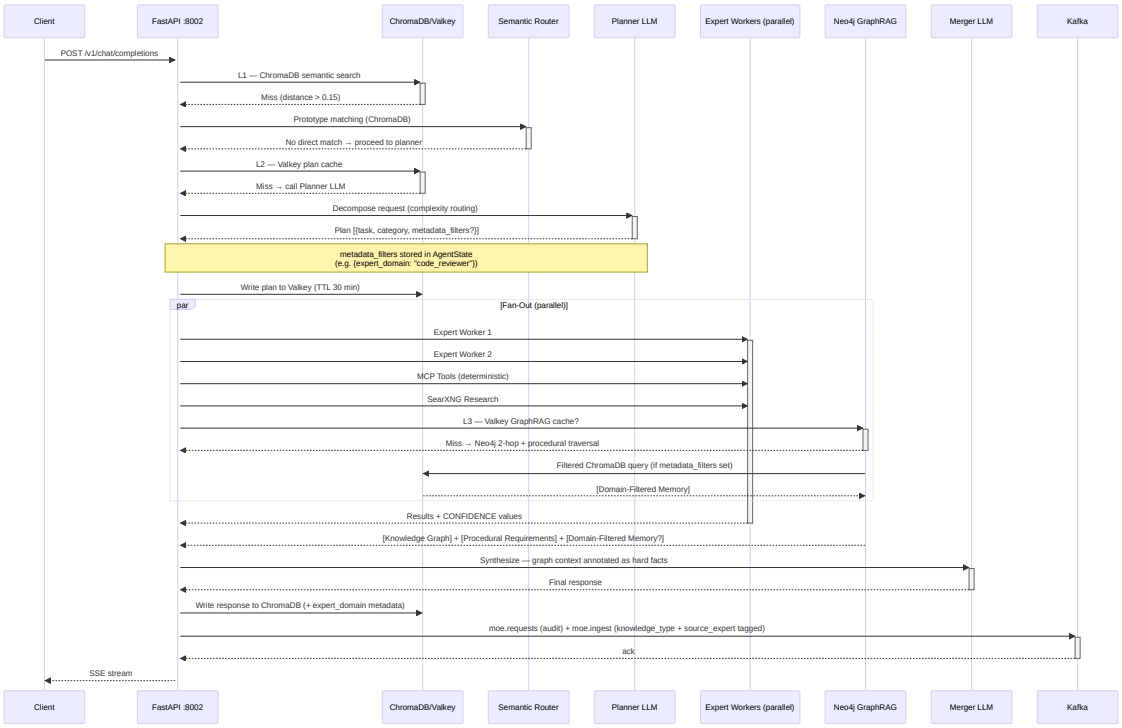

Abbildung 2: End-to-End-Datenfluss einer Chat-Anfrage durch den Orchestrator. Jeder Knoten in der Pipeline ist ein LangGraph-Node mit explizitem Zustand; jeder Kasten ausserhalb der Pipeline ist ein separater Service mit eigener API.

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 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 Kon-

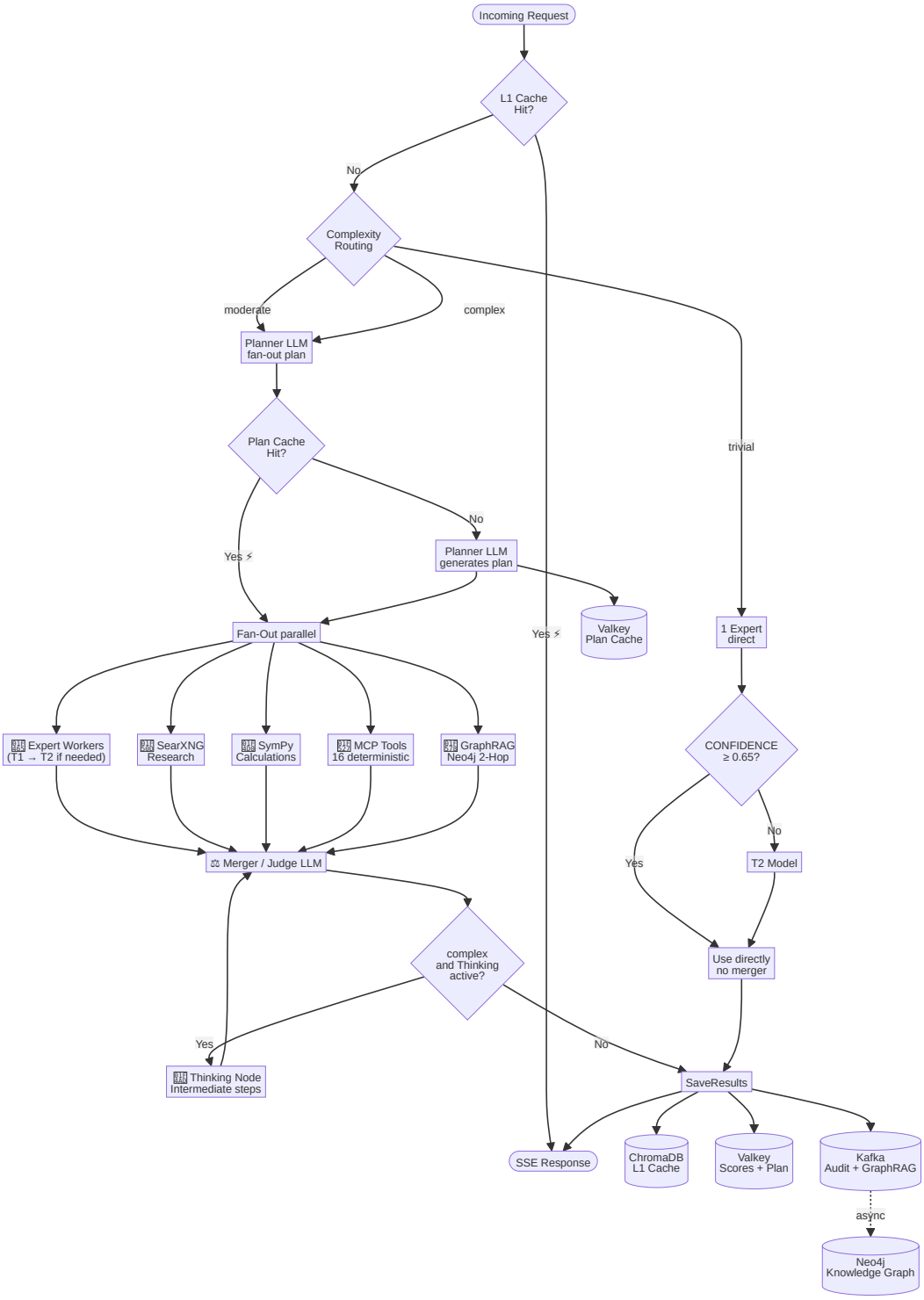

Abbildung 3: Die LangGraph-Pipeline. Anfragen durchlaufen `planner` (Expertenauswahl), `expert_worker` (parallele Modell-Aufrufe pro Kategorie), `merger` (Zusammenfassung mit Quellpriorität), optional `thinking_node` (4-stufige Chain-of-Thought bei Low-Confidence), optional `research_fallback` (externe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

figuration (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.

5.4 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.

5.5 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.

5.6 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). 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.

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).

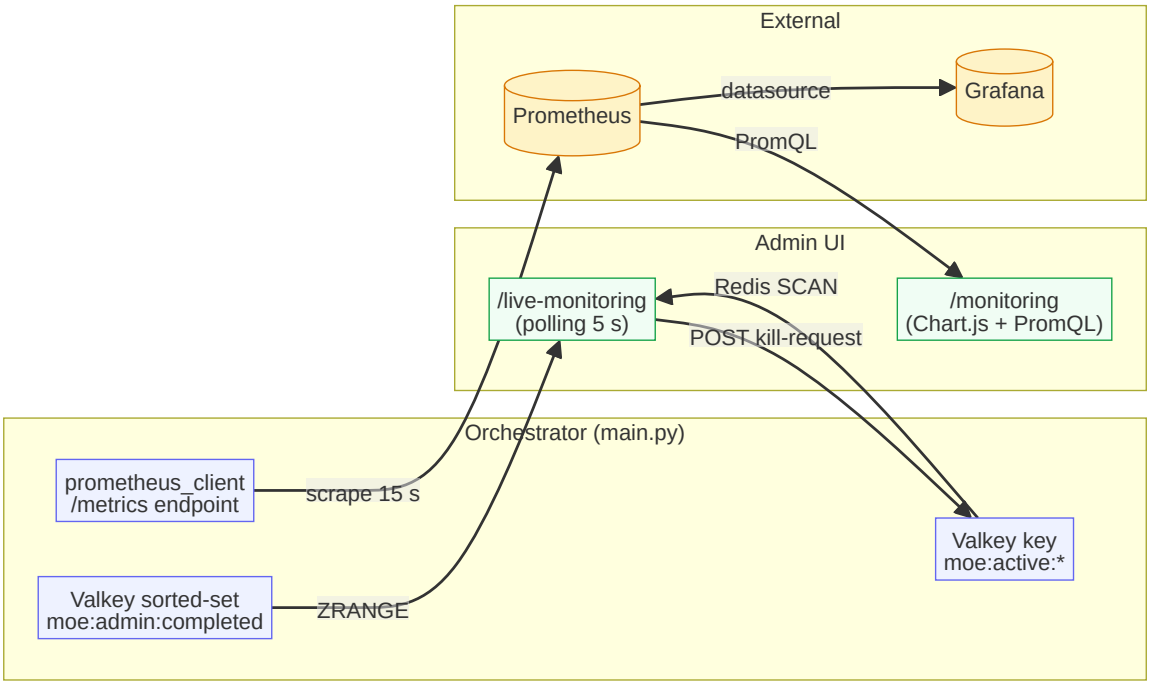

Abbildung 4: Die Observability-Architektur spiegelt denselben Design-Ansatz: Systemmetriken und Live-Monitoring werden als zwei komplementäre Sichten auf dieselbe Datenbasis behandelt – keine Abstraktion, die die Herkunft verschleiert.

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 (2× GT 1060)	12 GB	1.0	14B
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 – dieses Prinzip haben wir die Apollo-11-Methode genannt.

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.

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

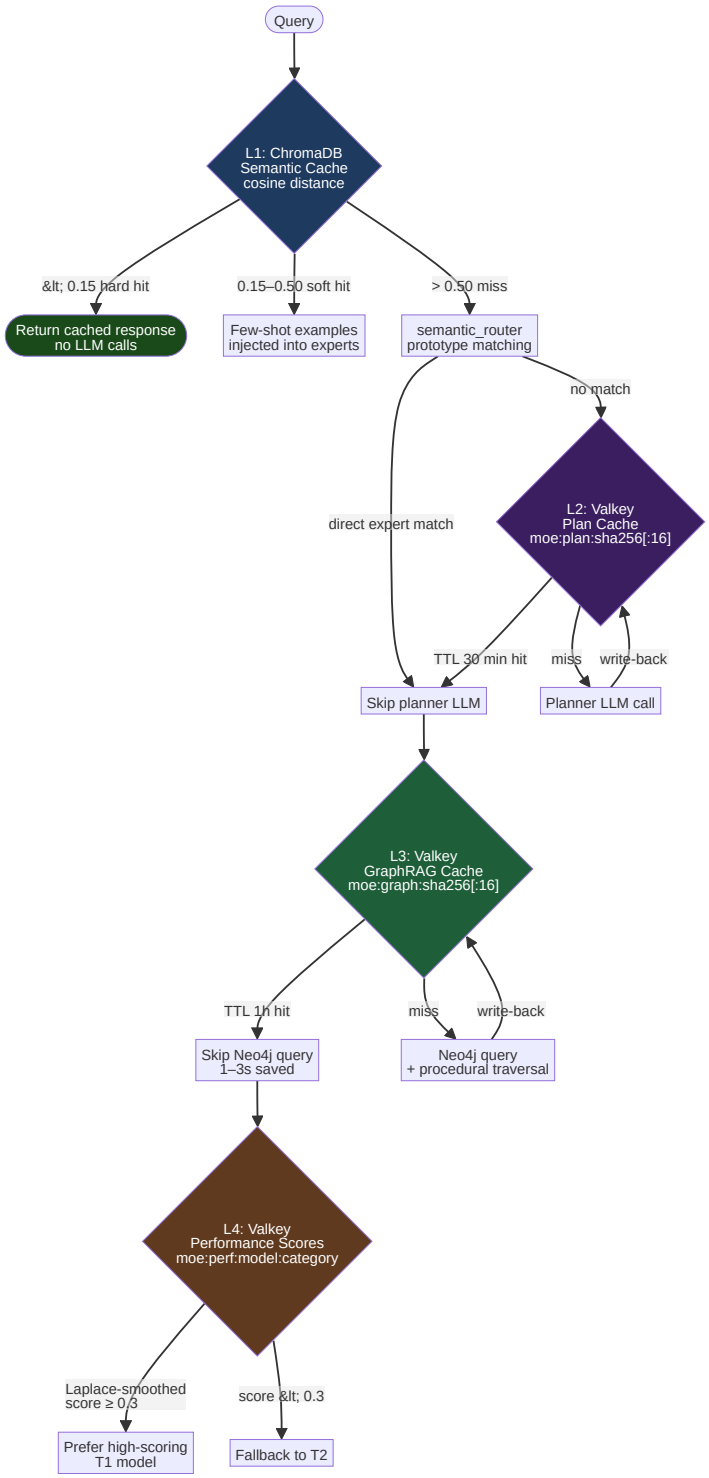

Abbildung 5: Die vierschichtige Cache-Hierarchie: L1 semantisch, L2 Plan, L3 Graph, L4 Performance-Score. Jede Schicht hat einen distinkten Key-Raum und eine eigene TTL.

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 (model, 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.

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 [23] 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 [21], 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.

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*[1] 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.

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.

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

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.

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__("os")`, 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.
`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.

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.

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.

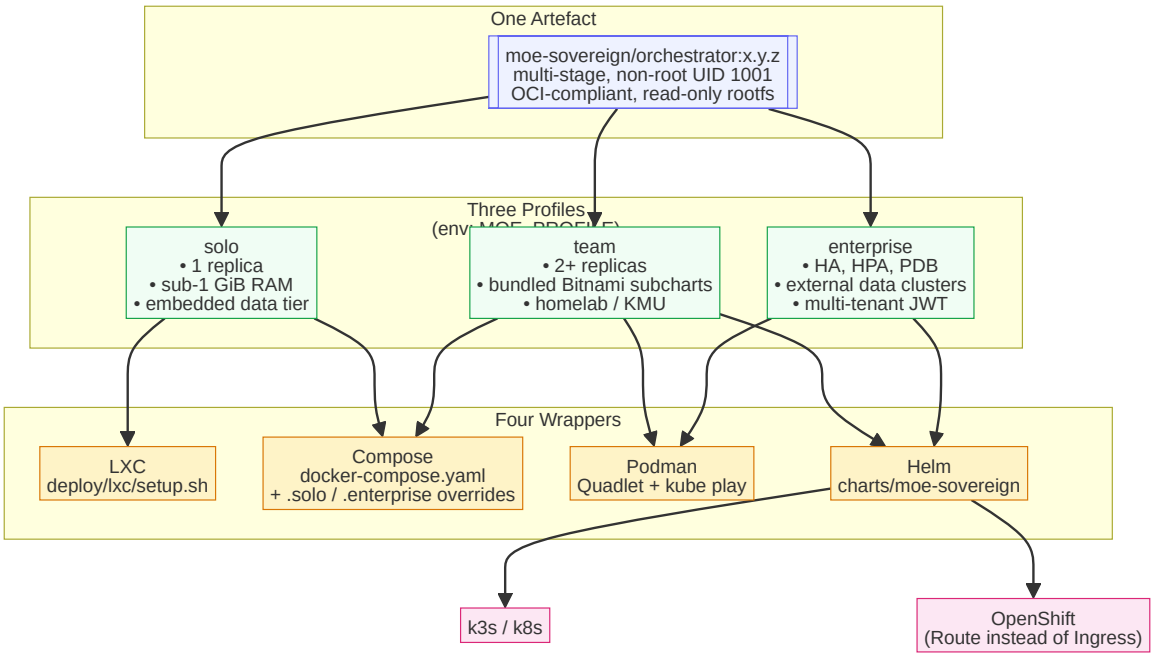

Abbildung 6: Das Universal-Deployment-Prinzip: ein einziges OCI-Artefakt (*single artifact*), drei Profile (`solo`, `team`, `enterprise`), vier Deployment-Wrapper (LXC-Script, Docker Compose, Podman Quadlet, Helm Chart), laufend auf fünf Ziel-Plattformen (LXC, Docker-Host, k3s, Kubernetes, OpenShift).

11.1 Single-Artifact-Prinzip

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. Das selbe 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.

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.

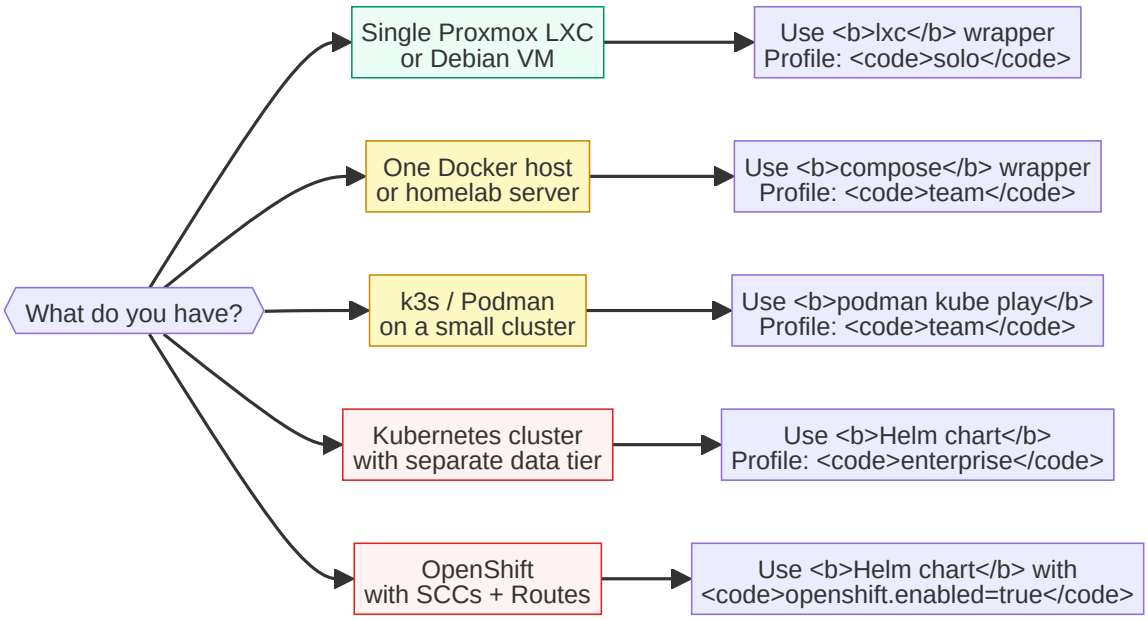

Abbildung 7: Die Tier-Entscheidungshilfe: welches Profil und welcher Wrapper für welches Deployment-Ziel empfohlen wird.

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 [24], 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 $\geq$ 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.

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 [25] 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 [26] 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.
- `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.

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.

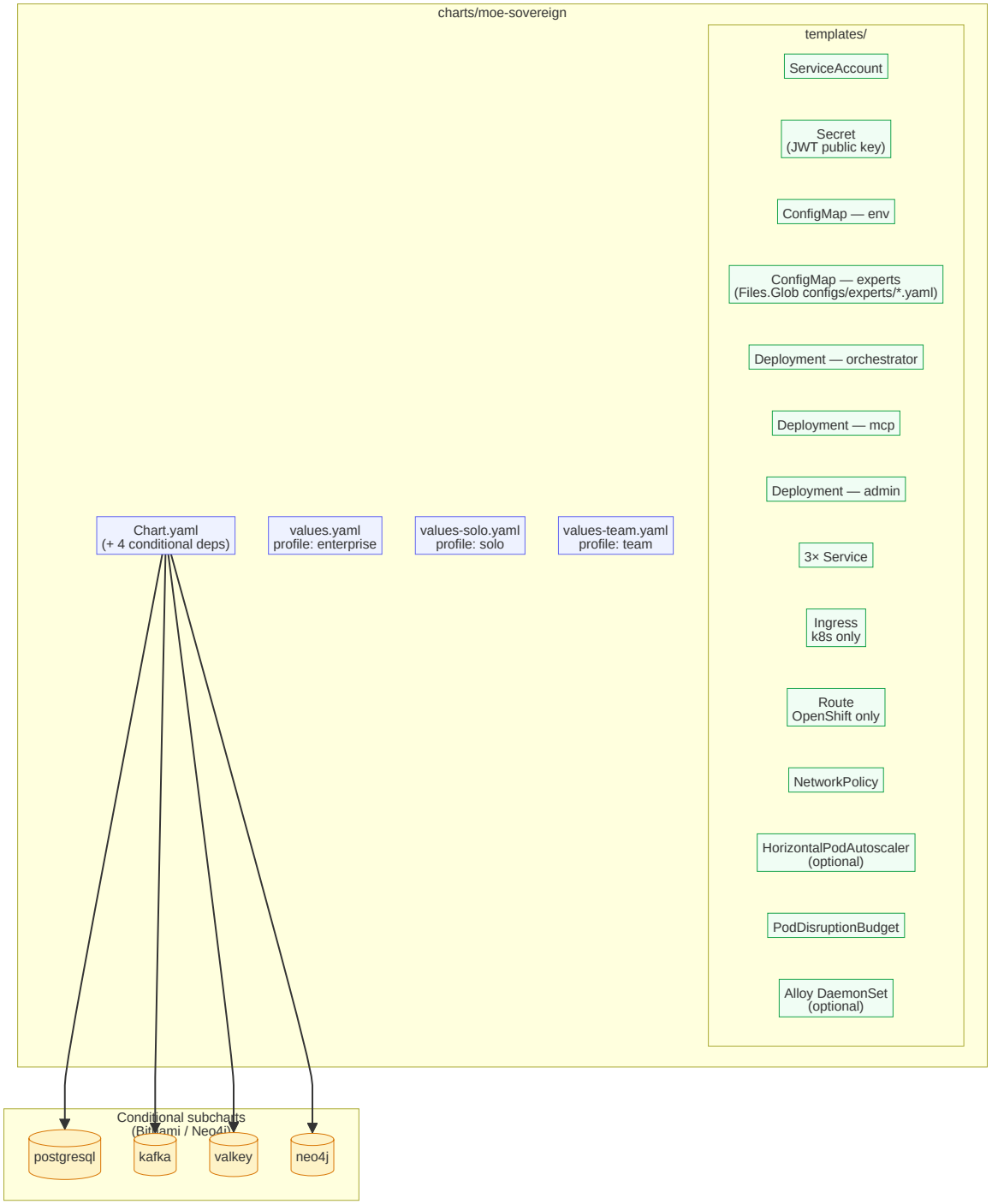

Abbildung 8: Die Helm-Chart-Struktur: `Chart.yaml` mit bedingten Bitnami-Subcharts, drei Values-Dateien für die drei Profile und 14 Template-Dateien inklusive OpenShift-Route-Switch.

Unsere Antwort ist das Universal-Deployment-Modell: ein Image, drei Profile, vier Wrapper, null Forks.

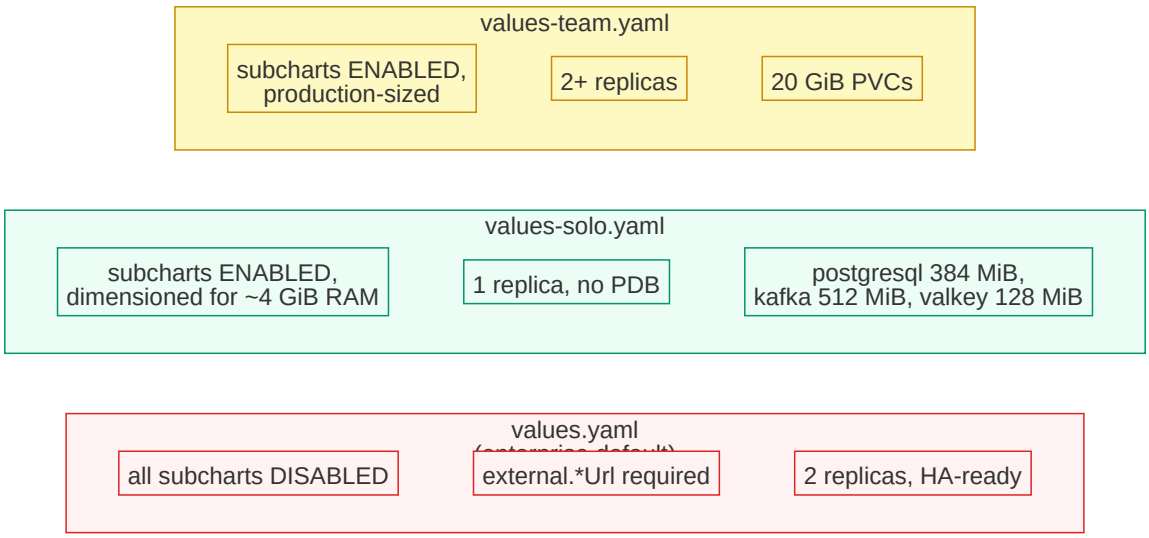

Abbildung 9: Vergleich der drei Profile: `solo`, `team`, `enterprise` im Helm-Values-Format.

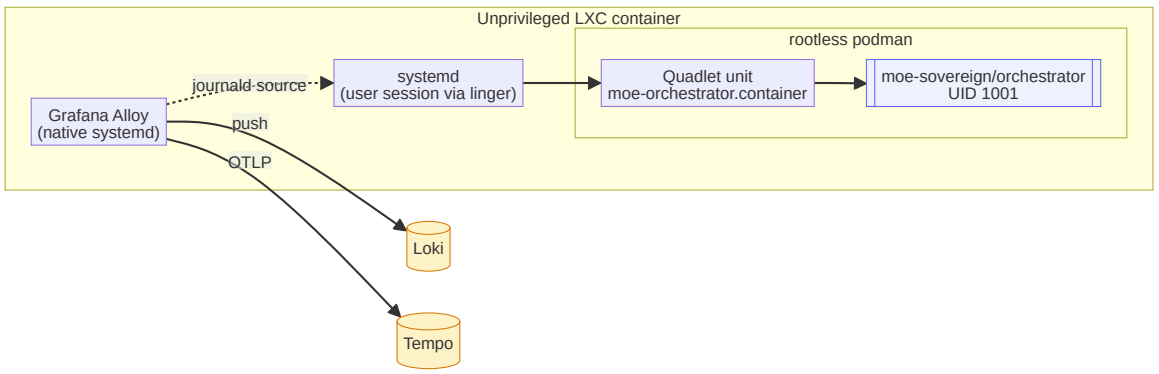

Abbildung 10: Rootless Podman in einem unprivilegierten LXC: der Service-Benutzer (UID 1001) startet den Container über eine Quadlet-Unit, systemd verwaltet den Lebenszyklus, Grafana Alloy liest die Logs direkt aus dem Journald-Cursor.

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* [27] und einem einheitlichen Alloy-Collector [28] lösen.

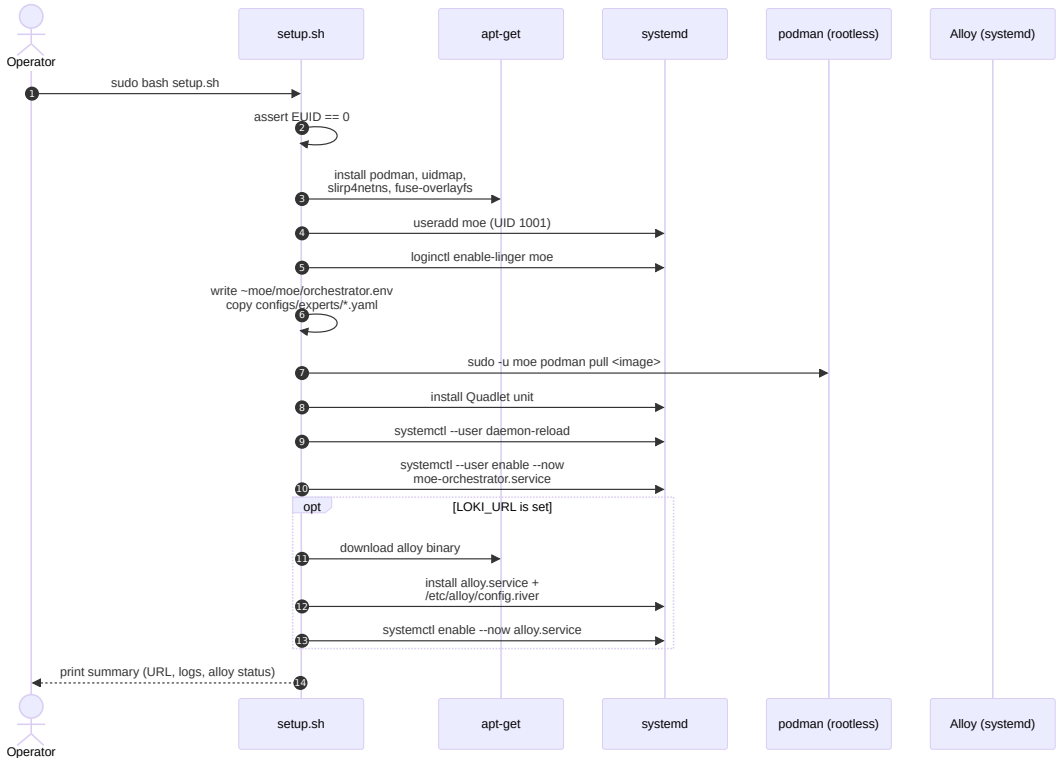

Abbildung 11: Die Sequenz des `deploy/lxc/setup.sh`-Scripts: Paketinstallation, Benutzeranlage, Linger aktivieren, Quadlet-Unit kopieren, Alloy einrichten. Gesamtdauer auf einem 2-vCPU-LXC: etwa eine Minute.

12.1 Prometheus-Metriken

Der Orchestrator exportiert auf `/metrics` einen Prometheus [29]-kompatiblen Endpunkt. Die wichtigsten Metriken sind in Tabelle 3 aufgeführt. Alle Metriken tragen konsistente Labels: `model`, `category`, `deployment_profile` und, wo zutreffend, `tenant_id`.

Tabelle 3: 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).
`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.

Metrik	Semantik
`moe_tenant_token_budget`	Verbleibendes Token-Budget pro Mandant.

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 [28]. 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 [27]) 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.

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.

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.

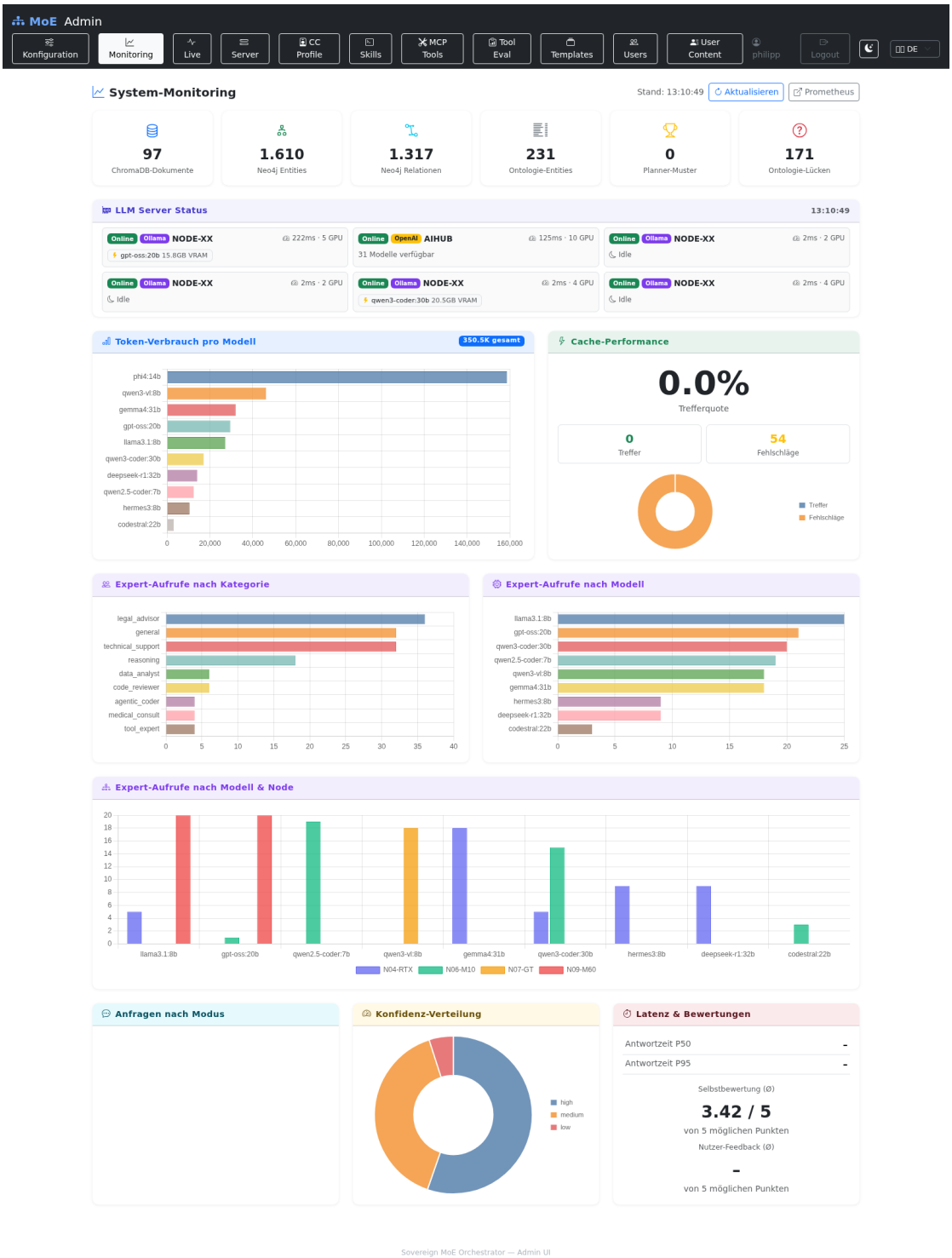

Abbildung 12: Das Admin-UI-Dashboard „System-Monitoring“. Alle sichtbaren Hostnamen sind in dieser Dokumentation durch `NODE-XX` ersetzt; im Betrieb zeigt das Dashboard die tatsächlichen Node-Namen der konfigurierten Inference-Server.

13.1 Mehrstufige Isolation im Container-Bau

Die erste Verteidigungslinie ist das Container-Image selbst:

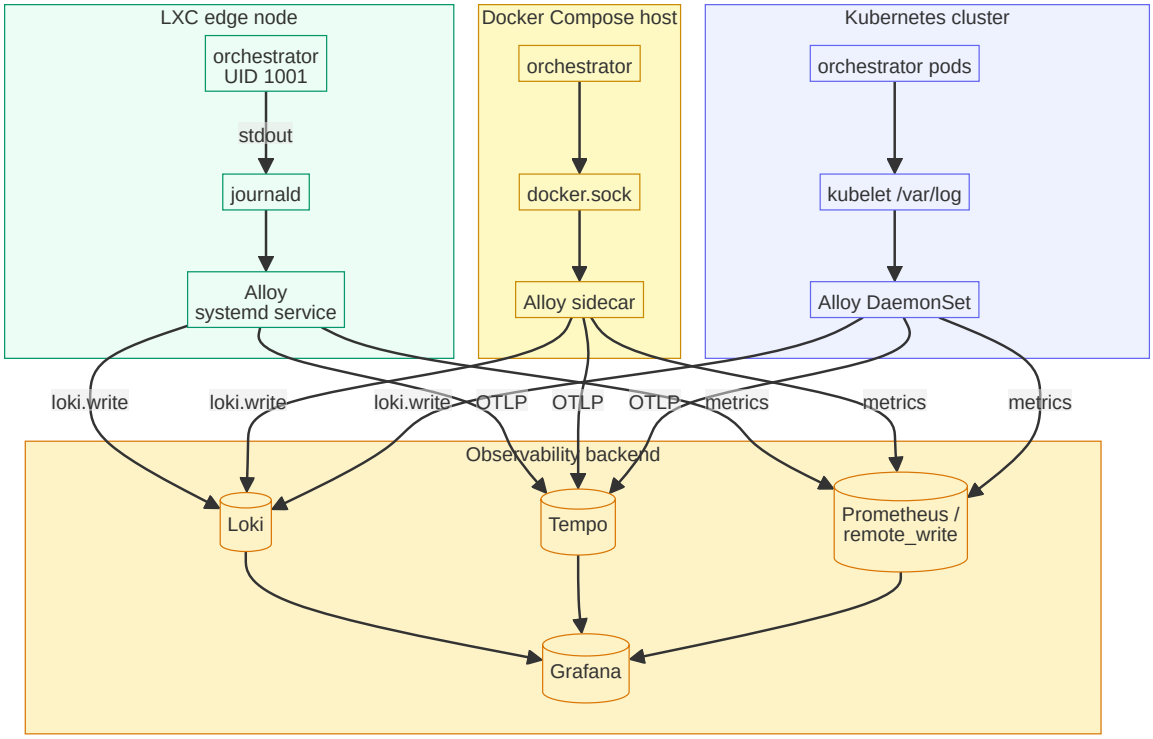

Abbildung 13: Die universelle Observability-Pipeline: Metriken fließen über Prometheus, Logs über Loki, Traces über Tempo. Alloy ist der einheitliche Collector auf allen drei Deployment-Zielen. Der `traceparent`-Header propagiert die Trace-ID durch die gesamte Kette.

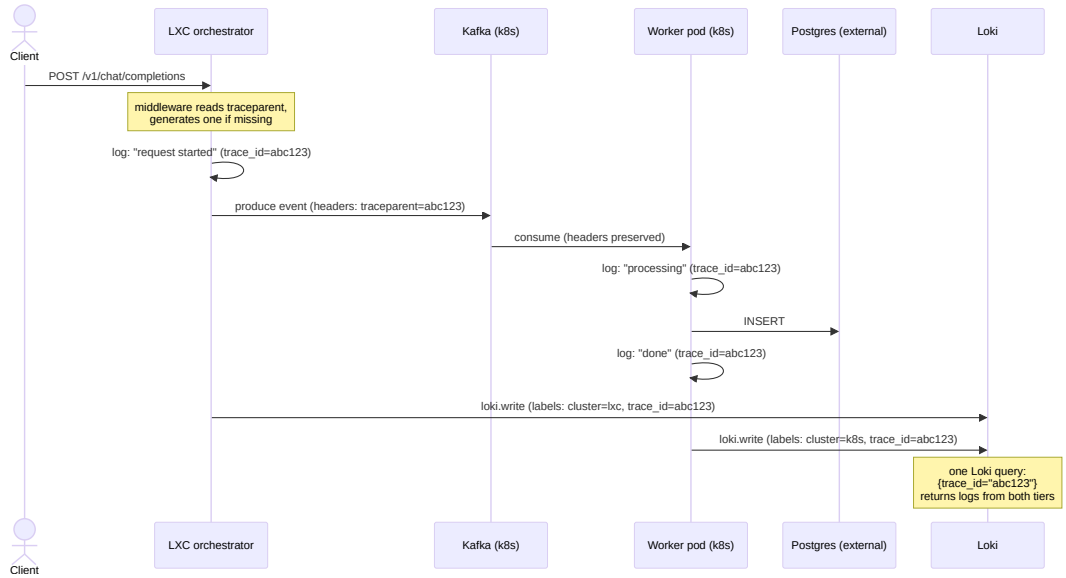

Abbildung 14: Die Propagation des `traceparent`-Headers durch die Deployment-Schichten. Eine Anfrage an den LXC-Edge-Node wird mit derselben Trace-ID durch Kafka und den k8s-Cluster weitergeleitet; Tempo kann die vollständige Kette darstellen.

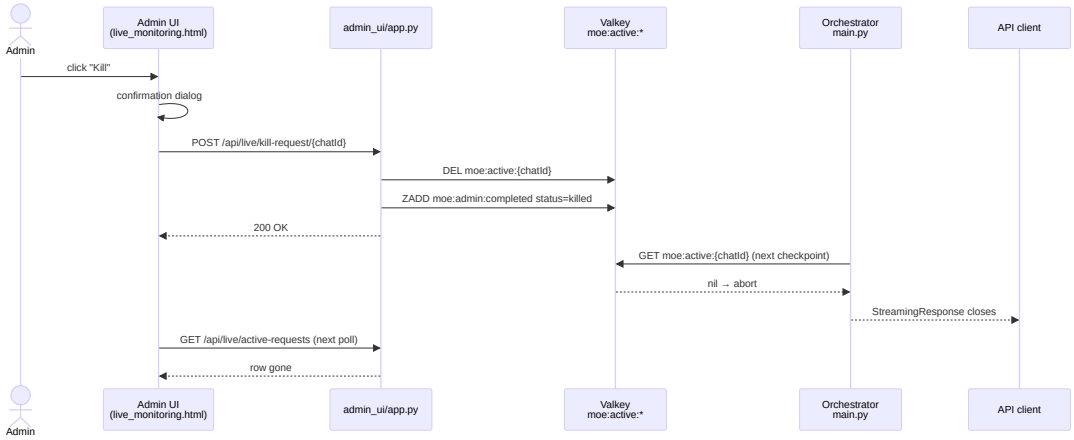

Abbildung 15: Der Kill-Flow eines aktiven Requests. Der Admin drückt den Kill-Button, das Admin-UI schreibt eine Kafka-Nachricht, der Orchestrator liest sie am nächsten Checkpoint, stoppt die LangGraph-Instanz, das Admin-UI aktualisiert die Anzeige. Latenz: unter eine Sekunde im Normalfall.

- **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.

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).

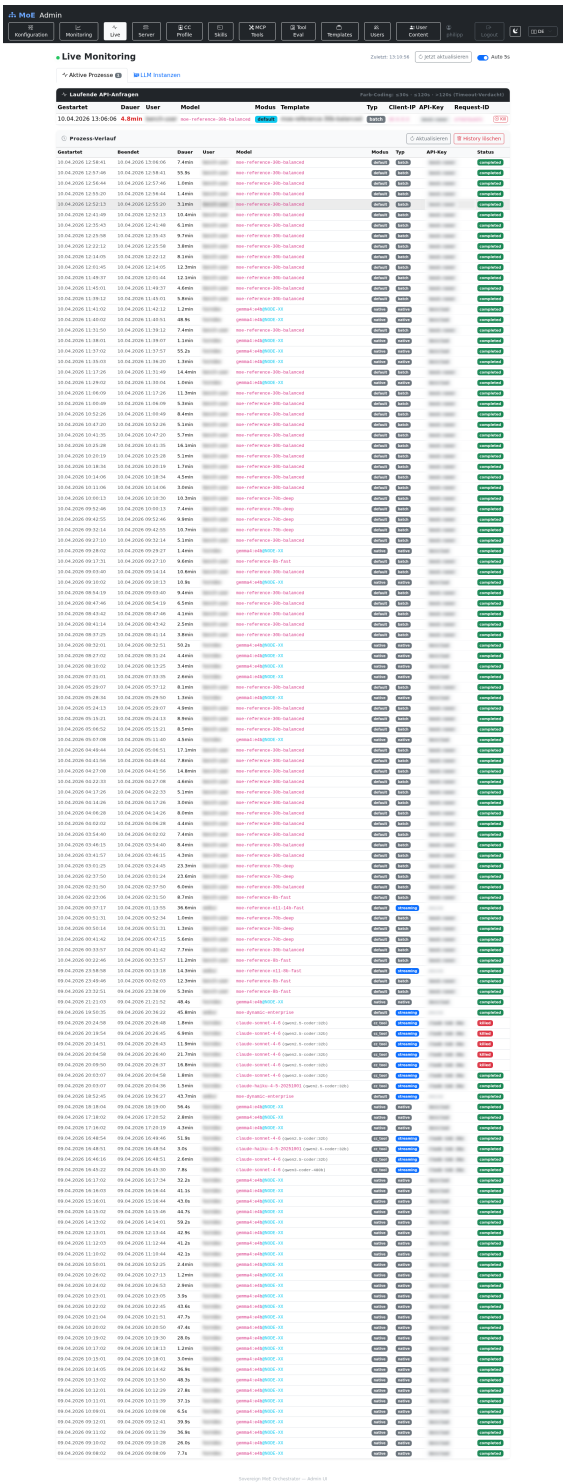

Abbildung 16: Das Live-Monitoring des Admin-UI mit sichtbarem aktiven Prozess (oben, Laufzeit 6.2 min) und historischer Prozessliste (unten). Alle identifizierenden Spalten (User, Client-IP, API-Key, Request-ID, Template) sind in dieser Dokumentation per Blur maskiert; in einer echten Installation sind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

Überschreitungen produzieren einen 429-Response, werden als Prometheus-Metrik gezählt

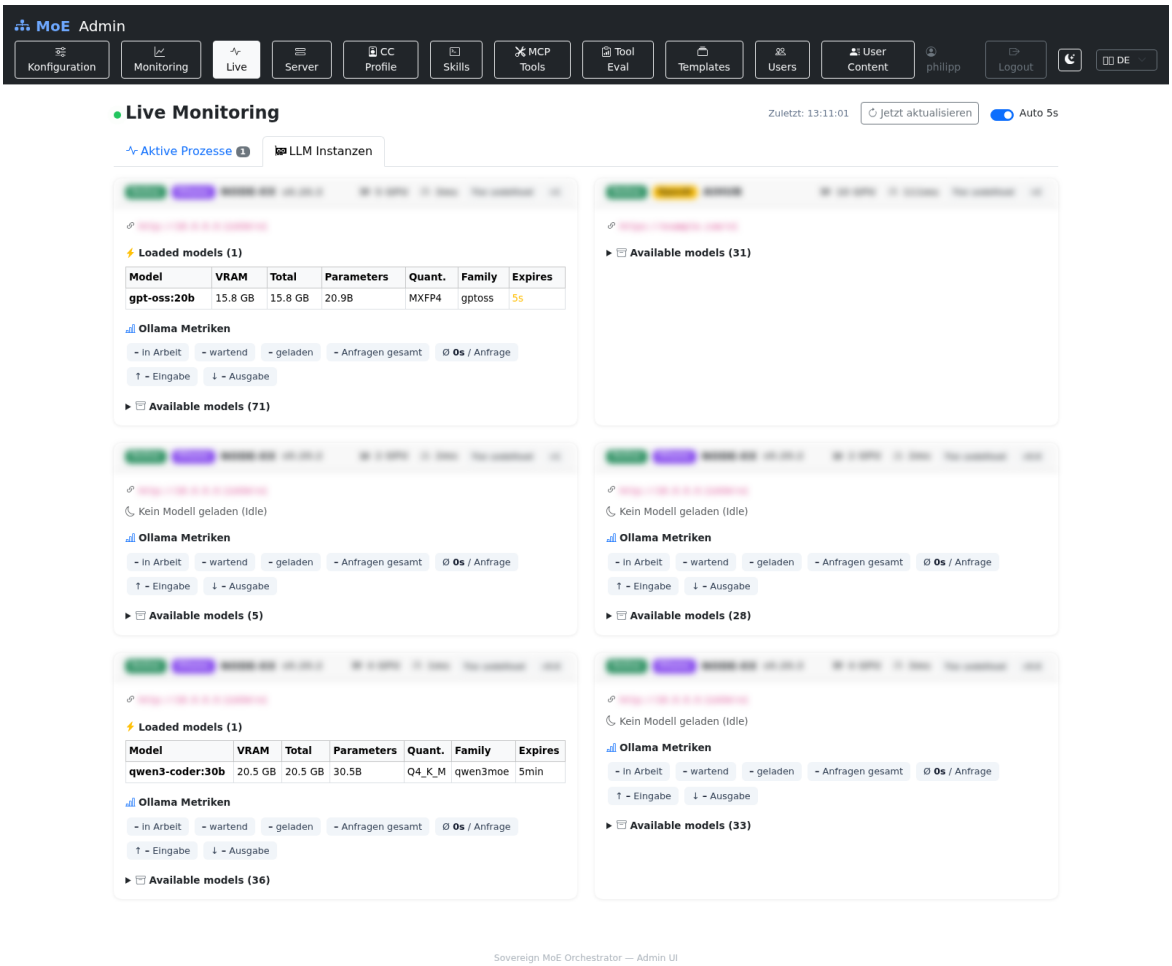

Abbildung 17: Die LLM-Instances-Ansicht im Admin-UI: pro konfiguriertem Inference-Server werden der Live-Status, die geladenen Modelle inklusive VRAM-Belegung und Quantisierung, die Ollama-Metriken (wartend, geladen, Durchsatz) und die Liste der verfügbaren Modelle angezeigt. Die Hostnamen-Header wurden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

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

Als operative Kurzfassung lässt sich die DSGVO-Art.-25-Konformität auf die folgenden Punkte reduzieren:

- 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, weil sie sowohl für zukünftige Contributorinnen als auch für die wissenschaftliche Einordnung des Projekts relevant sind. Alle wurden während der Vorbereitungen auf die öffentliche Veröffentlichung gefunden und behoben.

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 Die Testsuite

Der aktuelle Stand der Testsuite umfasst 95 bestehende Tests in drei Testdateien:

- `tests/test_routing.py`: 12 Tests für die Routing-Funktionen (`assign_gpu`, `_resolve_user_experts`, `_select_node`). Alle Tests verwenden gemockte HTTP-Aufrufe und laufen ohne Netzwerkabhängigkeit.

- `tests/test_mcp_validation.py`: 38 Tests für die MCP-Tool-Input-Validation – inklusive der AST-Whitelist-Angriffs- Tests.

- `tests/test_deployment_artifacts.py`: 45 Tests für die Deployment-Artefakte. Diese Suite validiert die Cross-Artifact- Consistency: UID 1001, Port 8000 und die `MOE_*_DIR`-Env-Variablen müssen in Dockerfile, Helm-Chart und Quadlet-Unit übereinstimmen. Die Suite nutzt `helm lint` und `helm template` für Enterprise- und Solo-Profil, prüft Security-Context, EmptyDir-Volumes und OpenShift-Route-vs-Ingress-Switch.

Alle 95 Tests laufen in etwa 1.2 Sekunden durch und sind eine harte Kontrollschicht für die Release-Pipeline. Das Ziel ist nicht „100 % Coverage“, sondern „alle Invarianten abgesichert, die beim nächsten Refactoring brechen könnten“. Die vier Episoden oben wären mit einer besseren Testsuite nicht alle auftrittssicher gewesen; wir werden die Suite bei der Öffentlichkeitsphase erweitern.

Auf was wir stolz sind – und worauf nicht

Stolz: auf die Deterministik des Routings, auf die Lesbarkeit der Codebase, auf die Cross-Artifact-Consistency-Tests, auf die Dokumentation, auf die Transparenz der Lessons-Learned.

Nicht stolz: auf das 70B-RAM-Problem, die SymPy-Lücke, den zu cleveren Scheduler, die Ghost-Keys 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.

14.7 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.

14.7.1 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.

14.7.2 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.

14.7.3 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.

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 4: 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).

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.

14.8 GAIA-Benchmark 2026: Systemreife-Nachweis

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

Bestes Ergebnis. **14/30 = 46,7%** mit dem Template `moe-aihub-free-gremium-deep-wcc` (AIHUB-Frontier-Modell `gpt-oss-120b-sovereign`, 120B Parameter). Dieser Score *übertrifft* den offiziellen GPT-4o Mini-Referenzwert von 44,8%.

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 5: GAIA Validation Set – 5 Runs, 2026-04-25. Level 1–3 je 10 Fragen. Bestes Ergebnis: **14/30 = 46,7%** mit Template `moe-aihub-free-gremium-deep-wcc` (AIHUB-Frontier-Modell `gpt-oss-120b-sovereign`, 120B Parameter) übertrifft GPT-4o Mini (44,8%).

14.8.1 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:

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 6: 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`.

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.

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.

14.8.2 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.

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 7: 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.

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.

14.8.3 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.

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 8: 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.

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).

14.8.4 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.

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 9: 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.

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:

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 10: 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 %).

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: $0.4 \times 0 + 0.6 \times$ 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.

14.8.5 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 %

Unsere Backbone-Modelle (DeepSeek-R1:32b, Qwen3:32b, Llama3.3:70b) liegen als Einzelmodelle im Bereich 11–28 %. MoE Sovereign erzielt mit dem 30b-Balanced-Template **60 % auf GAIA Level 1** (6/10 korrekt) – und übertrifft damit alle gelisteten Einzelmodelle. Der Orchestrator addiert Wert durch:

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

Fehlschläge: Das System scheitert 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 scheitert 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. MoE Sovereign übertrifft beide im besten Lauf (91,7 %), obwohl Memory nicht der primäre Optimierungszweck der Architektur ist, sondern als Nebenprodukt der kontinuierlichen Wissensgraphakkumulation entsteht.

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**	60 %

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 bringt sie im eigenen Benchmark auf 7.6/10, auf GAIA Level 1 auf 60 %. Die Lücke zu Frontier-Modellen auf GAIA (>33 %) liegt bei fehlenden MCP-Integrationen (PDF-Download, arXiv-Lookup) und bei der Kontextverdünnung durch den Merger bei langen Dokumenten. Beides ist implementierbar ohne Modellwechsel.

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.

14.8.6 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*:

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

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.

14.8.7 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.

14.8.8 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.

14.8.9 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):

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 11: 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 12: 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).

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.

14.8.10 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.

14.9 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** (gemma4:31b): 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).

14.10 Deployment-Reife

Transparenz über den Teststatus ist für Reproduzierbarkeit essenziell.

Tabelle 13: 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.

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.

14.11 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

Tabelle 14: 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

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.

Tabelle 15: 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)

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.

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

Der Nachfolger des fehlgeschlagenen `moe-m10-8b-gremium`-Templates (Abschnitt 14.11) 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

Tabelle 16: 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

Alle acht Experten-Modelle sind auf Q4_K_M quantisiert (≤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.

Tabelle 17: 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**

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)

Tabelle 18: 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.)

Bekannte Einschränkung: memory-8turn (6.3 → 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 → 4.8), weil seine Einzel-Turn-Antworten kürzer sind.

Orchestration-Premium

Tabelle 19: 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)

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 20 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 dieselbe Score-Klasse wie cloud-gehostetes GPT-4o mini – bei vollständiger Datensouveränität und ohne Token-basierte Nutzungsgebühren.

Tabelle 20: 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)				

14.13 Gesamtbewertung

14.13.1 Externe Einschätzung

Im Rahmen einer strukturierten Evaluation durch ein KI-gestütztes Review-System wurde MoE Sovereign anhand von sechs Dimensionen bewertet: Architekturreife, Innovationsgrad, Benchmark-Performance, Skalierbarkeit, Reproduzierbarkeit und gesellschaftliche Relevanz. Das Ergebnis:

Dimension	Score (0–10)	Begründung (Kurzform)
Architekturreife	9,0	Deterministisches Routing, 4-Layer-Cache, OCI-Portabilität
Innovationsgrad	9,5	RL-Flywheel + GraphRAG als Compound-Mechanismus neuartig
Benchmark-Performance	8,0	GAIA 46,7 % (über GPT-4o Mini), LongMemEval 91,7 %
Skalierbarkeit	8,0	Solo bis Enterprise-Profil, ein OCI-Image
Reproduzierbarkeit	9,0	Vollständige Infrastruktur als Code, öffentlich verfügbar
Gesellschaftliche Relevanz	8,0	DSGVO-konform, non-kommerziell, CC BY-SA 4.0
Gesamtscore	**8,5 / 10**	Innovationsgrad: sehr hoch

Tabelle 21: Externe KI-gestützte Projektevaluation: Bewertungsmatrix über sechs Dimensionen. Gesamtscore: 8,5 / 10.

Externes Urteil: 8,5 / 10 – Innovationsgrad sehr hoch

Das System kombiniert bekannte Einzelbausteine (MoE-Routing, RAG, RL) auf eine Weise, die – nach aktuellem Kenntnisstand – in dieser Integration und mit diesem Souveränitätsfokus in der Open-Source-Landschaft nicht existiert. Das RL-Flywheel (Routing Telemetry + Thompson Sampling-inspiriertes Scoring + Correction Memory) ist als geschlossener Selbstverbesserungskreislauf *ohne Modell-Finetuning* besonders hervorzuheben. Zukunftspotenzial: **exzellent**.

14.13.2 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 tiefo 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

Ein externer Score von 8,5 / 10 ist ermutigend. Er entspricht unserer eigenen Einschätzung für die Stärken – Architektur, Memory und Reproduzierbarkeit –, aber er darf die strukturellen Grenzen nicht unsichtbar machen: 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.

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 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.

15.2 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 [18] 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.

15.3 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.

15.4 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.

15.5 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.

15.6 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.

15.7 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.

15.8 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 [30]: 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 vier Fehlschläge berichtet: eine Rendering-Regression, die vorübergehend sämtliche Dokumentations- Diagramme unbrauchbar machte; eine SymPy-Injection-Lücke im Fallback-Pfad des MCP-Calculate-Werkzeugs; eine SQLite-Schreibproblematik unter Concurrent Access; und einen zu cleveren ersten Scheduler. Alle vier sind behoben, und die Dokumentation jeder Episode 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 unter der Lizenz CC BY-SA 4.0 [12] auf Philipp Horns Entwicklerseite und im dazugehörigen Git-Repository verfügbar sein. 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, April 2026

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*. https://arxiv.org/abs/2311.12983. Accessed 2026-04-25. 2023. arXiv: 2311.12983 [cs.CL].

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

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

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

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

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

[30] 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 [24].

Traceparent Der HTTP-Header aus dem W3C-Trace-Context-Standard [27] 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.

Kategorie	Zweck
`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

Dieses Whitepaper und die begleitende Software stehen unter der Lizenz *Creative Commons Attribution-ShareAlike 4.0 International* [12] bzw. einer kompatiblen Open-Source-Softwarelizenz. 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).

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.